Profiling, Analysis, and Performance

Adrian Jackson
EPCC
adrianj@epcc.ed.ac.uk
@adrianjhpc

Analysis and Profiling

- Analysis and browsing
  - Code browsing
  - Code analysis
- Performance profiling
  - Standard profilers
  - Timing by hand

Code browsing and analysis

- By “browsing and analysis” we mean
  - Reading and understanding code
  - Perhaps reformatting to make it more comprehensible
  - Analysing program structure
  - Essentially, understanding what it does and how to change it
- Understanding a 20 line program is quite easy
  - Reading it should probably be enough
- Understanding a 20,000 line program is almost impossible without tool help
  - Code browsers or IDEs are essential for this

Code browsing tools

- Understand source code syntax
  - Perhaps display code using “syntactic highlighting”
- Provide call-tree or class hierarchy views
  - Simple overview of program structure and control flow
- Declaration and reference lookups
  - Refactoring tools
- Trace variables through function/subroutine calls
  - Key data flow: when and how are data read? changed?
- Often offer customisable code formatting / reformatting
  - Very useful for later comprehensibility

Browsing tools

- OO-Browser
  - http://sourceforge.net/projects/oo-browser/
  - Browser/analyser for C, C++, CLOS/Lisp, Eiffel, Java, Objective-C, Python, Smalltalk
  - Can work as part of Emacs under X Window or MS Windows
  - Free download, commercial support
- f90browse
  - Sun WorkShop (old Cray Xbrowse)
  - Browser/analyser for Fortran 90/77
  - Uses compiler information files (CIF) produced by f90 –db
- F90tohtml
  - Creates website based on source code
  - http://code.google.com/p/f90tohtml/
- Cleanscape Xlint
  - http://www.cleanscape.net/
  - Based on Xlint output from FortranLint product
  - Displays call tree, source code, Xlint output
- OpenGrok
  - http://opengrok.github.io/OpenGrok/
  - A fast search engine for programs
  - Read-only web interface for version control systems history log of a file
  - A stand alone GUI that can be used

Other browsing tools

- IDEs very useful
  - Eclipse, NetBeans, etc..
  - Usefulness depends on code and language use
  - Good refactoring and browsing tools
  - On-fly checking through compilation
- Your favourite debugger
- Debuggers don't have to be used for buggy code
- Many debugger features very useful
  - Breakpoint setting
  - Single stepping
  - Variable monitoring
  - Run-to-mark
- GUI-driven debuggers best, but command line can be useful too

Code analysis tools

- Less than a full browser
  - Typically command-line not GUI driven
  - Kind of commands to call from your Makefile
  - Possibly build into testing structure
- Offer “second opinions” on code structure
- Can offer more in-depth analysis than browsing tools

```
#include<stdio.h>

int main(){
  int i, a;

  for(i=0; i<100; i++){
        a += i;
  }

  printf("a: ", a);

  return 0;
}
```

Code analysis tools

- lint
  - Checks C program “correctness”
  - Identifies semantic errors, poor programming practices, portability and robustness issues
  - Splint is new open source version: http://www.splint.org/download.html

```
(8) warning: variable may be used before set: a

function returns value which is always ignored
    printf

too many arguments for format
    printf                assemloop.c(11)
```

- flint – cleanscape has a version
  - Same for Fortran
- ftnchek - http://www.dsm.fordham.edu/~ftnchek/ - F77
  - Similar “correctness” checker
  - Identifies semantic errors
- cscope – open source
  - Interactive browsing/analysis tool for C
- cppcheck, splint, cscout
- lint4j, jlint, findbugs, any IDE you are using

Code analysis tools

- cflow – gnu http://www.gnu.org/software/cflow/
  - Generates function call graph with code line numbers
  - Works on C source or (non-stripped) object files

```
1  main: int(),
<assemloop.c 3>
2    printf: <>
```

- cxref – open source http://www.gedanken.org.uk/software/cxref/
  - "Big brother" of cflow
  - Includes external function calls
  - Also includes variables etc.

```
NAME            FILE          FUNCTION        LINE
__func__        modassemloop. main               1*
a               modassemloop. main               3*     6=
i               modassemloop. main               2*     5=     5     6     7=
main            modassemloop. ---                1*
```

Code analysis tools cont.

```
assemloop.c:
NAME          FILE         FUNCTION       LINE
BUFSIZ         stdio_iso.h   ---            105*
EOF            stdio_iso.h   ---            133*
FILE           stdio_iso.h   ---            75-
FILENAME_MAX    stdio_iso.h   ---            137*
FOPEN_MAX       stdio_iso.h   ---            136*
L_ctermid       stdio.h       ---            126*
L_cuserid       stdio.h       ---            127*
L_tmpnam        stdio_iso.h   ---            144*
NULL            stdio_iso.h   ---            101*
P_tmpdir        stdio.h       ---            133*
SEEK_CUR        stdio_iso.h   ---            140*
SEEK_END        stdio_iso.h   ---            141*
SEEK_SET        stdio_iso.h   ---            139*
TMP_MAX         stdio_iso.h   ---            142*
_ALIGNMENT_REQU isa_defs.h    ---            308*
_BIG_ENDIAN     isa_defs.h    ---            297*
_BIT_FIELDS_HTO isa_defs.h    ---            300*
_CHAR_ALIGNMENT isa_defs.h    ---            303*
_CHAR_IS_SIGNED isa_defs.h    ---            302*
_CONSOLE_OUTPUT isa_defs.h    ---            316*
_DMA_USES_VIRTA isa_defs.h    ---            314*
_DOUBLE_ALIGNME isa_defs.h    ---            307*
_FILEDEFED      stdio_iso.h   ---            74*
_FILE_OFFSET_BI feature_tests ---            96*   98    98
                stdio_iso.h   ---            52    87
                stdio.h       ---            90   141   297
```

Code analysis/checking

- Compiler checking
  - -Wall for gnu compilers
  - -fcheck=all – gfortran
  - -pedantic

```
adrianj@gateway:~$ gcc -Wall -pedantic -o main main.c
main.c: In function âmainâ:
main.c:10:3: warning: too many arguments for format [-Wformat-extra-args]
```

  - Use different compilers
    - Cray, Nag, etc…
    - Try something other than gnu if possible
    - Different compilers have different checking functionality etc…

```
adrianj@hector-xe6-10:~> craycc -o test test.c
        a += i;
CC-7212 craycc: WARNING File = test.c, Line = 7
  Variable "a" is used before it is defined.
Total warnings detected in test.c: 1
```

Performance analysis

- Once you've a working code you may want to optimise it
  - *"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified"* — Donald Knuth
  - *"The First Rule of Program Optimization: Don't do it. The Second Rule of Program Optimization (for experts only!): Don't do it yet."* — Michael A. Jackson
- First step is Profiling
  - Discover if it's efficient or quick
- Highlight areas that are computational expensive
  - Often not where you'd expect
- Focus work on areas where impact will be biggest
  - Need hard data to make these choices
- Profiling tells you *where* the code is spending all of its time
  - Optimisation without profiling is leaping before looking!
  - Avoid working on speeding up part of an algorithm if the application only spends small part of its time there

Code profiling

- Code profiling is the first step for anyone interested in performance optimisation
- Profiling works by instrumenting code at compile time
  - Thus it's (usually) controlled by compiler flags
  - Can reduce performance
- Standard profiles return data on:
  - Number of function calls
  - Amount of time spent in sections of code
- Also tools that will return hardware specific data
  - Cache misses, TLB misses, cache re-use, flop rate, etc…
  - Useful for in-depth performance optimisation

Standard profilers

- Standard Unix profilers are prof and gprof
- Many other profiling tools use same formats
- Usual compiler flags are **`-p`** and **`-pg`**:
  - **`f90 -p mycode.F90 -o myprog`** for prof
  - **`cc -pg mycode.c -o myprog`** for gprof
- When code is run it produces instrumentation log
  - **`mon.out`** for prof
  - **`gmon.out`** for gprof
- Then run prof/gprof *on your executable program*
  - eg. **`gprof myprog`** (*not* **`gprof gmon.out`**)

Standard profilers

- **`prof myprog`** reads **`mon.out`** and produces this:

%Time	Seconds	Cumsecs	#Calls	msec/call	Name
32.4	0.71	0.71	14	50.7	relax_
28.3	0.62	1.33	14	44.3	resid_
11.4	0.25	1.58	3	83.	__f90_close
5.9	0.13	1.71	1629419	0.0001	_mcount
5.0	0.11	1.82	339044	0.0003	__f90_slr_i4
5.0	0.11	1.93	167045	0.0007	__inrange_single
2.7	0.06	1.99	507	0.12	_read
2.7	0.06	2.05	1	60.	MAIN_

Standard profilers

- **`gprof myprog`** reads **`gmon.out`** and produces something very similar
- gprof also produces a program calltree sorted by inclusive times
- Both profilers list all routines, including obscure system ones
  - Of note: **`mcount`**(), **`_mcount`**(), **`moncontrol`**(), **`_moncontrol`**() **`monitor`**() and **`_monitor`**() are all overheads of the profiling implementation itself
  - **`_mcount`**() is called every time your code calls a function; if it's high in the profile, it can indicate high function-call overhead

Java Profilers

- Java also has profiling tools
  - jvisualvm
  - yourkit
  - Jprofiler
  - Jensor
- Primarily connect to code in JVM
  - Profile memory usage and time spent in code section

TAU – Parallel Profiler

- TAU – Tuning and Analysis Utilities

  http://www.cs.uoregon.edu/research/tau/home.php
  - Portable profiling and tracing toolkit for performance analysis of programs written in Fortran, C, C++, Java, Python
  - Primarily for parallel (or multi-threaded) programs
- Heavier weight that standard profilers
  - Produces its own compiler wrappers
  - Based on code instrumentation
  - Instrumentation can be inserted in the source code automatically (like prof and gprof), dynamically, at runtime in the Java virtual machine, or manually using the instrumentation API
  - Can also access hardware counters through PAPI or PCL
  - Can be run through Eclipse, either Java TAU version or through Eclipse Parallel Toolkit
- Graphical visualisation tool
- Memory profiling built in

Vampirtrace

- Vampirtrace
  - Profiling library for MPI communications
  - Also allows instrumenting of code
  - Produces trace files in Open Trace Format (OTF)
  - Can be viewed in KOJAK, Vampir, TAU, etc…

- Lots of other examples
  - Craypat, …

Hardware Profilers

- PAPI – Performance API
  - http://icl.cs.utk.edu/papi/index.html
  - Set of tools built on it which can give you the info you want (i.e. perfsuite, TAU, KOJAK, etc…)
- AMD Tool - CodeXL
  - http://developer.amd.com/tools-and-sdks/heterogeneous-computing/codexl/
  - Hardware counters and general profiling
  - Open source
- Intel Tool – Vtune
  - http://software.intel.com/en-us/intel-vtune-amplifier-xe /
  - Costs money
  - Includes old performance tuning utility (PTU)

Timing by hand

- Instead of profiling a whole code, sometimes you want to time just one or two routines
- Add explicit timing calls to your code
  - More control over overheads
  - Less impact on overall code performance
  - Switch in and out with conditional compilation flags
- However, beware portability!
  - There is generally no such thing as a “standard”, “portable” timing function or routine
  - Resolution very important for timers
  - HPC platforms in particular have their own ways of timing
    - Platform X will have a good, high-resolution timer but it will be completely non-portable to platform Y

Portability of timing functions

- In Fortran, two fairly standard library functions:
  - **`dtime()`**
    - Returns elapsed time since last call to **`dtime`**
  - **`etime()`**
    - Returns elapsed time since start of execution
  - NB: for parallel or multithreaded systems, read the manual pages carefully to check what the implications are
    - Although this is no guarantee... A quote from a certain HPC system:

      `“For etime, the elapsed time is:`

      `- For multiple processors: the wall-clock time while running your program”`
    - **This lies**; it actually sums time over all threads; *not* what you want...

Portability of timing functions

- `system_clock` – returns elapsed wallclock time
  - Takes 3 integer arguments:
    - **count, intent(out)**
    - **count_rate: intent(out)**
    - **count_max: intent(out)**
  - counts time at a rate of count_rate counts per second up to count_max
  - call system_clock(count, count_rate, count_max)
  - Elapsed time = (count2-count1)*1.0/count_rate
- Best function for C is probably
  - `gettimeofday(struct timeval *tp, void *);`
  - Again, a library function
  - Need to `#include <sys/time.h>`
- Anyone writing MPI programs can use
  - `double MPI_Wtime(void)`
    `double precision MPI_WTIME()`
  - Defined parts of the MPI 1 standard library
    - Not always implemented...
  - Return times local to the calling process

Other timing commands

- Simple Unix/Linux solution for whole program runs is '**`time`**':

```
$ time ./mg-test
[program output]
real    0m3.832s
user    0m2.330s
sys     0m0.210s
```

- Always check accuracy of your timers
  - e.g. above, real (elapsed) time is +/- 1s while user and system (CPU) times are +/- 0.01s
- Portable timing routines are good examples of things to put in your own utility libraries

Summary

- Understanding big codes is hard
  - But there are plenty of helpful tools
  - Learn to use them!
- Profiling: look before you leap
  - Don’t attempt any optimisation work until you’ve profiled your code
  - Learn to use and understand prof and gprof
  - More hardware specific data is available
  - Timing by hand is very useful, but beware portability of timers

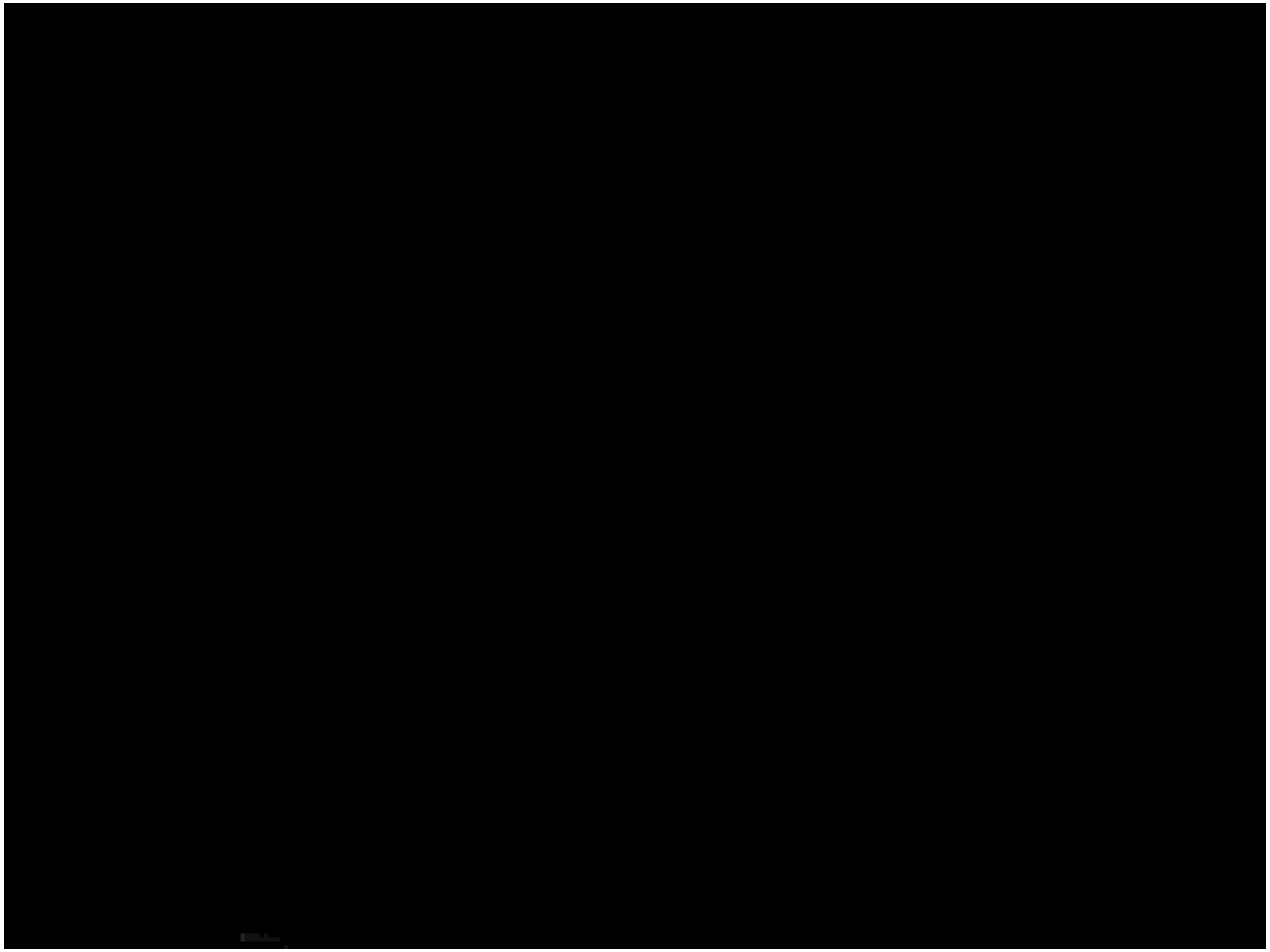
Overview

- Displaying and manipulating performance data from HPC code
  - Easy to draw a simple graph
  - “Real” story not always obvious from basic graphs
  - Need to understand data and what you want to show
- Basic example, principles apply to all cases
  - Always important to understand that data can be shown in many different ways
  - Errors should always be considered
    - Performance variation

Visualisation of Data

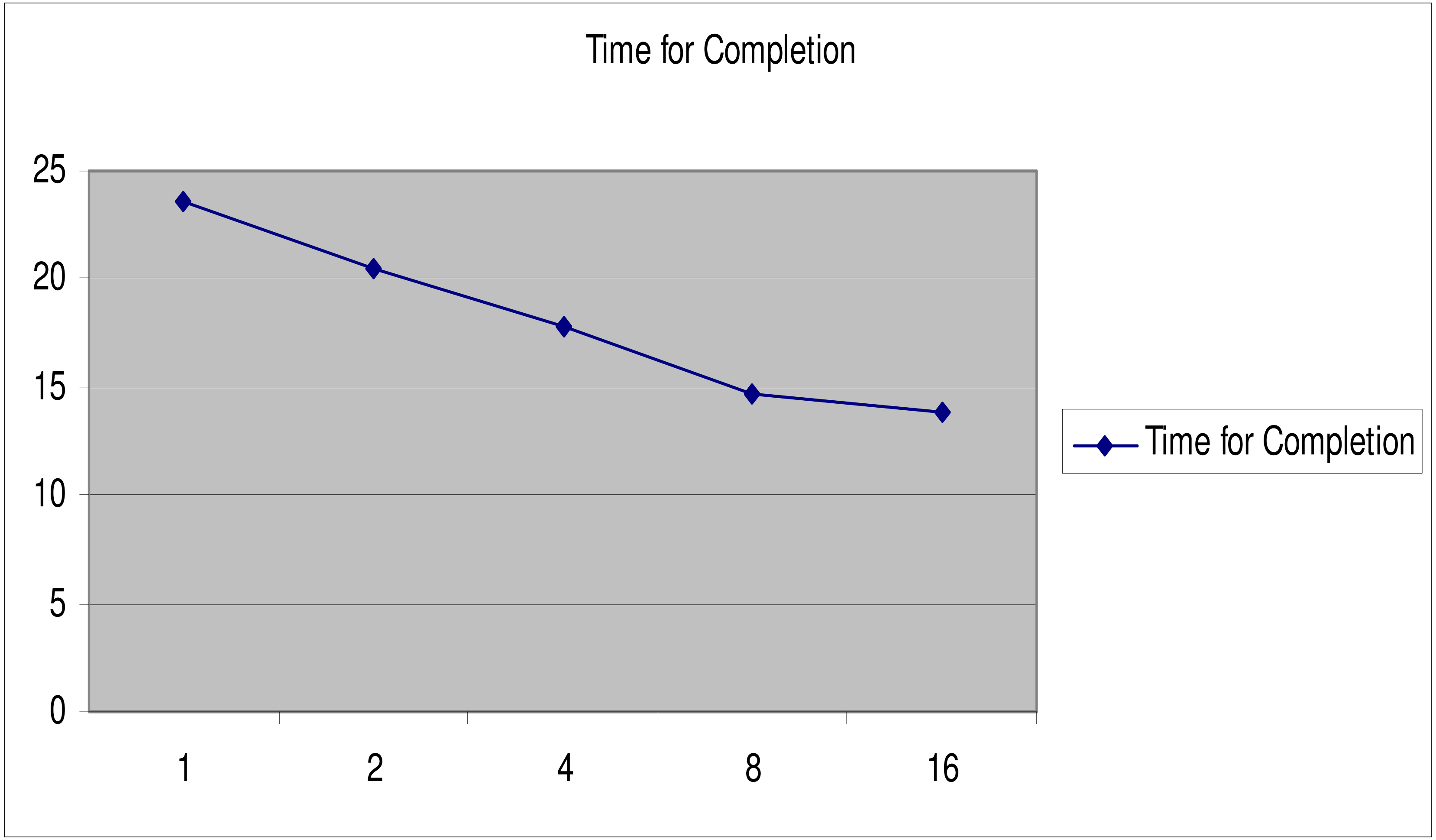
- Once HPC code run data must be analysed
  - Often most important step
  - Visualisation can be useful
- Tables, graphs, pictures, animations, etc…
  - Many ways to show data, each useful for particular scenarios
  - Always focus on what you want to display
  - Often, hardest part of visualisation is choosing which technique/display method to use
- We often focus on performance data
  - How data collected is important
  - Errors must be considered here too

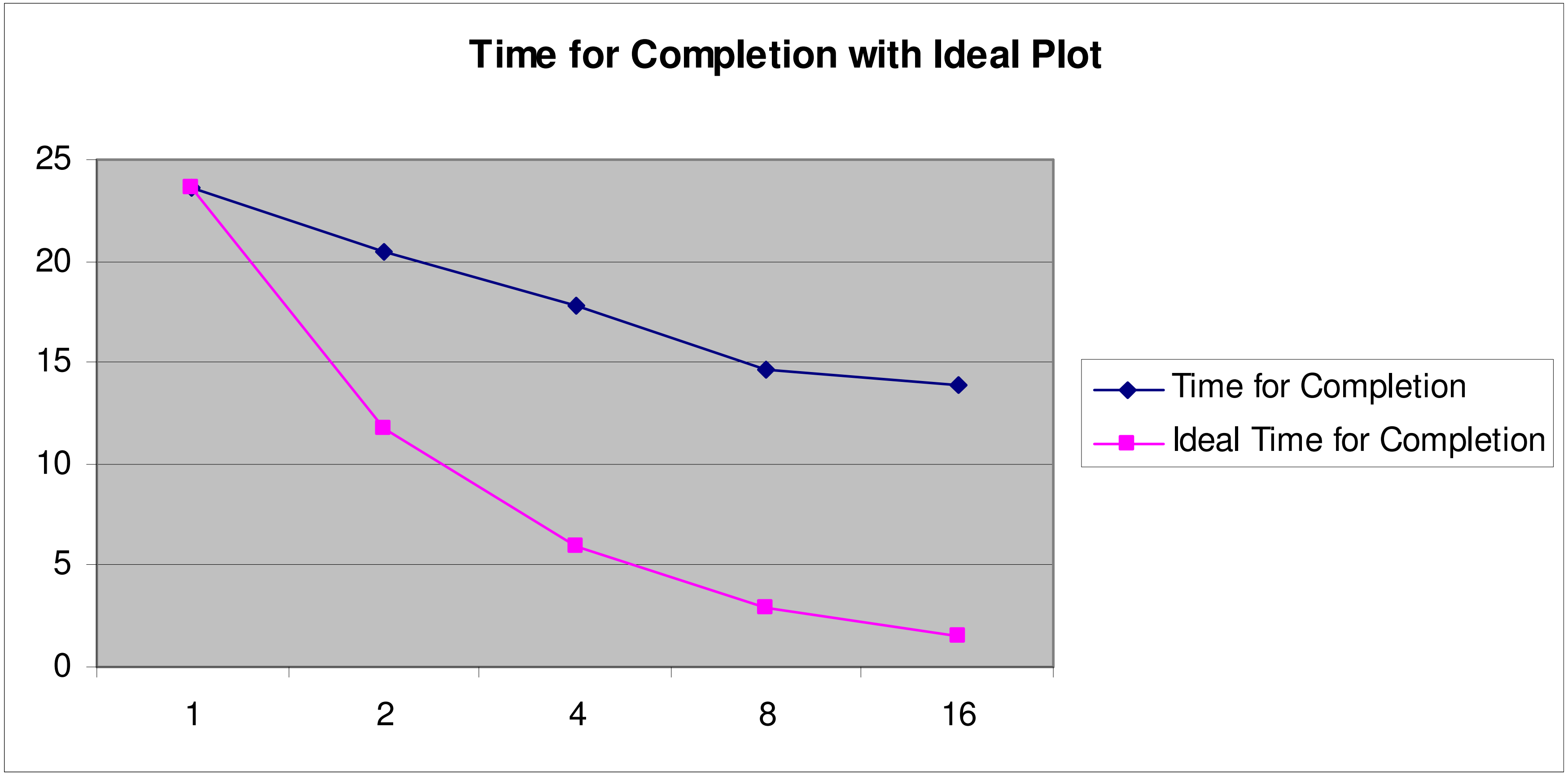
Visualisation cont…

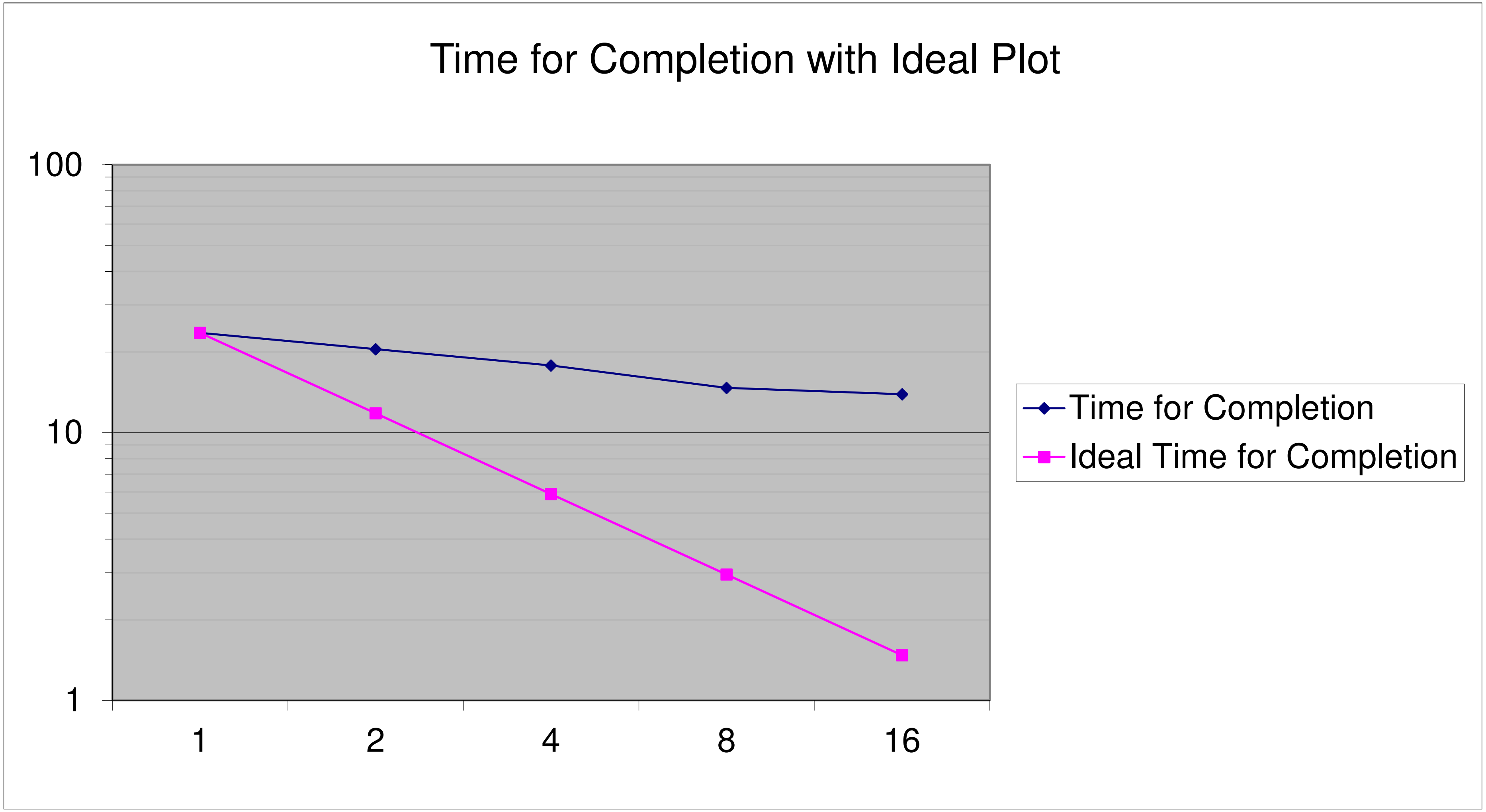


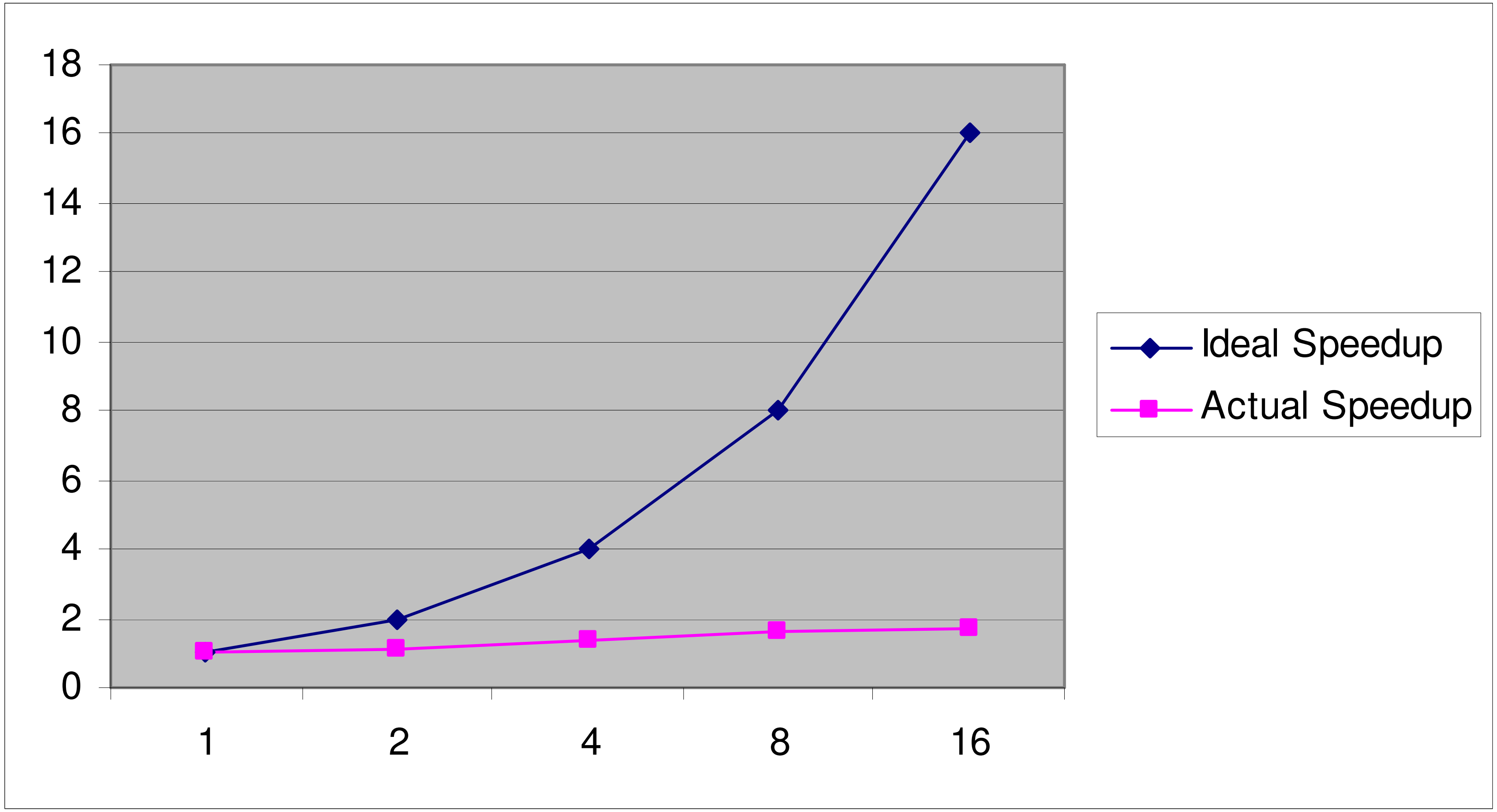
Performance Graph Example

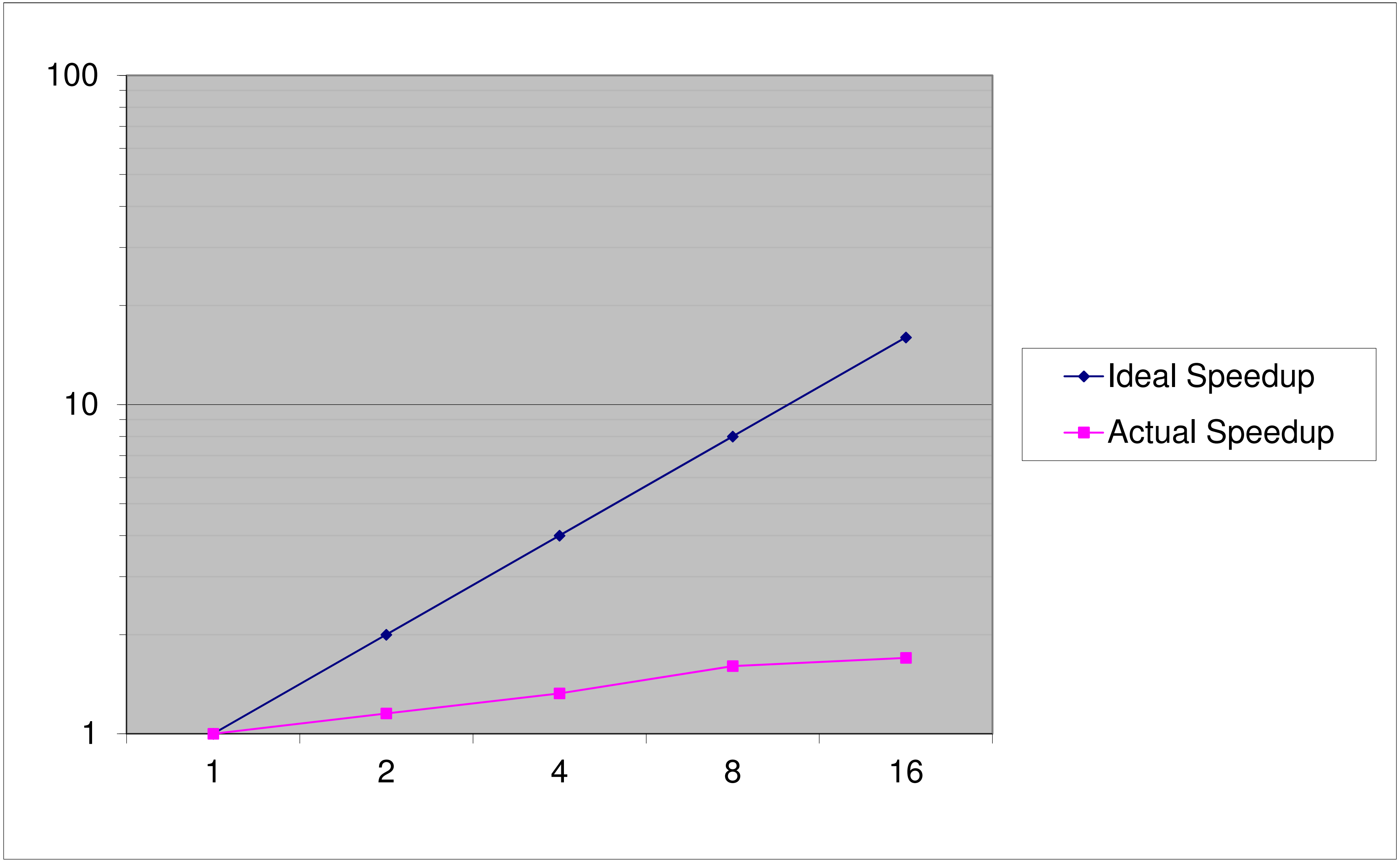
Number of Processors	Total Runtime (seconds)
1	23.6
2	20.5
4	17.8
8	14.7
16	13.9

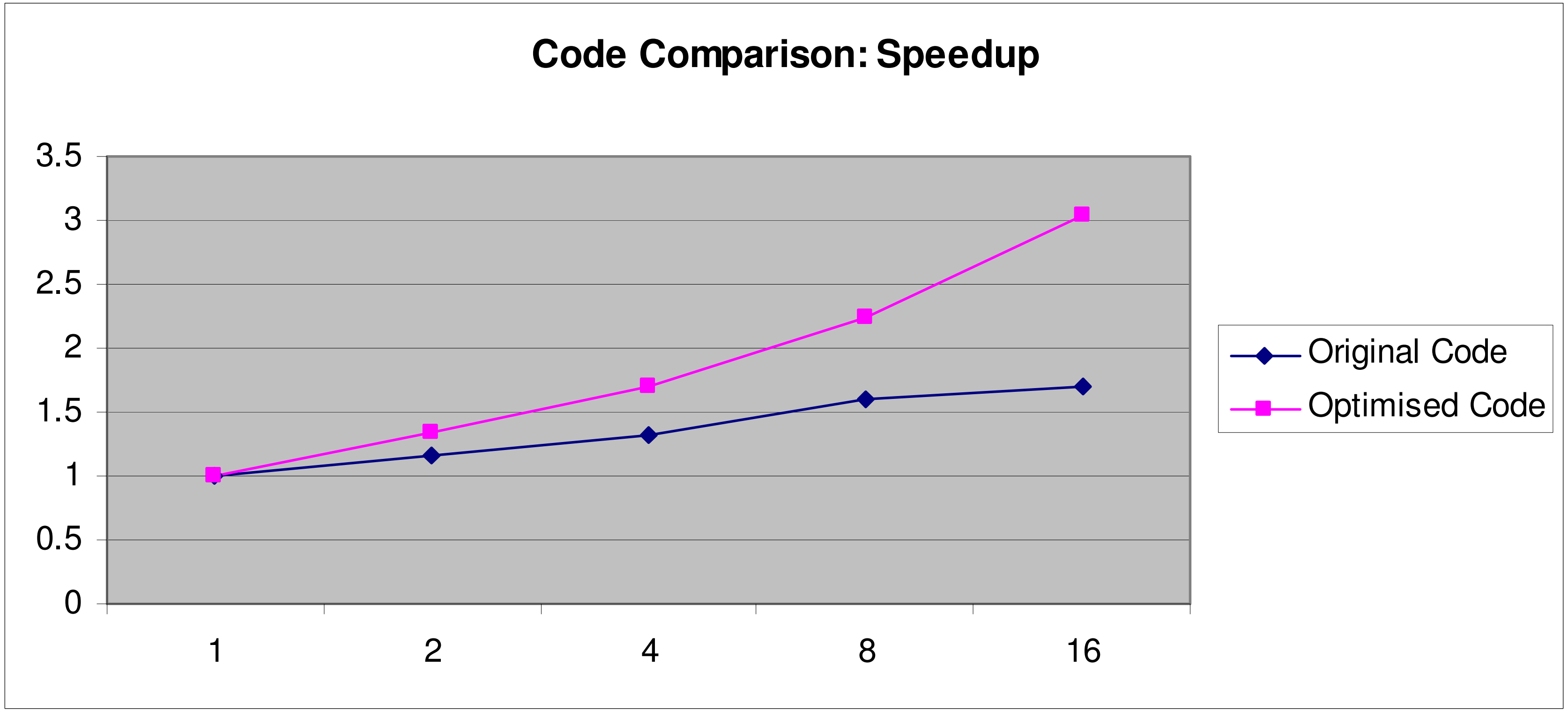

Time for Completion
25
20
15
10
5
0
1
2
4
8
16
Time for Completion

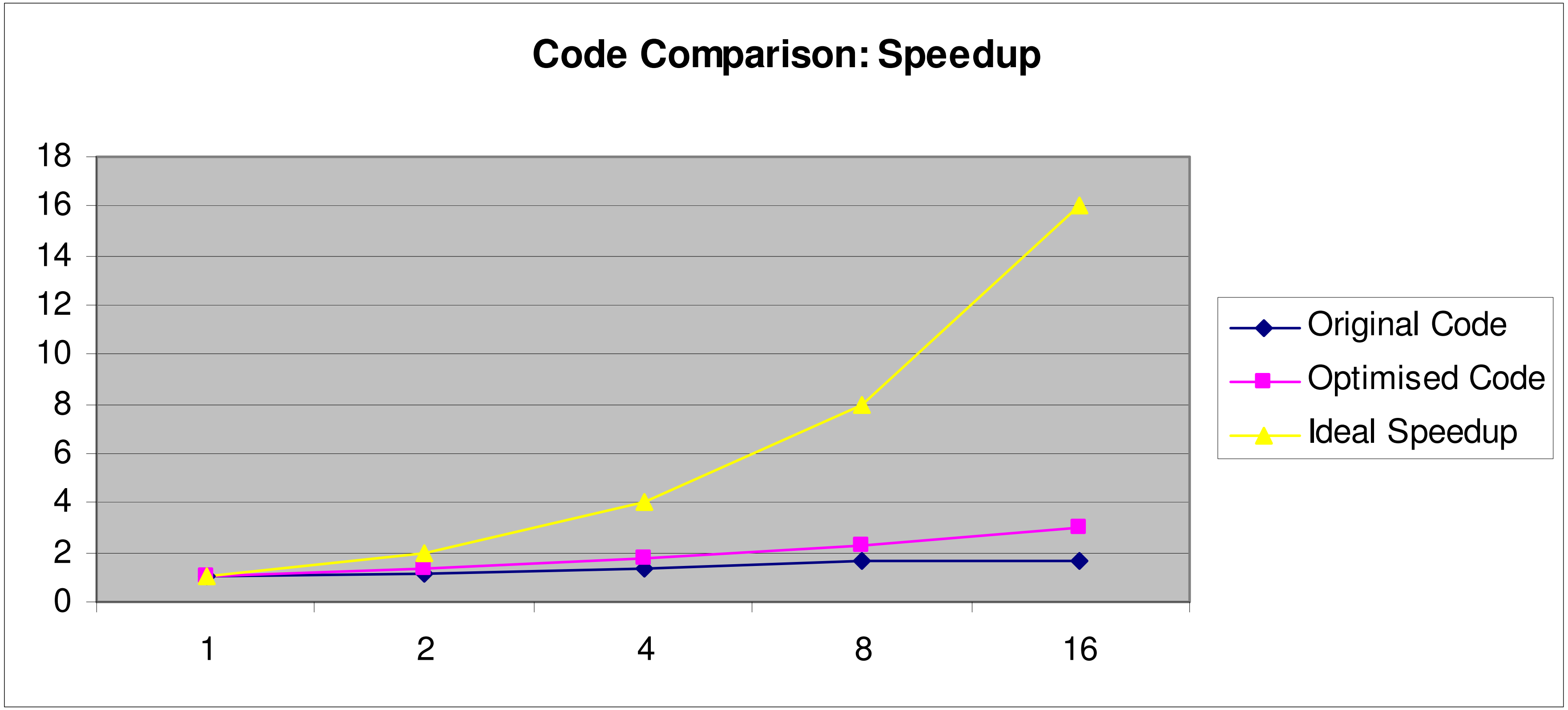

Time for Completion with Ideal Plot
25
20
15
10
5
0
1
2
4
8
16
Time for Completion
Ideal Time for Completion


Time for Completion with Ideal Plot
100
10
1
1
2
4
8
16
Time for Completion
Ideal Time for Completion

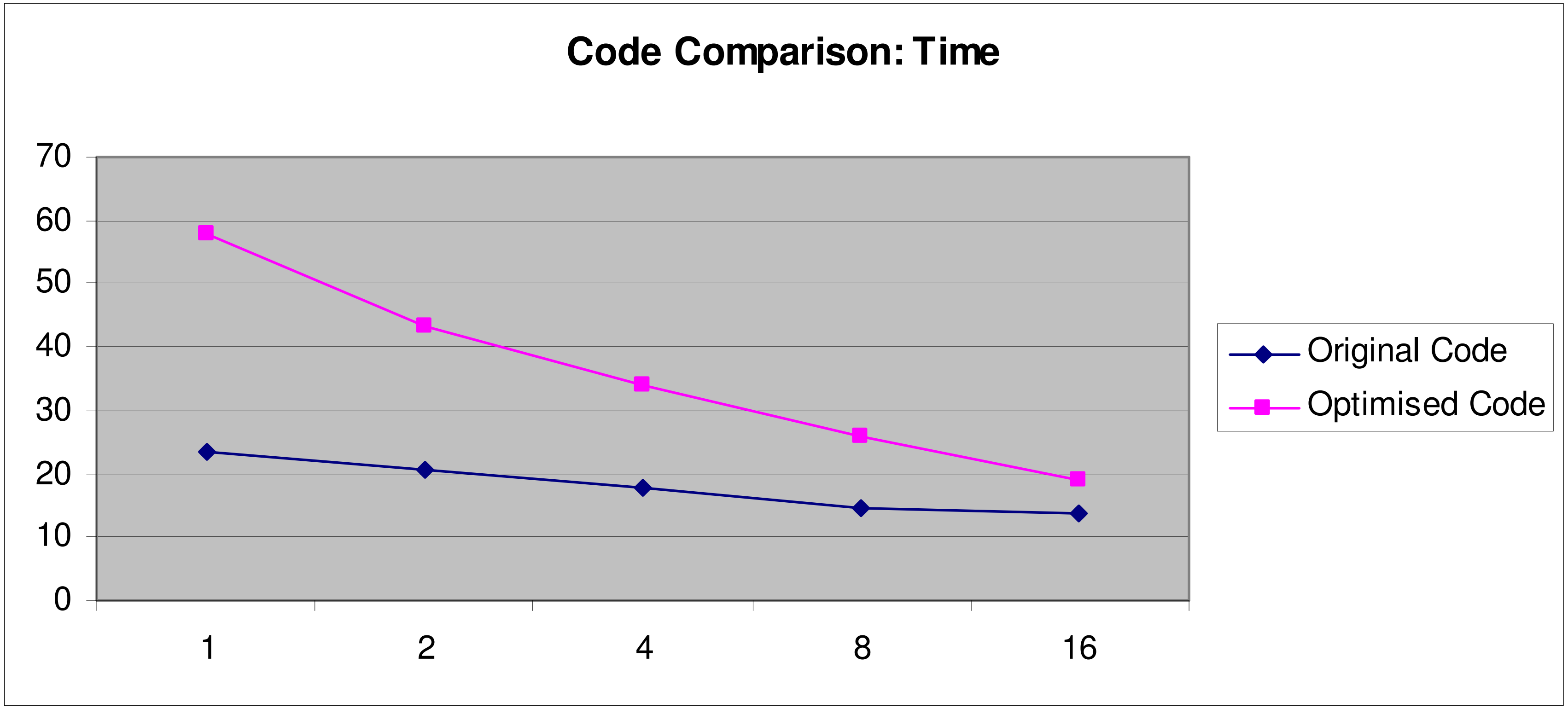

18
16
14
12
10
8
6
4
2
0
1
2
4
8
16
Ideal Speedup
Actual Speedup


100
10
1
1
2
4
8
16
Ideal Speedup
Actual Speedup


Code Comparison: Speedup
3.5
3
2.5
2
1.5
1
0.5
0
1
2
4
8
16
Original Code
Optimised Code


Code Comparison: Speedup
18
16
14
12
10
8
6
4
2
0
1
2
4
8
16
Original Code
Optimised Code
Ideal Speedup


Code Comparison: Log Speedup
100
10
1
1
2
4
8
16
Original Code
Optimised Code
Ideal Speedup


Code Comparison: Time
70
60
50
40
30
20
10
0
1
2
4
8
16
Original Code
Optimised Code

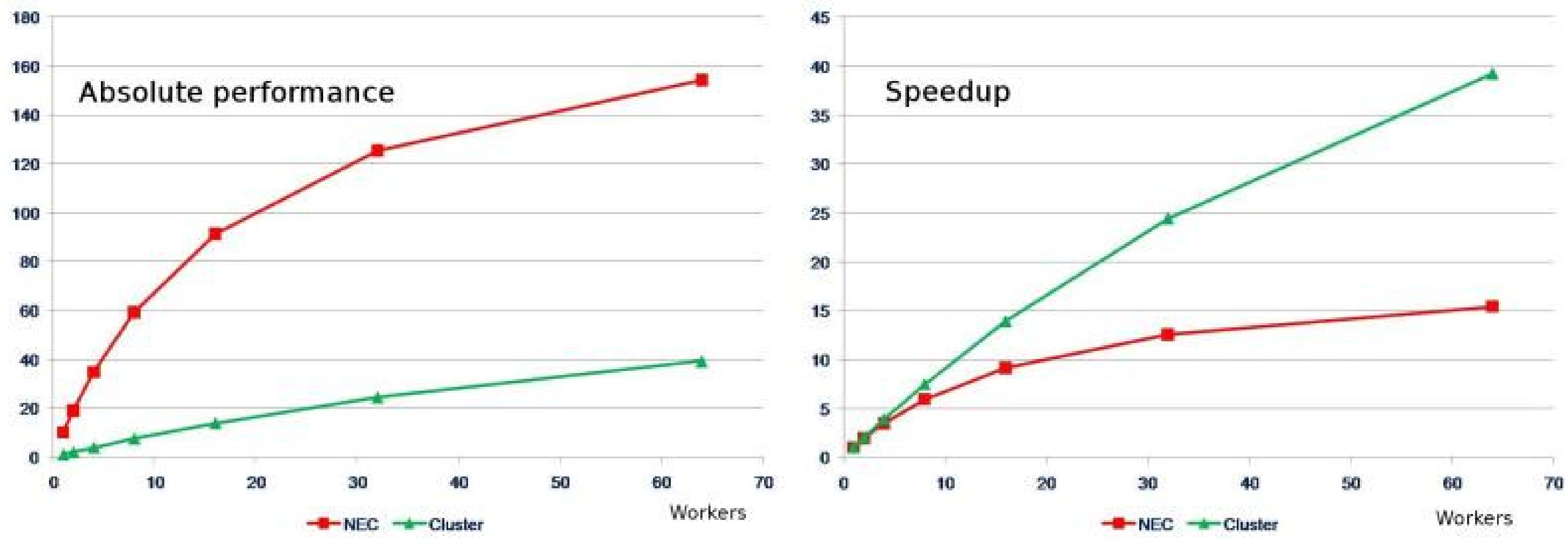
Number of Processors	Original Code Runtime (seconds)	New Code Runtime (seconds)
1	23.6	57.9
2	20.5	43.2
4	17.8	33.9
8	14.7	25.8
16	13.9	19.1

Fooling the masses: Georg Hager

- http://blogs.fau.de/hager/category/fooling-the-masses/
- David H. Bailey: “Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers” (1991)
  - Quote only 32-bit performance results, not 64-bit results.
  - Present performance figures for an inner kernel, and then represent these figures as the performance of the entire application.
  - Quietly employ assembly code and other low-level language constructs.
  - Scale up the problem size with the number of processors, but omit any mention of this fact.
  - Quote performance results projected to a full system.
  - Compare your results against scalar, unoptimized code on Crays.
  - When direct run time comparisons are required, compare with an old code on an obsolete system.
  - If MFLOPS rates must be quoted, base the operation count on the parallel implementation, not on the best sequential implementation.
  - Quote performance in terms of processor utilization, parallel speedups or MFLOPS per dollar.
  - Mutilate the algorithm used in the parallel implementation to match the architecture.
  - Measure parallel run times on a dedicated system, but measure conventional run times in a busy environment.
  - If all else fails, show pretty pictures and animated videos, and don’t talk about performance.

Fooling the masses: Georg Hager

- **Stunt 1: Report speedup, not absolute performance!**



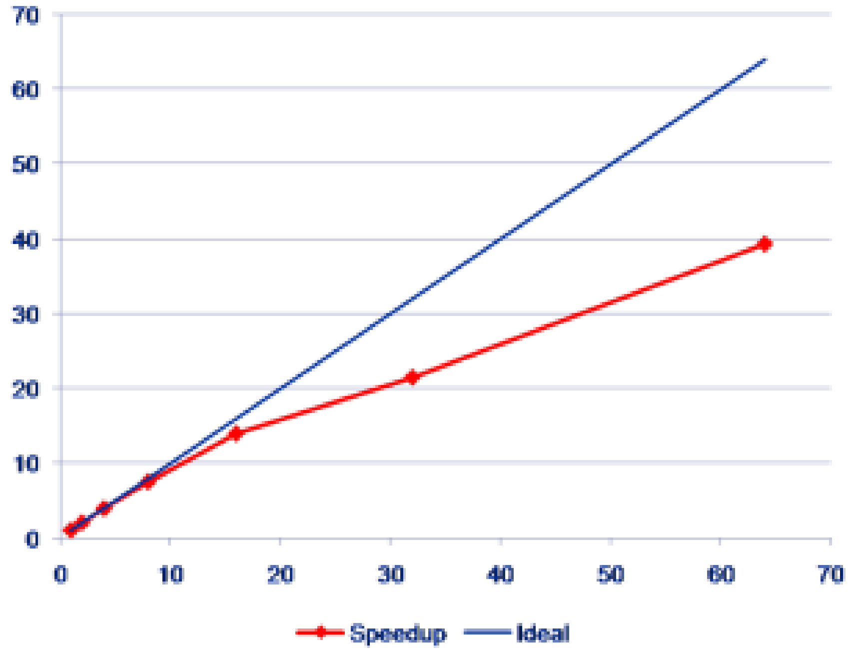
Fooling the masses: Georg Hager

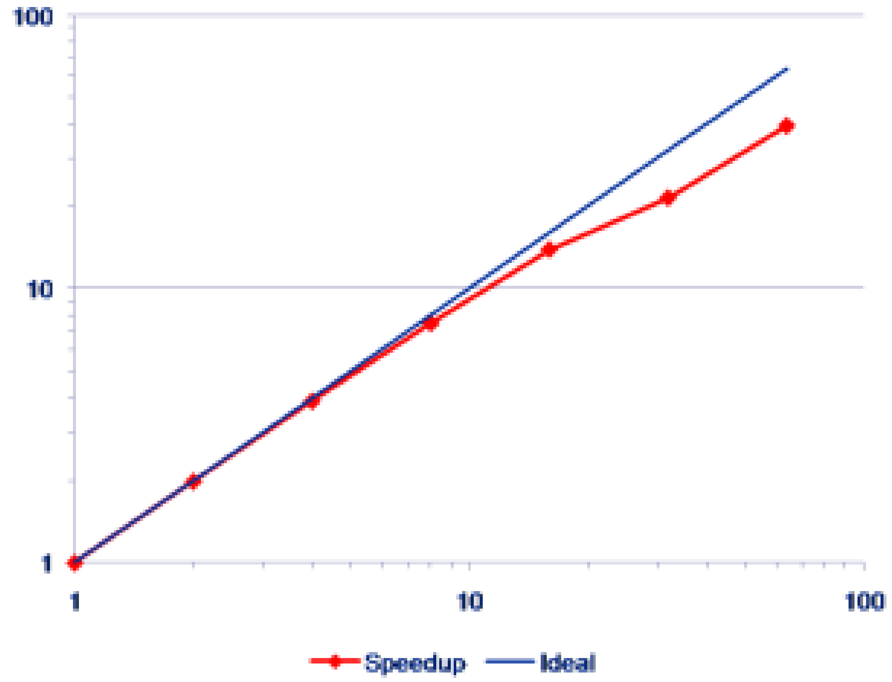
- **Stunt 2: Slow down code execution!**

$$S(N) = \frac{s + (1-s)N^{\alpha}}{s + (1-s)N^{\alpha-1} + \boxed{c_{\alpha}(N)}}$$

Fooling the masses: Georg Hager

- **Stunt 3: The log scale is your friend!**





Reporting Performance

- Context:
  - Always important to report details of performance data collection:
    - Machine hardware
    - OS
    - Third party tools
    - Software version
    - Technique for collecting data (average, fastest, slowest)