

# Wo klemmt es wirklich bei wissenschaftlichen Berechnungen ?

Forscher wären gut beraten, sich einiger Arbeitsmethoden zu bedienen, die in der Software-Industrie üblich sind.

Von Gregory V. Wilson

»Mein Freund wusste nicht, was Versionskontrolle war«

**A**ls ich mich 1986 erstmals mit Computerwissenschaft befasste, wurde gerade eine neue Generation von schnellen und billigen Chips eingeführt, welche die Ära der Low-Cost-Supercomputer einläutete. Diese Rechner bestanden aus einer Vielzahl von Prozessoren, die ein vorgegebenes Problem parallel durcharbeiteten. Und plötzlich begann natürlich jeder, der sich mit komplexen Computerproblemen befasste, seine Programme so umzuschreiben, dass sie die Vorteile dieser neuen Rechner auch nutzen konnten. Das war mühsam. Denn die Compiler, welche die Programme auf Parallelcomputern lauffähig machen sollten, spielten oft verrückt. Es gab keine »Debugging«-Werkzeuge zur Fehlersuche; und der Versuch, eine Lösung anzugehen, glich oft dem, tausend Kreuzworträtsel gleichzeitig lösen zu wollen. Doch die Ergebnisse schienen viel versprechend. Die meisten Forscher waren überzeugt, dass mit Parallelrechnern einmal Computersimulationen möglich würden. Damit ließe sich ein ganzes Spektrum von Prozessen berechnen, die sich mit herkömmlichen Labormitteln oder gar mit Papier und Bleistift nicht lösen ließen; entweder weil sie zu umfassend, klein, schnell, langsam, gefährlich oder einfach zu kompliziert waren.

Dennoch hatte ich Mitte der 1990er Jahre das Gefühl, dass irgendetwas nicht stimmte. Auf jede Gruppe mit einer erfolgreichen Simulation des globalen Klimas kamen ein Dut-

zend andere, die nur damit kämpften, ihre Programme zum Laufen zu bringen. Niemals war die Software schon genügend ausgereift, um auf Konferenzen präsentiert oder auch nur auf der Titelseite des Newsletters ihres Supercomputer-Centers erwähnt zu werden. Monate und Jahre plagten sich viele Forscher damit, ihre Codes umzuschreiben oder zu verbessern, bis die Software etwas Besseres machte, als nur stehen zu bleiben oder bei einer Division durch Null den Dienst zu quittieren. Aus irgendeinem Grund ließ der Erfolg stets viel länger auf sich warten als gedacht.

Ich befragte daher diese Wissenschaftler, wie sie denn ihre Programme schreiben würden. Die Antworten waren ernüchternd: Nur wenige hatten etwas mehr Ahnung als die meisten Programmierer kommerzieller Software, mit denen ich zusammengearbeitet hatte. Die Mehrheit benutzte dagegen noch antiquierte Texteditoren wie »Vi« oder »Notepad«, tauschte Codes mit Kollegen per E-Mail aus und absolvierte praktisch keine systematischen Tests der Programme.

## Suche nach dem »Rückgängig«-Knopf

Schließlich fragte ich einen Freund, der in Teilchenphysik promovierte, warum er darauf bestand, alles »auf die harte Tour« zu machen. Warum benutzte er nicht eine integrierte Software-Entwicklungsumgebung mit einem symbolischen Debugger? Warum schrieb er keine Tests für Module? Warum nutzte er kein System zur Kontrolle der Programmversionen? Seine Gegenfrage: »Was ist ein System zur

Kontrolle von Programmversionen?« Eine solches System, erklärte ich ihm, sei eine Software, die Änderungen an Dokumenten, Programmen, Webseiten, Förderungsanträgen und so ziemlich allem anderen überwacht.

Es funktioniert wie der »Rückgängig«-Knopf in deinem Lieblingsprogrammeditor. Du kannst jederzeit eine ältere Version reaktivieren und sehen, welche Unterschiede es im Vergleich zur aktuellen gibt. Du kannst außerdem feststellen, wer zwischenzeitlich das Dokument editiert hat, und Konflikte aufspüren zwischen einzelnen Veränderungen durch andere und denen, die du gerade selbst anbringst. Die Programmversionskontrolle ist für das Programmieren ähnlich wichtig wie penible Notizen über Laborabläufe in der experimentellen Wissenschaft. Sie ist das Werkzeug, bei dessen Nutzung man sagen kann: »Auf diese Weise bin ich zu diesen Ergebnissen gekommen« statt »Hm, also ich glaube, für diesen Graph benutzen wir den neuen Algorithmus, äh, ich meine den alten neuen Algorithmus, nicht den neuen neuen Algorithmus.«

Mein Freund ist intelligent und mit den Problemen beim Schreiben großer Programme bestens vertraut – er hatte ein Programm mit 100 000 Zeilen übernommen und selbst weitere 20 000 Zeilen hinzugefügt. Als ich bemerkte, dass er gar nicht wusste, was Versionskontrolle ist, kam er mir vor wie ein Chemiker, dem nicht klar ist, dass man die Reagenzgläser zwischen den Versuchen spülen sollte. Auch er war über unser Gespräch nicht sehr erfreut. Er seufzte und sagte: »Hättest du mir das nicht schon vor drei Jahren erzählen können?«

Als ich wusste, wonach ich zu suchen hatte, fiel mir dieser »programmtechnische Analphabetismus« fast überall auf. Die meisten Wissenschaftler hatten einfach niemals gelernt, wie man effektiv programmiert. Nach einem Einführungskurs für Studienanfänger in C oder Java und vielleicht einem Kurs in Statistik oder Numerik im weiteren Studium wurde erwartet, dass sie alles andere allein hinkriegten. Das ist etwa so, als würde man jemandem zwar zeigen, wie man Polynome differenziert, aber dann von ihm verlangen, die Tensorrechnung zu beherrschen.

Ja, im Internet stehen dazu alle nötigen Informationen, aber verteilt über Hunderte



von Webseiten. Noch hinderlicher ist, dass die Forscher sich monate- oder jahrelang Hintergrundwissen aneignen müssten, bis die im Internet verstreuten Infos für sie verständlich würden. Als ich einem anderen Physiker (etwas älter und zynischer als mein Freund) sagte, er solle sich doch ein paar Wochen freinehmen und endlich mal »Perl« lernen, antwortete er mir: »Gern, wenn Sie einige Wochen Zeit übrig haben, um sich mit Quantenchromodynamik zu befassen, damit Sie auch meine Arbeit machen können.«

### »Vorsicht, ich bin Computerwissenschaftler und möchte Ihnen helfen!«

Im Gespräch mit ihm stieß ich noch auf einen weiteren Grund, warum viele Forscher sich keine besseren Arbeitsmethoden aneigneten. Von ständig neuen Programmierkonzepten und -ideen überrollt, reagieren diese Forscher vernünftigerweise erst einmal skeptisch, wenn ihnen jemand sagt: »Ich bin Computerwissenschaftler und komme, um Ihnen zu helfen.« Von objektorientierten Sprachen bis hin zum heute hoch gelobten Agilen Programmieren mussten sich die Wissenschaftler von einer Programmiermarotte zur nächsten durchkämpfen, ohne dass ihr Berufsleben dadurch merklich erleichtert worden wäre.

Forscher sind oft auch frustriert von der »zufälligen Komplexität«, mit denen sie die Informatik konfrontiert. So gehört zum Beispiel zu jeder modernen Programmiersprache ein Verzeichnis häufiger Ausdrücke – Mustervorlagen, mit denen in Textdokumenten bestimmte Daten gefunden werden können. Doch die Regeln der einzelnen Sprachen, wie diese Muster anzuwenden sind, unterscheiden sich leicht. Wenn allein schon das weit ver- ▷

»Selbst der schnellste Rechner kann die Zeit bis zur Publikation höchstens halbieren«

▷ breitete Betriebssystem Unix drei oder vier unterschiedliche Notationen für ein und dasselbe Konzept kennt, ist es kein Wunder, dass viele Forscher verzweifeln und sich nur auf den kleinsten gemeinsamen Nenner einlassen.

Welche Folgen hat das mangelnde Programmierwissen aber für die Wissenschaftler? Um Ihnen eine Vorstellung zu geben, bitte ich Sie, kurz eine der grundlegenden Regeln der Computerwissenschaft zu betrachten, das so genannte Amdahl-Gesetz. Nehmen wir an, es dauert sechs Monate, um ein Programm zu schreiben und von Fehlern zu reinigen; dieses läuft dann weitere sechs Monate auf aktueller Hardware, bis Resultate zur Veröffentlichung vorliegen.

Selbst ein beliebig schneller Computer (vielleicht einer, den Physiker einer fernen Zukunft bei einem missglückten Experiment zurück in unsere Gegenwart befördern) könnte die durchschnittliche Dauer bis zur Veröffentlichung höchstens auf die Hälfte verkürzen, da die Hardware nur eines von mehreren zeitintensiven Gliedern in der ganzen Prozesskette darstellt. Daher besteht der wichtigste Zeitfaktor für Wissenschaftler, die mit Computern arbeiten, zunehmend darin, wie schnell und zuverlässig sie ihre Ideen in funktionierenden Code umsetzen können.

**Weniger Zeit für den Kampf mit dem Code**

Im Jahr 1998 begannen Brent Gorda (heute am Lawrence Livermore National Laboratory) und ich damit, dieses Problem in Angriff zu nehmen. Wir gaben im Los Alamos National Laboratory (LANL) kleinere Kurse für Wissenschaftler, mit denen sie ihre Fähigkeiten in der Software-Entwicklung verbessern konnten. Wir hatten gar nicht das Ziel, aus LANL-Physikern oder Metallurgen Computerwissenschaftler zu machen. Vielmehr wollten wir ihnen lediglich die zehn Prozent an modernen Software-Entwicklungstechniken beibringen, mit denen sie 90 Prozent ihrer Aufgaben bewältigen konnten.

Die ersten Kurse hatten Höhen und Tiefen. Doch danach zu urteilen, was die Teilnehmer uns sagten und was sie taten, nachdem sie den Kurs absolviert hatten, war klar, dass wir uns auf dem richtigen Weg befanden. Bereits einige wenige grundlegende Techniken und eine Einführung in die Werkzeuge, die man dafür benötigte, ersparte vielen Forschern beträchtliche Frustrationen. Wir bemerkten weiterhin: Die meisten Wissenschaftler standen diesen Ideen sehr offen gegenüber – was uns, gemessen am praktischen Nutzen, eigentlich nicht so sehr überrascht haben sollte. Denn schließlich wurde ihnen die Wichtigkeit methodischen Vorgehens be-

reits in den ersten Studiensemestern eingebläut.

Sechs Jahre und einen Dotcom-Boom später erhielt ich Fördermittel von der Python Software Foundation, um diesen Kurs auf den neuesten Stand zu bringen und ihn zur freien Verfügung ins Internet zu stellen, sodass jeder, der ihn nutzen wollte, dies auch gratis tun konnte. Der Kurs liefert Tools und umfasst Arbeitsmethoden, die sowohl die Qualität wissenschaftlicher Programme erhöhen als auch deren Fertigstellung beschleunigen. Damit vergeuden Forscher weniger Zeit für den »Kampf mit dem Code« und gewinnen mehr Zeit für ihre wissenschaftliche Arbeit.

Zu den Kursthemen zählen weiterhin Versionskontrolle, die Automatisierung sich wiederholender Aufgaben, systematisches Testen, Programmierstile und Code-Lesen, einige Grundlagen für die Verarbeitung großer Datenmengen, Webprogrammierung – und ein Überblick, wie man Software-Entwicklung in kleinen, an (geografisch) verschiedenen Orten tätigen Teams koordiniert. Das ist alles kein bahnbrechend neues Wissen, sondern lediglich die computerwissenschaftliche Entsprechung von Routine-Laborarbeiten wie der Titration einer Lösung oder der Kalibrierung eines Oszillografen.

Wissenschaft ist viel mehr als nur eine Ansammlung von Fakten. Sie ermöglicht Menschen, die durch Ozeane voneinander getrennt sind, in unterschiedlichen Dekaden leben, verschiedene Sprachen sprechen oder anderen Ideologien unterliegen, wechselseitig auf den Entdeckungen der jeweils anderen aufzubauen. Computer spielen dabei von Jahr zu Jahr eine größere Rolle. Dennoch erfüllen nur wenige Forschungsprogramme die methodischen Standards, die Pioniere wie Lavoisier oder Faraday vor über 200 Jahren bei ihren experimentellen Studien gesetzt hatten.

Natürlich kann diese Lücke vor allem durch eine bessere Ausbildung während des Studiums geschlossen werden, doch das allein wird nicht reichen. Auch Fachzeitschriften müssen allmählich darauf bestehen, dass die Software der Wissenschaftler den gleichen Ansprüchen an Qualität und Reproduzierbarkeit genügt wie ihre Laborarbeiten. Außerdem brauchen wir dringend mehr Fachjournale, die gewillt sind, Beschreibungen zu veröffentlichen, nach welchen Methoden Forscher ihre Software entwickeln und wie diese Programme funktionieren. Schnellere Chips und anspruchsvollere Algorithmen sind nicht genug. Wenn wir uns Computerwissenschaft wirklich zu eigen machen wollen, müssen wir den Engpass zwischen unseren Ohren beseitigen. ◁

AUTOR UND LITERATURHINWEISE



**Gregory V. Wilson** ist Professor für Computerwissenschaft an der University of Toronto. Sein Kurs ist erhältlich unter [www.swc.scipy.org/](http://www.swc.scipy.org/). Seine E-Mail-Adresse: [gwwilson@cs.utoronto.ca](mailto:gwwilson@cs.utoronto.ca).

© American Scientist  
[www.americanscientist.org](http://www.americanscientist.org)

How to break web software. Von M. Andrews and J. A. Whittaker, Addison-Wesley, 2006

Debugging. Von D. J. Agans. American Management Association, 2002

Weblinks zu diesem Thema finden Sie unter [www.spektrum.de/artikel/852747](http://www.spektrum.de/artikel/852747).