

# **Role of Free/Open Source Software and Open Standards in Scientific Computing**

Manoj Warriar

BTDG, Bhabha Atomic Research Center, Trombay, Mumbai, India – 400085

**© IOSN South Asia Node (CDAC Chennai) - 2007**

This work is released under the Creative Commons Attribution 2.5 License. For full details of the license, please refer

<http://creativecommons.org/licenses/by/2.5/legalcode>

## **Abstract**

Free/open source operating systems and software constitute a major portion of the top 500 supercomputers [1]. Nearly 50 % of the top 500 supercomputers are used for Academic / Technological research which involves advanced scientific computations. Computational science (as opposed to computer science) is a relatively new methodology for scientific research compared to usual theoretical and experimental methods. It faces a crisis of confidence amongst the scientific community [2]. Adoption of better software engineering practices by researchers [3] and open standards for evaluation of research publications can help in alleviating many of the concerns. This paper puts in perspective the role of Free/Open Source software (FOSS) and Open Standards in scientific computing. It makes a case for the use of such software and more open standards in scientific software development and in the refereeing process of scientific publications. A need for introductory courses at the graduate and post-graduate levels on basic computational science and the available software using FOSS tools on a GNU/Linux platform is presented within the Indian context so that trained manpower for scientific computations is available.

## Table of Contents

Introduction.....	3
Defining open source:.....	3
Defining open standards:.....	5
What is Computational Science?.....	6
Layout of this paper:.....	8
FOSS, Computational Science and the top 500 supercomputing sites.....	10
Current status of Computational Science.....	12
Role of FOSS and Open Standards in overcoming the current limitations of Computational Science.....	16
Code Verification and Validation.....	16
Quality control during the code development process.....	17
Better means of Evaluation and Distribution of Scientific Work.....	18
FOSS Tools for Computational Science.....	20
GNU Compiler collection (GCC).....	21
GNU Scientific Library (GSL).....	22
Debugging Code.....	23
GNU Make.....	25
GNU profiler – gprof.....	27
Version control systems (cvs, svn).....	27
Parallel programming FOSS tools.....	28
Comparison of FOSS and Proprietary software for scientific computing.....	29
A Case for Introducing GNU-Linux in the Syllabus at the Universities.....	30
Summary and Conclusions.....	32

# Introduction

## ***Defining open source:***

All software on a computer, for example, a web browser, e-mail client, text editor, media player, games, etc, are the end result of someone writing a computer program and compiling it to create an executable. This executable is known by its "software name", for example, Firefox, Thunderbird, etc. The computer program is written to carry out tasks which allows a user to interact with the computer. The technical details of the underlying processes associated with making a computer carry out various tasks is hidden from the average user, usually, behind a good looking screen. If the user does not have access to the computer program, i.e. to the source code, the program is classified as **closed source**. On the other hand if the user has access to the computer program it is classified as **open source**.

From the above description, a non-technical person who does not have sufficient programming knowledge may be under the impression that it does not matter if the program source is open or closed. This is not true, since, an open source program has the possibility of being reviewed by several people with the capability of verifying and validating the program and detecting undesirable parts of the code like spy ware, viruses, etc. For a technical person, open sources allow the possibility of adaptation of the code to one's needs. This is especially useful in the field of scientific research wherein one wants to not only run a computer program (hitherto called code) but also adapt and optimize it for ones needs. Over and above having access to the source code, if one also has the freedom to distribute the software, it will help in

- Increasing contacts and collaborations; In non-professional parlance this means that one will have the freedom to share any useful software with ones friends and colleagues thereby being more friendly / neighborly and probably receiving similar help or assistance which leads to a nice professional and social environment. It also helps in reducing the gradients between the haves and the have-nots leading to a more level playing field. For example, researchers at badly funded universities would have the same software as those used by researchers in well funded institutes if the computational software they used is Free and Open. Check out [4] for a detailed real life story of how some vendors restrict the freedom of people using their software – in this case leading to the development of the free software movement by Richard Stallman.
- Improving the end-product because more minds can work on debugging and optimizing the code. This is one of the most strong arguments for having an open-source development environment. Firefox, Thunderbird are examples of

such a development process. I will dwell more on this in the context of open standards for reviewing computational science work.

- Spawning various forks of the code, adapted to do different things. This is again arising out of the fact that several diverse minds with diverse aims have access to the source code and will look upon the tool from their points of view leading to the same code being forked and developed to achieve their diverse ends. Alternatively, the same code can be used to do a wide variety of things. For example, konqueror [5] can be used as both, a web-browser and a file manager. In scientific research, this is more the norm wherein a code developed is then used to study different problems of interest; for example see LAMMPS (Large-Scale Atomic/Molecular Massively Parallel Simulator) [6] available under the GNU General Public License (GPL) [7] using which atomic simulations and also particle simulations at the mesoscale or continuum levels are possible and have been modeled using a single code. Scientific software released under the GNU GPL abounds in such examples.

This freedom to access the code sources and in distributing the code is the freedom referred to by the "F" in FOSS. There are no restrictions on charging for the distribution as long as the persons receiving the code also are given the same freedom. For a more detailed introduction and discussion of the pros and cons of FOSS, see [8][9].

Many people point out that this freedom is different from the freedom implied by "Free Beer". Because of the nature of the GPL [7] any software available under this license is usually also available free of cost as in "free beer". It must be pointed out that from a point of view of a badly funded researcher in an university and also from a well funded researcher in a well to do Institute, the "free beer" aspect is also welcome. The reasoning being that the badly funded researcher can use the free tools for his work and perhaps contribute by being one of the pairs of eyes that help in debugging or hands that help in creating new forks or adaptations of the software. How will the researcher in a well funded institute benefit from the free – beer – software? In most Institutes there are committees like (i) need aspects committee, (ii) computer committee and (iii) purchase committee, for no doubt very well delineated functions. A researcher has to pass his needs through all these three committees and based on their recommendations buy the software. This process can at times take as long as a year or not at all because of objections by one of the committees. In the free-beer-software case it will take a couple of clicks of the mouse and the download time. The researcher from the well funded institute can then contribute to FOSS again as a pair of eyes and a pair of hands and also by recommending to his Institute library to buy the relevant books written by the developers of the software.

It will be ideal that well funded institutes or governments themselves fund the development of free software. FOSS, as is obvious from the above discussion, creates a level playing field across Institutes, Universities and various colleges

affiliated to Universities. This paper is an attempt to put into perspective the contributions of FOSS and Open Standards to Scientific Computation and thereby to a nation's development. It is hoped that it will provide policy makers enough reasons to support FOSS and Open Standards.

### ***Defining open standards:***

A standard is a specification that is generally accepted and widely used. Standards ensure the quality and interoperability of products in spite of being from different vendors. For example the universal serial bus (USB) is a serial bus standard to interface devices as varied as keyboards, mouse devices, PDAs, joysticks, digital cameras, scanners and printers, to a computer [10]. It is therefore clear that the standard a product adheres to, plays an important part in the public buying/using the product. Cartels or big businesses that set standards can leverage the standards to favor their products leading to monopolies. For example read what Richard Stallman has to say about “winmodems” [11], wherein part of the job done by a modem is built into the software and what that software is supposed to do is not known, “closed source – closed standard”. Therefore there is a need for open standards which are characterized by the following properties as listed by Nah Soo Hoe [9]:

- Easy accessibility for all to read, use and implement.
- It is developed by a process that is relatively easy for anyone to participate in, and
- There exists no tie-in by any specific group or vendor.

One example of such an open standard is the Message Passing Interface (MPI) standard. Despite processor speeds doubling roughly every two years (Moore's law) [12], it was seen that scientific computing always needed much higher performance than any single processor was capable of delivering. Parts of the algorithm that are independent of each other were then identified, and these in principle could be run on different processors, leading to the paradigm of parallel computing. In parallel computing the computational job is shared amongst several processors. This is possible by means of a library to communicate amongst the different processors and carry out different tasks in parallel amongst the processors. There exists several libraries to do this task (ex: PVM, ANULIB, etc). It was however realized that the programs written to use one library was neither portable nor shareable. This deficiency was acute enough for the various vendors and developers of parallel libraries to meet and form an open standard for passing messages across processors. The Message Passing Consortium was thus formed and they came up with the MPI standard [13]. There now exist several FOSS implementations of this standard like mpich [14] and Open MPI [15]. This is a prime example of how the

needs of portability and sharing along with a need for a level playing field naturally leads to an open standard.

## ***What is Computational Science?***

*Computers are incredibly fast, accurate, and stupid; humans are incredibly slow, inaccurate and brilliant; together they are powerful beyond imagination – Albert Einstein.*

Till the advent of computers there were two ways of studying natural phenomena: Carry out experiments and make a theoretical analysis. Experiments were carried out to study a phenomenon of interest and its results noted. Models for the observations, based on known physics were made and the corresponding equations to be solved are identified. The equations are then solved using various assumptions and simplifications to make the equations analytically tractable. When the results from such a model are consistent with experimental observations, it affirms the validity of the model, the assumptions and the simplifications and leads to understanding of the phenomena. The advent of computers complemented this process of understanding nature in the following ways:

- It made hitherto impossible calculations possible due to the large number of floating point operations (FLOPS) that could be performed. Earlier, though the equations to be solved were known, due to large volumes of the calculation that have to be performed, the calculations were not attempted. This helps to relax constraints in models and to explore model behavior in new domains. Example: N particle dynamics using molecular dynamics [16]; Ab-Initio quantum chemistry – Paul Dirac [17] was quoted as saying, *“The fundamental laws necessary for the mathematical treatment of large parts of physics and the whole of chemistry are thus fully known, and the difficulty lies only in the fact that application of these laws leads to equations that are too complex to be solved”*. John A. Pople got a noble prize in chemistry, in 1998 [18] for his development of computational methods in quantum chemistry (Walter Kohn shared the noble with him for his development of the density functional theory [19]).
- It is an important aspect in the design of complex experiments. For example, machines called tokamak are the most promising devices to generate energy from fusion. Several countries are collaborating to build a tokamak called ITER [20]. Computer simulations play a major part in the design and performance evaluation of each component of ITER [21].
- It helps in testing the assumptions made in theoretical models. Generally it is seen that when interpreting results from computational science codes, a better understanding of the underlying phenomena results. This leads to a

better feel for the qualitative and quantitative contributions of various effects and therefore to a better idea about the validity of various assumptions that can be made by a theoretical model to explain the phenomena. Example: A multi-scale model to study the diffusion of hydrogen in porous graphite using the following computational tools has been developed [22]. It consists of an open-source, but not exactly free, molecular dynamics code to study diffusion of hydrogen in crystal graphite, a kinetic Monte-Carlo (KMC) code using exclusively FOSS tools and a combination of KMC and Monte-Carlo Diffusion (MCD). Using the insights gained from the detailed computer simulations, analytical models for trace hydrogen diffusion in porous graphite was proposed [23].

- It is a medium to do experiments. Techniques have been developed to take advantage of the large number crunching capabilities of computers and do simulation experiments on the computers. Monte Carlo methods (MC), particle in cell methods (PIC) and molecular dynamics (MD) simulations are examples of simulation experiments carried out on a computer. Example: See the nice PIC simulations of plasma sheaths and plasma behavior in electromagnetic fields available at [24].
- Finally, it helps in the analysis and interpretation of large amounts of experimental data using a variety of data analysis tools like fast fourier transforms (FFT), Hilbert Huang Transforms, statistical analysis of experimental data, etc; Quick codes that do the necessary calculation or codes that work on a previously created data base like function parametrization or neural nets are required for feed-back control of experiments.

Therefore, the computer, in addition to being a tool to carry out analytically insolvable calculations is not only a new tool to studying natural phenomena, but is also the only tool to study certain physical phenomena [2]. The three pathways to understanding nature is illustrated in figure.1.

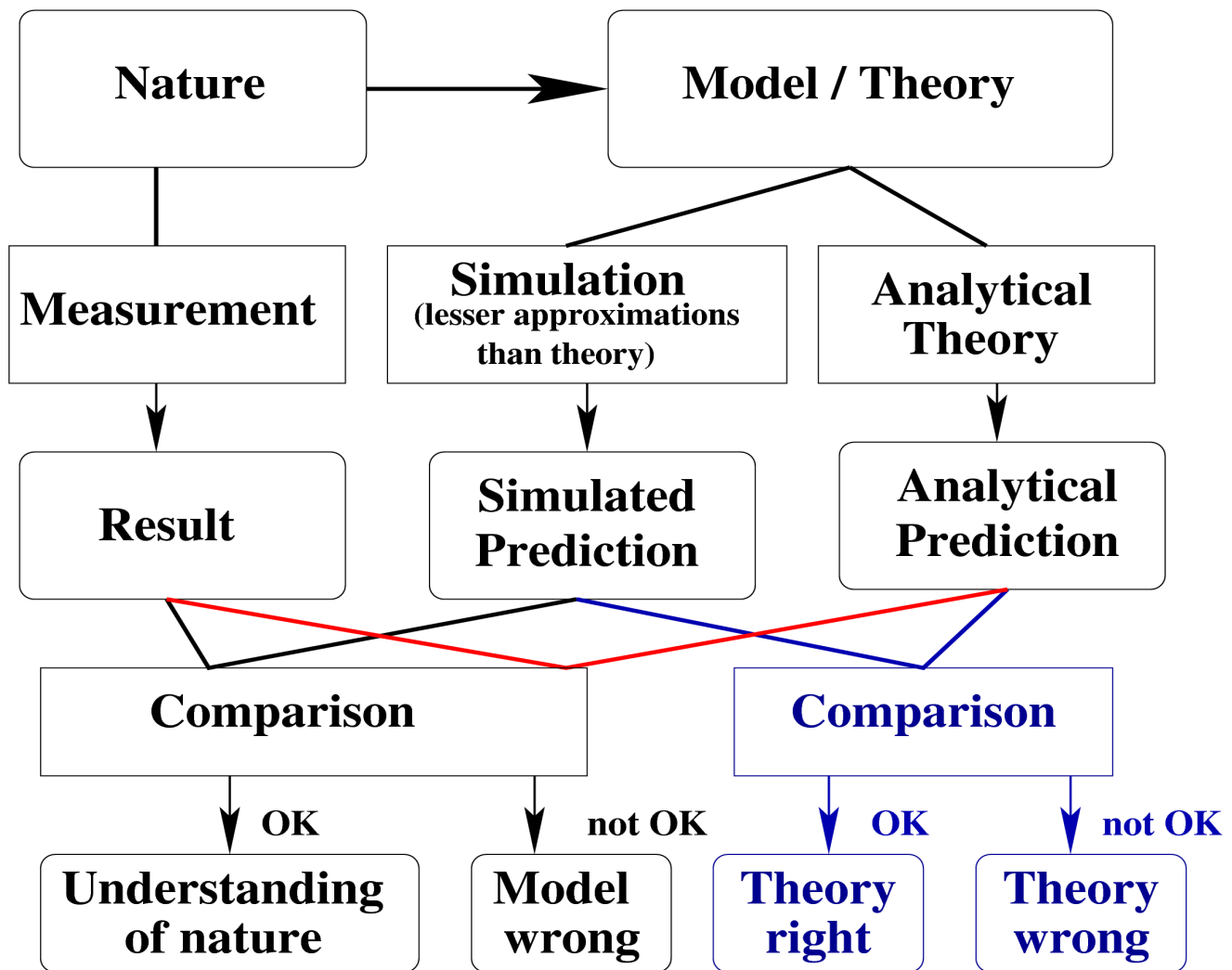


Figure.1: Relationship between the three paths to understand nature, viz. Experiments, Theory and Computer Simulations (Adapted from Prof. Kai Nordlund's notes from <http://beam.acclab.helsinki.fi/~knordlun>)

### **Layout of this paper:**

In the next section it will be shown that the free and open source Operating System (OS), Linux, is an essential part of state of the art in scientific computing. It will be shown that it is the dominant OS used in the top-500 supercomputers. A few open source computational science programs that use these supercomputers will be introduced.

In the third section, the current problems faced by computational science and the role FOSS and Open Standards can play in the development of computational

science as a reliable, complementary tool to furthering our knowledge of natural phenomena will be presented.

In the fourth section, the role of FOSS and Open Standards in overcoming the problems faced by computational science will be discussed in detail. Better software engineering practices by researchers [3] will be shown to help in optimizing the code, decreasing errors and decreasing production time in computational science codes. Open standards for evaluation of research publications which lay to rest at least some of the misgivings that exist in current methods of evaluation of scientific publication will be proposed. Open archives will be shown to be better than or at least an inexpensive alternative to existing methods of disbursement of scientific knowledge.

In the fifth section an introduction to free and open source scientific software repositories is given along with recommendation for the choice of linux distribution. A few subject independent software development tools are introduced along with links to documents for further reading. The pros and cons of using FOSS tools as opposed to proprietary closed source software for scientific computations is discussed in light of the discussions of earlier sections.

From the point of view of equal access and a level playing field across research institutes and universities, where the amount of funding has large gradients, it will be argued that these open source software and open archives can be used to reduce the “access to resources” gradient. Finally, a case for introducing scientific computing courses using FOSS at the graduate level and an appropriate syllabus based on open standards for a smooth transition of a student from being a science graduate to being a researcher capable of scientific computing in the Indian context.

## **FOSS, Computational Science and the top 500 supercomputing sites**

In the past couple of years there is a whole lot of speculation and activity going on about GNU-Linux conquering the desktop. However in one sphere of computing – scientific computing – it already has the lions share. The statistics page at the top-500 supercomputing sites page, shows that GNU-Linux is prevalent in more than 77 % of the top 500 supercomputers (figure .2). Note how Linux, the Free and Open Source Operating System, has become an increasingly dominant aspect of supercomputing at the cost of mainly Unix. Between 2002 to 2004 it grabbed the lions share and it's share continues to grow.

Nearly half of the top 500 supercomputers are used for scientific research (see Table-1 below). From the table it is seen that 43 % of the supercomputers in the list is from the Academics and Research segment. This segment also takes up 64.61 % of the floating point operations per second (FLOPS) of the total FLOPS capacity of the top 500 supercomputers and 69.69 % of all the processors in this list.

Some of the fastest computers in this list are used to run scientific computations in biology and material science [6][25]. These codes are open source and available under the GNU-GPL. Some of these codes are contributing to bleeding edge scientific discovery [26] which would never have been possible without supercomputers. Over and above this, most of these supercomputers are either clusters or massively parallel processor systems which contribute to typically around 96 % of the processors, FLOPS and number of supercomputers count of the top 500 supercomputers. Managing and monitoring these systems requires a variety of tools, many of which are FOSS [27]. Most of the codes use parallel programming libraries based on MPI/OpenMP which are open standards and have the possibility of being compiled with the FOSS compiler collection, gcc. Unfortunately the detailed statistics on these issues are not available and would be of interest from a “FOSS in top-500 supercomputers” point of view. In a way, the count of number of supercomputers from a region to the list is also an indicator of the scientific and technological development of the region.

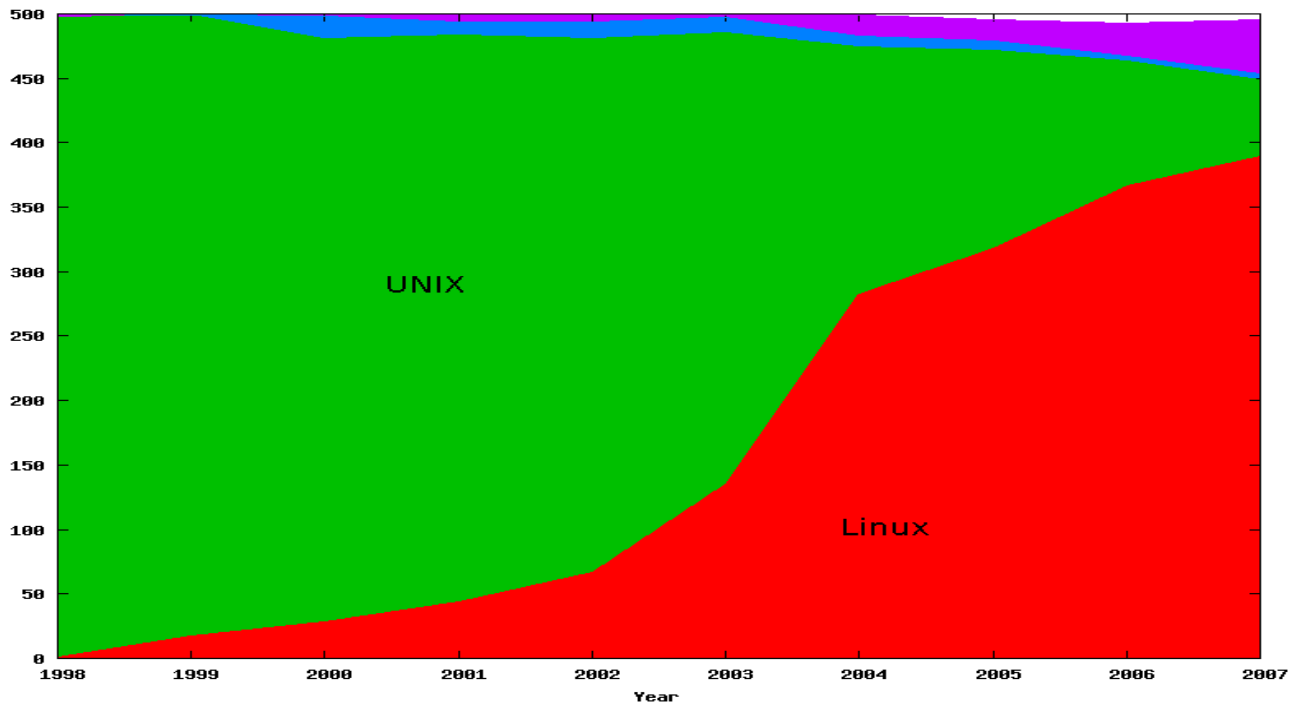


Figure.2: This figure shows the growing dominance of Linux as the operating system of choice in supercomputers. This figure is based on the data from [1]. The blue color is the share of BSD and the magenta that of other/mixed OS. The thin white line at the top is the single digit contributions from Macs plus Windows OS.

Sector	Count	%	Rmax (GFlops)	%	No. of processors	%
Academic & Research	215	43.0	3195692	64.61	851006	69.69
Classified	7	1.4	60485	1.22	15892	1.30
Government	2	0.4	9248	0.19	2000	0.16
Industry	268	53.6	1596662	32.28	329668	27.00
Vendor	8	1.6	84500	1.91	22548	1.85
All	500	100	4946590	100	1221114	100
IOSN-SA-Node Countries	8	1.6	45697	0.92	10336	0.85

Table 1: Segment-wise break-up of the top-500 Super Computers for the year 2007 [1]. The South Asian node countries contribution to this (only India contributes to this list) is shown at the bottom of the table.

# Current status of Computational Science

Based on the merits listed in the previous section, viz..

- making hitherto impossible calculations possible due to higher floating point operations per second.
- designing of complex experiments,
- testing assumption of theoretical models,
- possibility of carrying out simulation experiments, and
- analyzing experimental data,

it is clear that computational science is a worthy means of exploring natural phenomena. Douglas E. Post and Lawrence G. Votta however argue that [2], “Computational Science has reached a stage where better organization and new methods of verifying and validating complex data are now mandatory if it has to fulfill its promise to science and society”. In the next few paragraphs I shall try to present their arguments for this conclusion.

Their thesis is that any engineering technology, typically passes through four stages on their way to maturity:

1. Initial adaption of the technology. Here the design limits are not known leading to over-design and therefore no chance of failure of the technology.
2. Then there are cautious design improvements based on the experience gained from working on the initial adaption. Since these are cautious design improvements, they too stand the test of matching up to the purpose of their design.
3. Then a stage is reached where the design limits are stretched leading to failure of the technology.
4. Analysis of the failures lead to better understanding of the limits, leading to maturity in the field and optimized, efficient design.

D. E. Post and L. Votta assert that computational science is at this precarious stage of development wherein the application limits are stretched leading to failures and give some spectacular examples (page 36-37 of [2]). They also rightfully observe that, “the pioneering computational scientists had strong backgrounds in the disciplines that were subjects of their computations and they retained a healthy skepticism about their computational results”. This led to better verification and

validation and therefore to correct conclusions. This statement, by what it does not say, leaves an implication that not all among the current crop of computational researchers have a strong grounding in subject they simulate and also that they are probably not as skeptical as they ought to be about their results.

Based on their experience as referees and editors (D. E. Post is a Computational Physicist at Los Alamos National Laboratory and Associate Editor in Chief of *Computing in Science and Engineering*; L. Votta has been an Associate Editor of *IEEE Transactions on Software Engineering*), they state that the peer review process in computational science does not provide as effective a filter as it does for Experiments and Theory. They say that it is hard to decide if a code is right or wrong. The various parts of a code that can contribute to its flaws are:

1. There may be hidden faults in the code like it may be applying algorithms improperly or the spatial or temporal resolution applied may be inappropriately coarse.
2. The models and equations can be wrong.
3. The data used in the code can be inaccurate or have inadequate resolution.
4. Perhaps someone who uses the code written by somebody else does not know how to set up and run the problem properly or how to interpret the results.

In my view, most of the above reasons have their parallels in Experiments and lesser so in theory. The point to note is that this is in resonance with what Paul Ginsparg says about the refereeing process for scientific documents. In a paper titled, "Can Peer Review be better Focused?" [28], he says, "*Outsiders to the system are sometimes surprised to learn that peer-reviewed journals do not certify correctness of research results. Their somewhat weaker evaluation is that an article is (a) not obviously wrong or incomplete, (b) is potentially of interest to readers in the field*". Therefore if the aim of a reviewer is to only satisfy criteria (a) and (b) above, at least points 1, 3 and 4 above can sneak through a poor publication.

Post and Votta list out the following challenges faced by computational science:

- **the performance challenge**, which is being met by faster processors as seen in Figure.3.
- **the programming challenge** which is the development of languages and software tools to facilitate programming for massively parallel computers (MPC), which is partially met by the introduction of new standards [13] [15] and libraries [14] and tools for managing MPCs, and
- **the prediction challenge**, which is where they assert computational

science researchers are faltering .. at the precarious third stage of development.

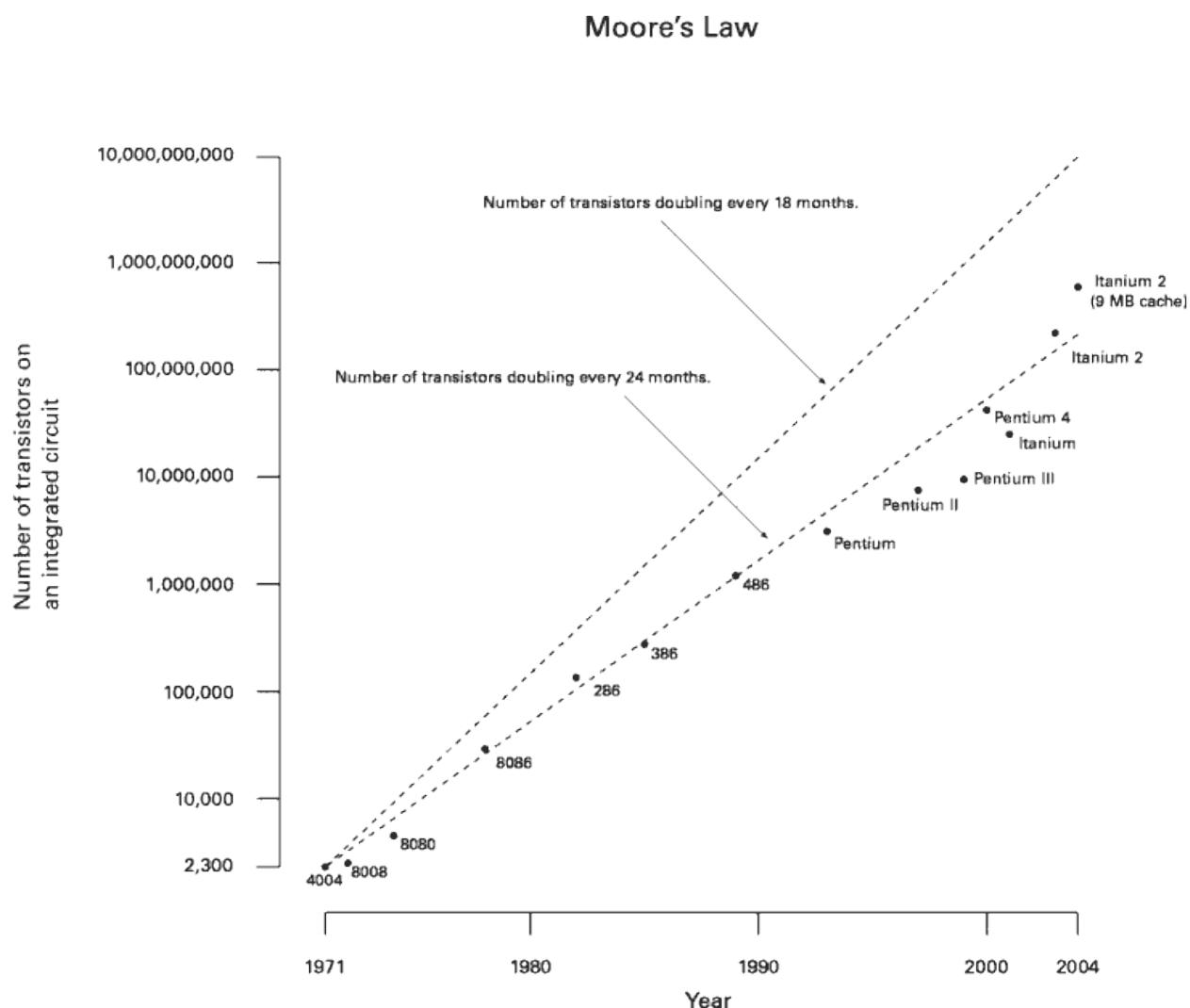


Figure.3: Moore's Law, which states that the number of transistors that can be placed on an integrated circuit doubles every two years. The number of transistors on Intel processors is also depicted for comparison. (figure, courtesy of W. G. simon at the English Wikipedia project).

They say that the field is in transition from simple modest codes to large codes spanning several temporal and spatial scales. They present a case study of six large scale computational projects and conclude that *verification, validation* and *quality control* are all equally crucial for the success of a large scale computational science project. On the dissemination of information point of view *there is a distinct perception that better means of evaluating scientific work in general* [28] and

*computational science in particular [2] is essential.*

Soon after the article by Post and Votta, Gregory Wilson wrote an article titled, "Where is the real bottleneck in Scientific Computing?". Citing Amdahl's law he argues that lack of awareness of some basic software engineering tools and methods is what delays computational science projects and not the run time of the computer codes. He therefore puts forward the argument that teaching basic software engineering practices will speed up the whole process of obtaining scientific computation results, because computational scientists spend a lot of time in code development, debugging, verification and validation. The code runtime is small compared to these activities. Amdahl's law is simply illustrated as follows [29]. Given two independent parts of a process A and B, A taking 80 % of the run time, and B, 20 %. Note that it is better to speed up A by two times rather than speeding up B by five times. See Figure.4 for a pictorial view of this.



Figure.4: Illustration of Amdahl's law from ([http://en.wikipedia.org/Amdahl's\\_law](http://en.wikipedia.org/Amdahl's_law))

## ***Role of FOSS and Open Standards in overcoming the current limitations of Computational Science***

From the above section, it is clear that (i) Code verification, (ii) Code validation, (iii) Quality control during the code development process and (iv) better means of evaluating and distributing scientific work are crucial. Here I will discuss each of these aspects and how FOSS and Open Standards can help in overcoming these limitations.

### ***Code Verification and Validation***

This can be done in various ways:

- Comparing code results to a known answer at various analytical limits. This helps in gaining confidence in at least some parts of the code.
- Checking if truncation error varies as expected with grid size; or checking if the results follow a square root of sample size variance in the case of Monte Carlo (MC) and Molecular Dynamics (MD) codes. This gives some confidence that the implementation of the numerical method (in case of varying grid size) and implementation of the MC and MD algorithm is fine.
- Monitoring conserved quantities, symmetry properties, etc. This gives confidence that there are no unknown sources or sinks in the system.
- Benchmarking with existing codes if any. This helps in increasing the confidence level of both the codes, especially if a different simulation method is used in the codes.

Making the code OPEN SOURCE with a free license so that many others can use it and report errors helps. Each of the four points above can then be explored by many more people who have access to the source code. It is the diversity of the people using the code that is the strength of the FOSS model here. Different analytical limits of which the original authors were not even aware of can be checked; Different sample/grid sizes can be checked depending on the problem at hand and the availability of computational resources – this also makes sure the portability of the code; Source code availability also improves the chances of comparison with other codes.

Code validation is usually done by benchmarking with experimental results or with known results from other codes employing either similar or alternative computer simulation techniques. For example, the Monte Carlo method can be used to validate the results of equilibrium properties of a Lennard-Jones fluid obtained from molecular dynamics. D. E. Post and L. Votta suggest [2] that experiments must be

specially designed to validate large code projects that are of benefit for science and humanity.

### ***Quality control during the code development process***

Adopting some basic software engineering practices also helps in the quality of written code. Gregory Wilson has developed a whole course on the basic 20 % of Software Engineering that he feels computational science researchers need to know to overcome their main bottleneck – rapid code development, debugging and other quality control aspects. The course is available under an open license [30]. Some of the tools taught in the course and otherwise necessary for quality control are:

- Debugging using FOSS tools like gdb [31], ddd [32], etc. These are state of the art tools that speed up your debugging and therefore code development and testing time enormously. Imagine being able to fire off your buggy code within a debugger and when it crashes all you have to do is to move your mouse over various variables of the source code which is open in another window. The values of the variables and therefore the reason for the crash becomes immediately obvious without having several print statements in your code.
- Using a version control system like either cvs [33] or svn [34]. The benefits of these in a multi-developer environment cannot be emphasized, however these have their uses for a single developer too. It is many times useful to go to an earlier version of your code to do a related problem. Using these tools not only helps in tracking the code version you want but also committing (rectifying) program errors found in one version, across versions.
- Using error handlers in your code and having several layers of debugging output which can be turned on or off. Error handling means not only checking if your inputs are right, but also checking if various parameters passed to different functions, procedures, etc are correct. Several layers of debugging output, depending on the amount of debugging desired is easily possible using the “`#ifdef DBuG`” to “`#endif`” loop.
- Creating test suites which test your program at various limits. The importance of this cannot be emphasized, especially for a code that is well validated for a certain set of parameters. The runtime executable may not have gone into regions of code that are far away from the parameters the code has been validated at into regions that contain bugs. Sharing your code (as FOSS) is a straightforward way of ensuring that its runtime space is fully explored because different people use it for different purposes. You will have a million hands testing your code.
- Structured programming and compilation. It is a good idea to break up your program into different repeatable tasks and create subroutines or functions

that carry out the tasks even if you do not use an object oriented programming language. Each of these can be then part of a separate file which can then be compiled together using a makefile [35]. Each function must not be more than 100-200 lines of code, beyond which the possibility of errors increases. Moreover since the program is structured, adaption or further development becomes much faster.

### ***Better means of Evaluation and Distribution of Scientific Work***

In the past decade we have witnessed the springing up of electronic on-line versions of traditional journals, so much so that the visits to the library for journal referencing has decreased drastically. These benefits have however not reached everyone equally. Badly funded universities in less developed countries cannot afford the costs of subscribing to these journals. Therefore we are in a situation where a researcher from an Institute publishes some work in a journal of repute and a fellow researcher from a not so well funded university from the same country is not able to access it just because he/she does not have access to the journal.

In a set of three papers ([28] and links thereof), Paul Ginsparg describes the advantages of moving to an open access system of scientific communication which is not only more cost effective, not only a better way of scientific communication between peers, but also creates a level playing field for researchers from poorly funded organizations by giving them access to the latest research at the cost of a computer connected to the internet<sup>1</sup>.

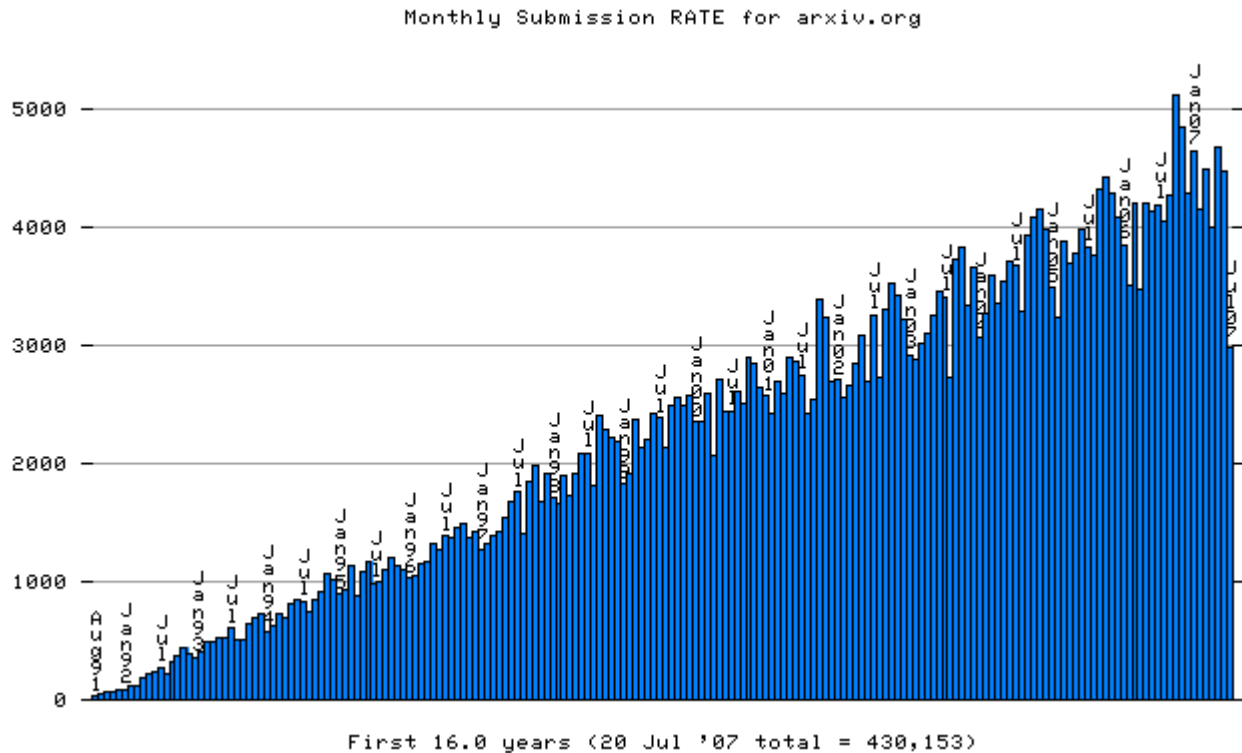
He argues that the current methodology of research and dissemination and validation is premised on a paper medium that is difficult to produce, distribute, archive and duplicate. With the development of new technologies and the benefits of using the electronic medium, each of the above drawbacks of the paper medium is overcome. Production has become easier by various new documenting formats and technologies, distribute (e-mail, ftp, http), archive (store on a web-page; searching in the electronic medium would also be much easier) and duplicate (a cp – copy for the uninitiated – command).

As mentioned earlier, he correctly says that the current refereeing process only says that a paper is not obviously wrong and is of possible interest to fellow researchers working in the same field. Based on the experiences from an open e-print archive [36] he suggests that researchers are willing to forgo the implicit weak recommendations of a refereed paper for the quick availability of information from an e-print archive. The monthly submissions to the open archive, arxiv, site is shown in Figure.5 to give an idea of its increasing popularity amongst researchers.

---

1 In India the typical cost of this would be less than a one time initial cost of around Rs. 20000 (roughly 500 USD) and a monthly cost of Rs. 250 (roughly 6 USD) for a broadband connection. The one time initial cost can even be brought down further if the aim is to only connect to the internet for browsing paper archives, a second hand PC with 256 MB RAM and a modest 15 inch monitor with an old P-III or celeron processor can be obtained at a much lesser price.





*Figure.5: The number of papers submitted to arxiv.org the open access archives of scientific papers ([http://arxiv.org/show\\_monthly\\_submissions](http://arxiv.org/show_monthly_submissions)).*

Finally coming to the role of FOSS in computational science publications, if the research being evaluated for publication is done using FOSS tools and based on an open source code, it will be easy for the referee (who probably uses the same code for his research) to evaluate the results. He would also be aware of the drawbacks and therefore can judge the conclusions presented in a better way.

## FOSS Tools for Computational Science

FOSS tools are used in almost all scientific topics and there exist extensive directories on the web (for example, [37] has a listing of searchable scientific software, both proprietary and FOSS) covering a wide range of topics in Physics, Mathematics, Chemistry, Biology, Graphics and Visualization. The freshmeat site [38] and the Sourceforge site [39] cover links to various subject specific softwares. For example, the Sourceforge site has links to around 11187 Scientific and Engineering projects. The Scientific/Engineering category of the freshmeat site has the following listing of topics and projects (as of September 2007):

- Archeology – 9 projects
- Artificial Intelligence – 314 projects
- Astronomy – 147 projects
- Bioinformatics – 214 projects
- Chemistry – 114 projects
- Electronic Design Automation – 121 projects
- Geographical – 145 projects
- Image Recognition – 79 projects
- Mathematics – 602 projects
- Medical Science Applications – 143 projects
- Neuroscience – 41 projects
- Physics – 48 projects
- Visualization – 637 projects

These directories are extensive, but do not contain all of what is available for scientific computing and it will be prudent to search the INTERNET with appropriate *key-words* and *GPL* for getting a list of software which is *free* in the topic of your interest. The sites mentioned above [37][38][39] contain links to various free and open source scientific software which are subject specific. They also contain links to a variety of code development tools. The Scientific Computing using GNU/Linux HOWTO [40] is another collection of links to tools for scientific computing and code development as opposed to being subject specific. It is a subset (slightly dated, and needing revision) of FOSS tools that is common across subjects and will be of general use for scientific computing.

There also exist subject specific Linux distributions like *Linux for Astronomy*, *Linux for Biotechnology* and *Linux for Chemistry* [41] over and above “Scientific Linux” distributions like Quantian [42] and Scientific Linux [43]. A general overview of subject specific FOSS is beyond the scope of this article due to the wide range of topics covered by science. I will focus on a few widely used tools (compiler, debugger, version control, profiler, etc) for scientific code development, irrespective of the subject, in this section. One question that is often asked is which Linux distribution is best for scientific computation. The answer is not straightforward but depending on whether one has the expertise to compile from source, one must take into consideration the following points when choosing the distribution:

- Do there exist pre-compiled binaries of the scientific software of interest in the distribution? RedHat [44], fedora [45], Debian [46] based systems usually have large repositories which have pre-compiled binaries for almost any software.
- If one has the expertise to compile the sources, and one has modest hardware specifications (P-IV, with only 512 MB RAM), one should probably go for a Slackware [47] or Debian based system with a simple window manager that does not hog resources. If one does not have the expertise to compile programs from sources, and have only modest resources, one is better off with a Debian based system with a simple window manager like icewm, fluxbox, xfce, etc [48] (lightweight is the technical term for such window managers).

In the next few subsections, I briefly introduce a small, but important subset of FOSS tools that can be used for scientific code development irrespective of the subject. Useful links to documents on using the tools will hopefully offset the brief introduction.

### ***GNU Compiler collection (GCC)***

GCC [49] is GNU's project to produce a world class compiler. It works on multiple architectures and diverse environments from mainframes to embedded systems. It is one of the reasons for the wide portability of the various free software being developed on GNU/Linux platforms. Currently GCC contains front ends for C, C++, Objective C, GNU Fortran-95, Java, and Ada, as well as libraries for these languages (libstdc++, libgcj,..). For manuals on using the various GCC compilers see the online documentation [50].

GCC has an extensive list of compile time options for controlling output, dialect, diagnostic messages formatting, suppress warnings, debugging, optimization controls, preprocessor controls, linking, target architectures, etc. For a detailed list of compile time options see the man page (`man gcc` at the command prompt) or see section.3 of the gcc online documentation [50]. Some of the compilation flags that I regularly use are the **-Wall**, **-g**, **-O2**; **-Wall** enables all the warnings about constructions that some users consider questionable and easy to avoid or modify to prevent the warning; **-g** enables production of debugging information in the operating system's native format which gdb can use; **-ggdb** produces information in a format which gdb or ddd can directly use. See below for a brief introduction to a couple of free and open source debuggers. **-O2** gives the code a level of optimization such that there are no space-speed trade-offs. It makes sense to switch off the **-g** flag for production runs.

Speed wise the GCC compilers are not the fastest compilers available, but they show a decent performance across different vendor CPUs; for example, the Intel

compilers may be optimized for Intel CPUs and show a much weaker performance on AMD CPUs [51][52]. The best advantage is of course the portability and sharing aspect which is indispensable for collaborations which is the lifeline of scientific activity. Recently gcc seems to have support for OpenMP which is an advantage keeping in mind the multi-core processors that seem to be the future of the desktop and of Linux clusters, at least for the next few years.

## ***GNU Scientific Library (GSL)***

The GSL [53] is a collection of numerical routines written from scratch in C. It was conceived in 1996 by M. Galassi and J. Theiler of Los Alamos National Laboratory who were joined by other researchers who also felt that the licenses of existing libraries were hindering scientific cooperation. It provides an Applications Programming Interface (API) for C programmers and also allows wrappers to be written for very high level languages. It covers a wide range of numerical computing topics, has a good manual, is widely portable and is distributed under the GNU General Public License. The library has functions and examples for a wide variety of numerical techniques like:

- Complex Numbers
- Roots of Polynomials
- Special Functions
- Vectors and Matrices
- Permutations
- Sorting
- BLAS Support
- Linear Algebra
- Eigen systems
- Fast Fourier Transforms
- Quadrature
- Statistics
- Histograms
- Random Numbers / Sequences / Distributions
- N-Tuples
- M-C Integration
- Simulated Annealing
- Differential Equations
- Interpolation
- Numerical Differentiation
- Chebyshev Approximation
- Discrete Hankel Transforms
- Root-Finding
- Minimization
- Least-Squares Fitting
- Physical Constants
- IEEE Floating-Point

- Discrete Wavelet Transforms

The sources and manual for GSL is available at [54]. The manual consists of around 500 pages and has several examples in each section on how to call the libraries. It is seen that anybody with a basic knowledge of C can easily start using the library. It has been developed on a GNU/Linux platform using gcc and is highly portable and compiles on the following platforms:

- SunOS 4.1.3 & Solaris 2.x (Sparc)
- Alpha GNU/Linux, gcc
- HP-UX 9/10/11, PA-RISC, gcc/cc
- IRIX 6.5, gcc
- m68k NeXTSTEP, gcc
- Compaq Alpha Tru64 Unix, gcc
- FreeBSD, OpenBSD & NetBSD, gcc
- Cygwin
- Apple Darwin 5.4
- Hitachi SR8000 Super Technical Server, cc

The purpose of the above list is to demonstrate the portability of software developed using gcc on a GNU/Linux platform. Since gcc has been ported to several platforms, it is natural that software written in gcc compiles and runs on all those platforms. This helps in collaborative research wherein it is not necessary that everybody involved in the collaboration has similar computer architectures.

## ***Debugging Code***

Any scientific code can contain three classes of bugs:

1. Those caused by non-standard constructs, simply bad syntax and badly structured careless programming,
2. Those due to variables blowing up wherein the program crashes with a segmentation fault.
3. Hidden bugs due to algorithmic errors or model errors.

The first class of bugs can be tracked down by using compile time flags like `-Wall`, programming discipline, and structural programming practices. The third class of bugs are the hardest to find and can be found to exist by verification and validation of the code as described earlier. Zeroing in on the region of error will depend on the kind of verification test failed. For the second class of errors, which is most common, two very good GNU based debuggers exist and are described below.

All programs written in the languages supported by GCC can be debugged using the GNU project debugger (gdb) [55] or the GNU data display debugger (DDD) [56], an excellent interactive, command line debugger. As mentioned above the programs must be compiled with either the `-g` option or the `-ggdb` option. This then compiles your code to produce debugging information during runtime such that either gdb can interpret it. It is invoked by typing `gdb` at the command prompt. This opens a shell with the command prompt now being `(gdb)`. At this prompt you can load an executable for debugging with the command `file executable.name` and run the executable using the command `run`. You can then inspect program crashes, variable names, back-trace function calls, etc within the gdb shell. These details are beyond the scope of this paper and the reader is referred to the gdb manual [57].

The GNU Data Display Debugger, GNU DDD, is a graphical front-end for command-line debuggers such as GDB, the Perl debugger, or the Python debugger. In Figure.6 you can see a typical c code being debugged. The source code is displayed on clicking the *interrupt* button. The green cursor indicates the point of interruption of the source code. By moving the cursor over the variables, the value of each variable will be displayed, both as a pop-up and also at the gray, bottom end of the DDD window. For instance, in the above example, by moving the mouse over the variables the following values of the variables are obtained: The value of `ChkParts` is 1, `NParts` being an array has the values `{24137, 7365, 1412, 268, 6172, 108, 15, 0, 0, 0, 0, 0}`, `Icount` is 1824, `DeltaT` is `1.9003450353129177e-17` and of `M` is 0. GDB commands can be entered in the box just below the menu on the top left hand side of the DDD GUI. In case of a segmentation fault, the `backtrace` command gives a detailed stack of functions with line numbers leading to the fault. Any other gdb command can also be entered at this location. All this helps in debugging a code without making it ugly / unreadable with hundreds of print statements and spending a long time in tracking the calls of various functions leading to the fault. Besides usual front-end features such as viewing source texts it also has a good interactive graphical data display, where data structures are displayed as graphs. A manual for DDD is available at [58].



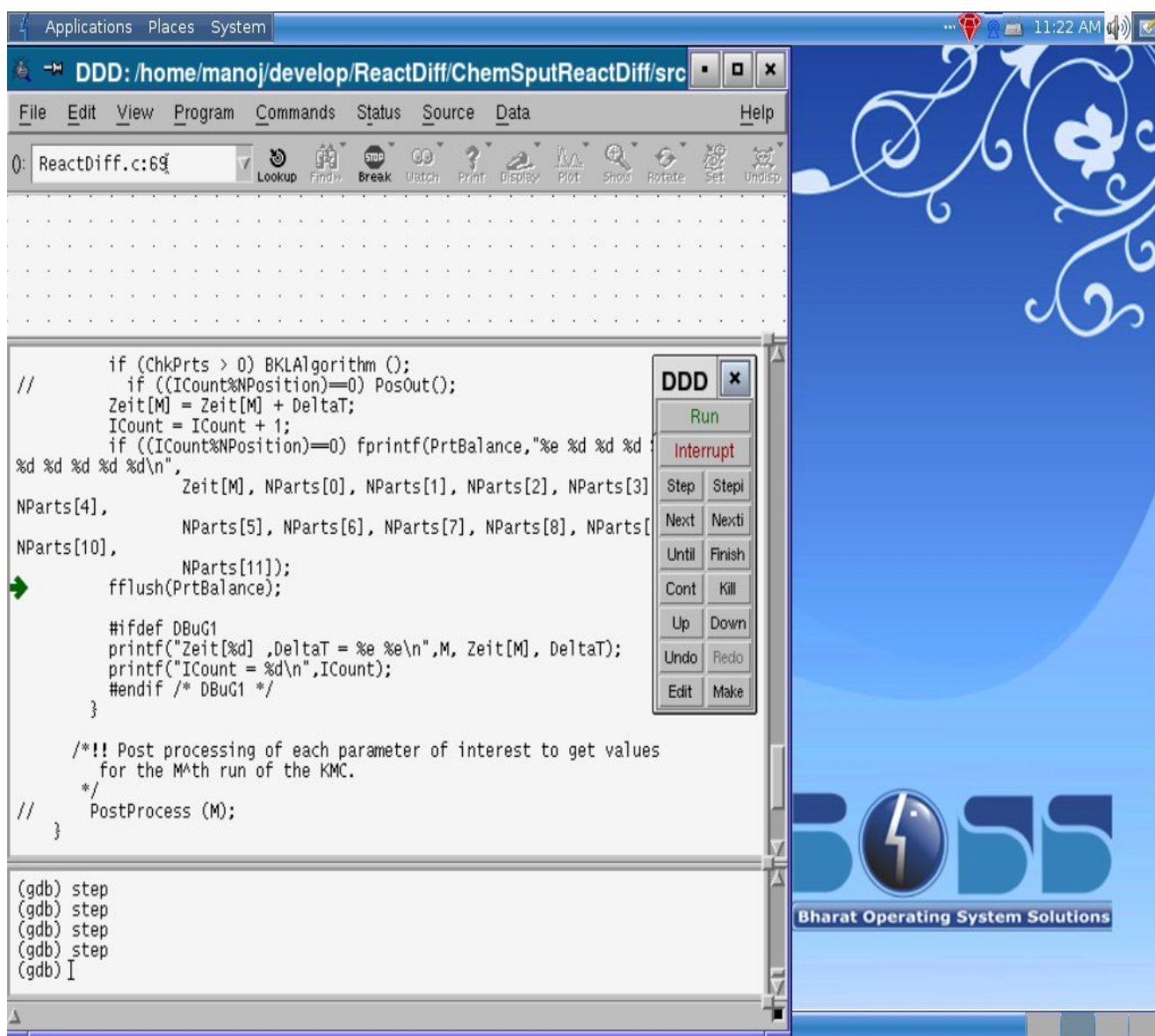


Figure 6: A typical DDD GUI debugging a C program. Various interactive debugging actions can be performed on the code during runtime. For details see the documentation [58].

## GNU Make

Most of the computational science codes (at least the well structured ones) have functions distributed across several files. The number of files can vary from tens to several hundreds in number. The larger codes may take a long time to compile, usually tens of minutes to maybe hours. If a small change is effected in one of the files it does not make sense to spend a long time recompiling the whole program. The GNU make [59] addresses this need and helps to recompile only that part of the executable that is affected by the changes made. It does this by means of a *makefile*, prepared by the code developers, which contains information on which

functions are associated with which files and their dependence.

An example *makefile* of an unusually small code is shown below:

```
01 FFLAGS = -g +es
02 INCLUDES = -I. -I/usr/include
03 LIBS =
04 FF= g77
05 #Rule to create .o files from .f files
06 .f.o:
07     $(FF) -c $(FFLAGS) $(INCLUDES) $*.f
08 smpsi : smpsi.o physput.o chmsput.o ressput.o thermev.o \
09     heatdif.o bshevi.o
10     $(FF) smpsi.o physput.o chmsput.o ressput.o thermev.o \
11     heatdif.o bshevi.o $(LIBS) -o smpsi
12 smpsi.o: smpsi.f physput.f chmsput.f ressput.f thermev.f \
13     heatdif.f bshevi.f
14 physput.o: physput.f
15 chmsput.o: chmsput.f
16 ressput.o: ressput.f
17 thermev.o: thermev.f
18 heatdif.o: heatdif.f
19 bshevi.o: bshevi.f
20 clean :
21     /bin/rm *.o smpsi
22 run:
23     ./smpsi
```

The `FFLAGS`, `INCLUDES`, `LIBS`, `FF` definitions (lines 01 to 04) are self explanatory if one glances at the lines (06,07) below the “rule to create .o files from .f files”. `smpsi` is the executable created and lines 08,09 and 12,13 show its dependencies, i.e the files containing the functions it calls. Therefore if any of these files are modified, the `smpsi` executable will be changed. Lines 10,11 show how the `smpsi` executable must be created. lines 14-19 contain the dependencies of the other files that make up the software (in this case the executable `smpsi`). Lines 20-21 and 22-23 show rules for cleaning up the compilation and also for execution of the program.

The compilation and runs can then be carried out using the command `make` at the command prompt. `make` searches the directory for a file named *makefile* or *Makefile* and proceeds to carry out the tasks specified in the file. If for some reason one wants to give the makefile a different name the name of the makefile can be specified by `make -f different.name` at the command prompt. To clean the compilation, do `make clean` and to execute the program do `make run` at the command prompt. A manual for GNU make is available at [60].

## **GNU profiler – gprof**

After debugging a code at times it is necessary to improve its performance. Note that many a time the amount of man-hours spent on optimizing a code could be longer than the hours gained due to the optimized code. Keeping this in mind, there exists the GNU profiler, `gprof` [61], which is a useful tool to find which function takes up the maximum CPU time and the number of times the function is called (called the *flat profile*). It can also give information on the number of times a function is called by other functions (called the *call graph*). The flat profile gives an idea of which functions can be tweaked for improving performance (remember Amdahl's law) and the call graph gives an idea of the interrelations between the various functions, which can suggest better ways of efficiently restructuring the various functions.

In order to get this information (*flat profile*, *call graph*), the code is compiled with `gcc` using the `-pg` flag. Upon running the so compiled program, a binary file named *gmon.out* is created in the directory of the run. This can then be processed with `gprof`, by running the executable *gprof* in the same directory, to generate human readable output. For further details see the `gprof` manual [62]. For a more detailed review of performance analysis (free and non-free) tools see the Lawrence Livermore National Laboratories tutorial on, “Performance analysis tools” [63].

## **Version control systems (cvs, svn)**

Version control is important not only in a multi-developer environment, but also for a single code developer. In scientific computation one sees that the same algorithm (or kernel) can be used to tackle a variety of problems with minor tweaks, additions or changes in the source code. This is an advantage FOSS has in the sense that several researchers with diverse needs would use the same software leading to a final product that can do a variety of calculations (see the quantum espresso [64] mailing list for a feel of the diversity). One would want backup copies of each of these changes and version control software is very useful in this case. A computational science researcher would probably need a version control tool much more that he/she would need a profiler.

Two of the best and widely used (even amongst proprietary closed source software developers) version control tools are the concurrent versions system (CVS) [33] and subversion (svn) [34]. These also have excellent online documentation [65][66] and the interested reader is encouraged to use these online books to learn version control. A brief introduction to version control is also available at the software carpentry course [67].

## ***Parallel programming FOSS tools***

During the discussions on open standards we have dwelt on the need for parallel programming, wherein independent blocks of a computing task are run concurrently on different processors. These blocks need to communicate with each other at least at the beginning and at the end of their run since they are part of the same code, referred to as an extremely trivial parallelization (ETP). Many Monte-Carlo codes can be parallelized this way to obtain much better statistics. In many cases the tasks are not fully independent and may need to communicate several times during their run. An open standard library, the message passing interface (MPI), was proposed for communication among processors [13]. There exist two free implementations of this standard as libraries – mpich [14] and lam-mpi [68] and a MPI-2 implementation open-mpi [15].

There exists several documents on how to set up a cluster of personal computers (PCs) connected to each other via a high speed switch (see [69] for links to documents, Linux distributions, clustering software, administering software, etc). However to learn the rudiments of parallel programming, one only needs a couple of PCs connected to each other. Download and install one of the free MPI libraries. It is generally believed that one needs to know a basic minimum of six MPI library calls to get started and this is sufficient for many ETP (embarrassingly trivial parallel or extremely trivial parallel) codes. For such a basic tutorial on getting started, see [70]. A more comprehensive tutorial is available at the LLNL site [71]. For a variety of freely available tutorials on the web see [72] and for a comprehensive course on MPI see [73]. This should be sufficient to get you started off on your supercomputing journey.

Of late, OpenMP [74], an application programming interface, consisting of a set of compiler directives, library routines, and environment variables has been developed for taking advantage of symmetric multi-processing systems. It provides a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to supercomputers. Of late, gcc has support for OpenMP [75]. A nice OpenMP tutorial, again from LLNL, is available at [76].

## ***Comparison of FOSS and Proprietary software for scientific computing***

Generally speaking there exist a variety of proprietary software for scientific computation. Rather than going into specific comparison of a few FOSS with proprietary software, the pros and cons of these two paradigms for scientific computation are presented. It must be kept in mind that the user in this case is not a lay-man who just wants to click the mouse and complete a task, but a technical person who has the expertise to modify sources, create new algorithms, etc.

- Not everybody can afford to use proprietary software. This hinders collaboration. Usually due to the limitations on number of users and number of machines the proprietary software is licensed to, there could be instances where a certain people within an Institute that has bought the software cannot use it at their convenience or cannot use it at all. This is not the case with FOSS.
- Procuring proprietary software takes time due to the various committees that have to approve the procurement and in many cases you do not have access to the source code. FOSS can be directly downloaded form the web. There may be many who consider that the time lost in procuring proprietary software is made up by not spending development time. However it must be realized that having access to the source code would teach one more about the software and one has the freedom to adapt the code to ones needs.
- Help on using proprietary software takes time, whereas mailing-lists of FOSS are a good source for guidance and help.
- Publications based on FOSS can be validated by any referee whereas those based on proprietary software cannot be fully validated by referees who do not have access to the software. However sufficient checks to assert that the work is not obviously wrong can be made.
- Proprietary scientific software usually has good quality control checks, but there always remain questions about the kind of algorithm implemented for a particular task. One might have wanted to use a newly established, different algorithm for the task. With FOSS there is the possibility to examine each sub-task and modify the algorithm as you want.

The above points being made, it must be mentioned that proprietary software in many cases provide the benchmark as far as features and graphical user interface is concerned that FOSS tries to achieve.

# A Case for Introducing GNU-Linux in the Syllabus at the Universities

A local, informal, non-professional questioning at my Institute, which contains researchers and research students from all across India, plus the experience of sitting at a few interview committees for new entrants to research, reveals that most of the researchers and research students came across GNU-Linux systems only after they start off doing their research. This was furthermore self taught or learnt from helpful seniors or peers. It is also seen that most of the researchers had an informal “Numerical Methods” course during their graduation<sup>2</sup> or at least during their post-graduation. A small fraction never have a numerical methods course. This shows a gap in standards for education across universities. Applying these numerical methods to solve problems in the natural sciences was not taught later on during the course work.

Numerical methods are only one part of computational science. Other aspects like simulation methods, namely molecular dynamics (MD), Monte Carlo methods (MC), particle in cell methods (PIC), Genetic Algorithm (GA), Cellular Automata (CA), etc, are not taught at either the graduate or post-graduate level, except in a few isolated cases or where the student does a project involving a simulation method during the post-graduation.

I would therefore strongly recommend teaching using GNU-Linux with numerical methods based on the GNU Scientific Library and a basic course on software carpentry [30] as part of the graduate syllabus plus including “Computational Methods in Science” as an option of specialization at the post graduate level in all universities offering a master's degree in science. The basis for making this recommendation is

- Computational science complements theory and experiments and is an alternative way of exploring nature. To occupy its position as the third pathway to understanding nature, basic training at the graduate and post-graduate levels is essential.
- FOSS tools are best suited for such a course, partly due to their easy availability and affordability. Over and above this, the availability of the source code helps the student understand how the theory (s)he learns in his/her numerical methods course is implemented. Open source also aids in tinkering and experimenting which aids deeper understanding. The quality of

---

<sup>2</sup> Typically in India after a schooling of 10 years, a student interested in pursuing a career in scientific research does a two year course in the subjects of his choice plus some ancillary subjects and compulsory language subjects. Then (s)he does his graduation in one group of science topics (Maths-Physics-Chemistry or Biology-Zoology-Chemistry) for three years. Finally (s)he does a Masters degree in the science topic of her/his interest.

FOSS software can also be better due to the more eyes to see – more hands to work – more varieties of minds to test the software as discussed in the Introduction to open source software section.

- Learning a course like the one in software carpentry [30] at the preparatory stage inculcates rudimentary software engineering practices that would help in the quality control which is essential for large code projects. I believe that it is much harder for researchers who already have set methods and beliefs of coding to inculcate these software engineering practices into their daily work.

## Summary and Conclusions

It has been shown that the GNU-Linux systems are the dominant systems of choice in advanced scientific computing. Scientific computing is at a crucial phase from where it can mature to have an equal footing with experiments and theory as a tool to understand nature. It needs better means of validation, verification and better quality control. Furthermore it is argued that the evaluation and dissemination of scientific research does not satisfy the needs of researchers on both, the quick availability front and the correctness of results front. Moreover the access to scientific publications is limited to researchers from a few well funded Institutes due to the prohibitively high cost charged by the non-open-access journals.

The advantages of Free and Open Source codes in better verification and validation, and in quality control has been presented. The availability of the sources allows for the possibility of the, *“many eyes to see, many hands to work and many differently-thinking-minds to test”* effect. The advantages of open standards for keeping in check monopolistic practices and its effect on software portability and thereby better collaborations have been discussed. It is argued that open archives can play a crucial role in quicker and better dissemination of scientific results.

In summary, FOSS, Open Archives and Open Standards can play a crucial role in not only improving the current status of scientific computation, but also improving its verification, validity and quality and in dissemination of scientific information in an equitable manner leading to better development opportunities for humanity.

## References:

- 1: Top 500 Supercomputing site, <http://www.top500.org/>
- 2: Computational Science Demands a New Paradigm Douglas E. Post, Lawrence G. Votta, Physics Today, Vol.58, No.1 (2005) 35-41.
- 3: Where's the Real Bottleneck in Scientific Computing? Gregory V. Wilson, American Scientist Online, Vol. 94, No.1 (2006) 5.  
<http://dx.doi.org/10.1511/2006.1.5>
- 4: Free as in Freedom, Sam Williams, ISBN: 0-596-00287-4,  
<http://www.oreilly.com/openbook/freedom/>
- 5: konqueror - web browser - file manager and more, <http://www.konqueror.org>
- 6: The Large-Scale Atomic/Molecular Massively Parallel Simulator,  
<http://lammmps.sandia.gov/>
- 7: GNU General Public License, <http://www.gnu.org/copyleft/gpl.html>
- 8: Kenneth Wong and Phet Sayo, FOSS: General Introduction, Elsevier (2004), ISBN-983-3094-00-7, <http://www.iosn.net/foss/foss-general-primer/>
- 9: Nah Soo Hoe, FOSS: Open Standards, Elsevier (2006) ISBN-13:978-81-312-038-9,  
<http://www.iosn.net/open-standards/foss-open-standards-primer/>
- 10: Universal Serial Bus, <http://en.wikipedia.org/wiki/USB>
- 11: The free software movement and the future of freedom, Richard Stallman  
<http://fsfeurope.org/documents/rms-fs-2006-03-09.en.html>
- 12: Moore's law, [http://en.wikipedia.org/wiki/Moore's\\_law](http://en.wikipedia.org/wiki/Moore's_law)
- 13: MPI-2: Extensions to the Message Passing Interface,  
<http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html>
- 14: The MPICH-2 Implementation, <http://www-unix.mcs.anl.gov/mpi/mpich2>
- 15: Open MPI: Open source high performance computing, <http://www.open-mpi.org>
- 16: Art of Molecular Dynamics Simulations D. C. Rapaport, ISBN-978-0521825689, Cambridge University Press (1995).

- 17: Paul Dirac, [http://en.wikipedia.org/wiki/Paul\\_Dirac](http://en.wikipedia.org/wiki/Paul_Dirac)
- 18: John Pople - Noble prize in chemistry – 1998, [http://nobelprize.org/nobel\\_prizes/chemistry/laureates/1998/pople-lecture.html](http://nobelprize.org/nobel_prizes/chemistry/laureates/1998/pople-lecture.html)
- 19: Walter Kohn, Noble Prize in Chemistry – 1998, [http://nobelprize.org/nobel\\_prizes/chemistry/laureates/1998/kohn-lecture.html](http://nobelprize.org/nobel_prizes/chemistry/laureates/1998/kohn-lecture.html)
- 20: International Thermonuclear Experimental Reactor, <http://www.iter.org>
- 21: The ITER Technical Basis, <http://www.iter.org/reports.htm>
- 22: Multi-scale modeling of hydrogen transport in porous graphite M. Warrier, Thesis, Ernst-Moritz-Arndt Universitat, Greifswald, Germany (2005) <http://www.ipp.mpg.de/~mow/PhDThesis>
- 23: Multi-scale modeling of hydrogen transport in porous graphite, M. Warrier, R. Schneider, E. Salonen and K. Nordlund, Jnl. Nucl. Mater, 337-339 (2005) 580-584
- 24: Plasma Theory and Simulation Group, <http://ptsg.eecs.berkeley.edu/>
- 25: <http://www.ks.uiuc.edu/Research/namd/>
- 26: US San Diego Supercomputer simulations may pinpoint the causes of Parkinson's Alzheimer's diseases, <http://ucsdnews.ucsd.edu/newsrel/supercomputer/03-07SDSC.asp>
- 27: Linux Clustering Software – Joe Greenseid, <http://freshmeat.net/articles/view/458>
- 28: Links - Can Peer Review be better Focused?, Creating a global Research Network, Winners and losers in the Global Research Village, Paul Ginsparg, Cornell University, <http://people.ccmr.cornell.edu/~ginsparg/blurb/>
- 29: Amdahl's law, [http://en.wikipedia.org/wiki/Amdahl's\\_law](http://en.wikipedia.org/wiki/Amdahl's_law)
- 30: Software Carpentry - Gregory Wilson, <http://swc.scipy.org>
- 31: The GNU Project Debugger, <http://sourceware.org/gdb/>
- 32: The Data Display Debugger, <http://www.gnu.org/software/ddd/>
- 33: Concurrent Versions System, <http://www.nongnu.org/cvs/>

- 34: Subversion, <http://subversion.tigris.org>
- 35: The GNU Make, <http://www.gnu.org/software/make>
- 36: Open e-print archive, <http://arXiv.org>
- 37: Scientific Applications on Linux, <http://sal.jyu.fi/index.shtml>
- 38: Scientific/Engineering directory at freshmeat, <http://freshmeat.net/browse/97>
- 39: Scientific/Engineering specific software from Sourceforge.net  
[http://sourceforge.net/softwaremap/trove\\_list.php?form\\_cat=97](http://sourceforge.net/softwaremap/trove_list.php?form_cat=97)
- 40: Scientific Computing with Free Software on GNU/Linux HOWTO,  
[http://tldp.org/HOWTO/html\\_single/Scientific-Computing-with-GNU-Linux](http://tldp.org/HOWTO/html_single/Scientific-Computing-with-GNU-Linux)
- 41: Subject specific Linux distributions, <http://www.randomfactory.com/>
- 42: The Quantian Scientific Computing Environment,  
<http://dirk.eddelbuettel.com/quantian.html>
- 43: Scientific Linux, <https://www.scientificlinux.org>
- 44: RedHat, <http://www.redhat.com>
- 45: Fedora, <http://fedoraproject.org>
- 46: Debian, <http://www.debian.org>
- 47: Slackware, <http://www.slackware.com>
- 48: FreeDesktop, <http://www.freedesktop.org/wiki/Desktops>
- 49: The GNU Compiler Collection (GCC), <http://gcc.gnu.org>
- 50: The GCC online documentation, <http://www.gnu.org/software/gcc/onlinedocs/>
- 51: Comparing Linux Compilers,  
[http://www.coyotegulch.com/reviews/linux\\_compilers/index.html](http://www.coyotegulch.com/reviews/linux_compilers/index.html)
- 52: Linux FORTRAN compilers, <http://www.polyhedron.com/compare0html>
- 53: The GNU Scientific Library (GSL), <http://www.gnu.org/software/gsl/>
- 54: The GSL manual, <http://www.gnu.org/software/gsl/manual/>
- 55: The GNU project Debugger (GDB), <http://www.gnu.org/software/gdb/>

- 56: The Data Display Debugger (DDD), <http://www.gnu.org/software/ddd/>
- 57: The GDB manual, <http://www.gnu.org/software/gdb/documentation/>
- 58: The DDD manual, <http://www.gnu.org/manual/ddd/>
- 59: The GNU make, <http://www.gnu.org/software/make/>
- 60: The GNU make manual, <http://www.gnu.org/software/make/manual/>
- 61: The GNU Profiler, <http://ftp.gnu.org/gnu/binutils>
- 62: The GNU profiler manual, <http://sourceware.org/binutils/docs-2.18/gprof/>
- 63: Performance analysis tools LLNL,  
[http://www.llnl.gov/computing/tutorials/performance\\_tools](http://www.llnl.gov/computing/tutorials/performance_tools)
- 64: Quantum espresso, <http://www.quantum-espresso.org/>
- 65: CVS red book, <http://cvsbook.red-bean.com/cvsbook.html>
- 66: Subversion red book, <http://svnbook.red-bean.com/>
- 67: Software carpentry, version control, <http://swc.scipy.org/lec/version.html>
- 68: lam-mpi, <http://www.lam-mpi.org/>
- 70: MPI: Getting started, <http://www.lam-mpi.org/tutorials/one-step/ezstart.php>
- 71: The LLNL tutorial on MPI, <http://www.llnl.gov/computing/tutorials/mpi/>
- 72: MPI Tutorials, <http://www.lam-mpi.org/tutorials>
- 73: Introduction to MPI, <http://webct.ncsa.uiuc.edu:8900/public/MPI/>
- 74: OpenMP: Simple, scalable SMP Programming, <http://www.openmp.org/drupal>
- 75: OpenMP in gcc,  
<http://cs.anu.edu.au/student/comp4300/lectures/Elliston-OMP.pdf>
- 76: The LLNL tutorial on OpenMP, <http://www.llnl.gov/computing/tutorials/openMP/>