

# Paralleles MATLAB: Viele Prozessoren und mehrfache Kerne

VON CLEVE MOLER

Vor zwölf Jahren, im Frühling 1995, schrieb ich einen Cleve's Corner mit dem Titel „Warum es kein paralleles MATLAB gibt.“ Dieser einseitige Artikel ist heute eine meiner am häufigsten zitierten Veröffentlichungen. Damals argumentierte ich, dass das zu jener Zeit von den meisten Parallelrechnern genutzte Modell eines verteilten Speichers nicht mit dem in MATLAB eingesetzten Speichermodell kompatibel sei, dass MATLAB nur einen kleinen Teil seiner Ausführungszeit für Aufgaben verwende, die sich automatisch parallelisieren ließen, und dass es nicht genügend potenzielle Kunden gebe, um den erheblichen Entwicklungsaufwand zu rechtfertigen.

Heute ist die Situation anders. Erstens hat sich MATLAB von einem simplen „Matrix Laboratory“ zu einer ausgereiften Umgebung für technische Berechnungen entwickelt und unterstützt auch umfangreiche Projekte, die über die reine numerische Lineare Algebra weit hinaus reichen. Zweitens besitzen die meisten aktuellen Mikroprozessoren zwei oder vier Rechenkerne (zukünftig wohl noch mehr) und heutige Computer verfügen über ausgeklügelte hierarchische Speicherstrukturen. Drittens hat die Mehrzahl der MATLAB-Anwender heute Zugang zu Rechner-Clustern und -Netzwerken und wird bald auch parallele PCs nutzen.

Aus all diesen Gründen haben wir heute nun doch ein paralleles MATLAB.

MATLAB unterstützt drei Typen der Parallelverarbeitung: Multithreading, Distributed Computing und die explizite Parallelverarbeitung (siehe Kasten). Die-

se verschiedenen Typen lassen sich auch kombinieren – so kann eine verteilte Berechnungsaufgabe auf Einzelrechnern mit Multithreading arbeiten und dann das Endergebnis aus einem verteilten Datenfeld (Distributed Array) abrufen. Die Zahl der beim Multithreading verwendeten Threads kann in MATLAB im Preferences-Feld festgelegt werden. Wir verwenden die Intel Math Kernel Library, die Multithread-Versionen der BLAS-Routinen enthält (Basic Linear Algebra Subroutines). Die Bibliothek der in MATLAB vorhandenen Grundfunktionen, zu denen beispielsweise trigonometrische und Exponentialfunktionen gehören, ist für Vektorargumente ebenfalls Multithreading-fähig.

Die jeweils richtige Methode für eine Parallelverarbeitung zu finden kann kompliziert sein. In diesem Cleve's Corner werden einige Experimente vorge-

## Die drei Typen der Parallelverarbeitung

### Multithreading

Beim Multithreading erzeugt eine MATLAB-Instanz automatisch mehrere Befehlsströme gleichzeitig. Mehrere Prozessoren oder Kerne, die sich den Speicher eines Einzelrechners teilen, führen diese Ströme aus. Ein Beispiel dafür ist die Addition der Elemente einer Matrix.

### Distributed computing

Distributed Computing bedeutet, dass mehrere MATLAB-Instanzen mehrere voneinander unabhängige Berechnungen auf verschiedenen Rechnern ausführen, die jeder über eigenen Speicher verfügen. Vor Jahren habe ich für diese verbreitete und wichtige Form der Parallelverarbeitung den Begriff „Embarrassingly Parallel“ („peinlich parallel“) verwendet, weil dazu keinerlei neue computerwissenschaftliche Erkenntnisse nötig sind. In den meisten Fällen wird einfach das gleiche Programm viele Male mit verschiedenen Parametern oder unterschiedlichen Sätzen („Seeds“) von Zufallszahlen durchlaufen.

### Explicit parallelism

Bei der expliziten Parallelverarbeitung laufen mehrere MATLAB-Instanzen auf mehreren Prozessoren oder Rechnern, die oft einen separaten Speicher haben, und führen gleichzeitig einen einzigen MATLAB-Befehl oder eine M-Funktion aus. Diese Form der Parallelverarbeitung wird durch neue Programmier-Konstrukte wie parallele Schleifen und verteilte Datenfelder (Distributed Arrays) beschrieben.

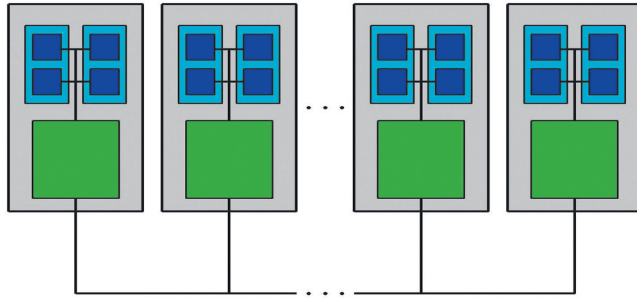


Abb. 1: Schema eines typischen parallelen Rechnerclusters.

stellt, bei denen Multithreading und Distributed Computing kombiniert werden.

## Parallele Rechnercluster

Abbildung 1 zeigt ein Schema eines typischen parallelen Rechnerclusters. Die grauen Kästen sind Einzelrechner mit eigenem Gehäuse, Netzteil, Netzwerkanschluss, Speicher und Festplatte. Die hellblauen Kästen darin sind CPUs, und die dunkelblauen Kästen darin sind wiederum Rechenkern. Die grünen Kästen stellen den Hauptspeicher dar. Es gibt eine ganze Reihe verschiedener Speichermodelle. Bei einigen Bauweisen greifen alle Kerne gleichberechtigt auf den gesamten Speicher zu. Bei anderen sind die Zugriffszeiten auf den Speicher nicht gleichförmig verteilt und der grüne Kasten mit dem Speicher könnte etwa in vier Teile unterteilt sein, einer für jeden Rechenkern.

The MathWorks verfügt über mehrere solcher Cluster unter Linux und Windows. Einer davon besteht aus 16 Rechnern mit je zwei AMD Opteron 285 Dual-Core-Prozessoren. Jeder dieser Rechner hat 4 Giga-byte Speicher, den sich die vier Kerne teilen. Der Cluster kann also 64 separate Rechenströme verarbeiten, hat aber nur 16 primäre Speicherbereiche. Diese Cluster bilden das HPC Lab (HCP steht für „High Performance Computing“ oder „High Productivity Computing“). Sie sind aber sicherlich noch keine Supercomputer.

Im Vergleich mit einem echten Supercomputer hat unser HPC Lab einen wich-

tigen Vorteil: Eine Person kann den gesamten Cluster für eine interaktive Sitzung für sich allein nutzen. Das ist luxuriös, denn normalerweise ist es üblich, dass die Nutzer eines Parallelrechners ihre Jobs an eine Warteschlange übergeben und dann auf deren Verarbeitung je nach freien Kapazitäten warten. Die Möglichkeit, auf Großrechnern interaktiv arbeiten zu können, ist selten.

Die erste 2004 erschienene Version der Distributed Computing Toolbox konnte mehrere unabhängige MATLAB-Jobs in einer solchen Umgebung verwalten. Die zweite Version aus dem Folgejahr wurde um eine MATLAB-Version des Message Passing Interface (MPI), dem Industriestandard für die Kommunikation zwischen Jobs, erweitert. Da MATLAB für „Matrix Laboratory“ steht, beschlossen wir, jede MATLAB-Instanz ein „Lab“ zu nennen und führten den Begriff `numlabs` ein, die Anzahl der an einem Job beteiligten Labs.

Mit Version 3.0 der Distributed Computing Toolbox erweiterte The MathWorks die Software um neue Programmierkonstrukte, mit denen MATLAB die in diesem ersten Benchmark genutzte „peinlich“ (embarrassingly) parallele Verarbeitungsweise mit Mehrfach-Jobs weit hinter sich lässt.

## Einsatz Multithreading-fähiger mathematischer Bibliotheken in MATLAB Tasks

Den Ausgangspunkt für dieses Experiment bildet `bench.m`, der Quellcode für die Benchmark-Demo. Die gesamte Grafikfunktionalität und der gesamte Code zur Berichtser-

zeugung wurden herausgenommen, so dass nur vier Berechnungstasks übrig bleiben:

**DGL** – Lösung der Van der Pol-Gleichung (einer gewöhnlichen Differentialgleichung) über ein längeres Zeitintervall mit Hilfe von ODE45

**FFT** – Berechnung der Fouriertransformation eines Vektors mit 220 Elementen mit FFTW

**LR** – Faktorisierung einer reellen dichtbesetzten 1000-mal-1000-Matrix mit LAPACK

**Sparse** – Lösung eines dünnbesetzten linearen Gleichungssystems der Ordnung 66.603 mit 331.823 von Null verschiedenen Elementen mit `\b`.

Mit der Distributed Computing Toolbox und MATLAB Distributed Computing Engine habe ich eine Reihe von Kopien dieser verkürzten Benchmark-Demo durchlaufen lassen. Die Multithreading-fähige Berechnung ist peinlich parallel – nach dem Start gibt es keinerlei Kommunikation zwischen den Tasks, bis diese alle beendet sind und die Ausführungszeiten vorliegen.

Die Diagramme in den Abbildungen 2 bis 5 geben jeweils das Verhältnis der Ausführungszeit eines Singlethread-Tasks mit einem Lab im Vergleich zur Ausführungszeit eines Multithread-Tasks mit mehreren Labs an. Die horizontale Linie markiert die ideale Effizienz –  $p$  Tasks mit  $p$  Labs werden so schnell berechnet wie ein Task mit nur einem Lab. Die erste und wichtigste Beobachtung dabei ist, dass die blaue Kurve bis hinauf zu 16 Labs bei allen vier Tasktypen mit der horizontalen Linie zusammen fällt. Daraus ist ersichtlich, dass Singlethread-Tasks am effizientesten sind, solange jeder Rechner genau einen Task berechnet. Das ist nicht überraschend, denn wenn dies für solche peinlich parallelen Tasks nicht zuträfe, wäre das ein Zeichen eines schweren Fehlers in unserer Hardware, Software oder den Messungen. Bei mehr als 16 Labs oder mehr als einem Thread je Lab sind die Verhältnisse allerdings komplizierter. Um das zu verstehen, müssen wir uns jede Berechnung für sich ansehen.

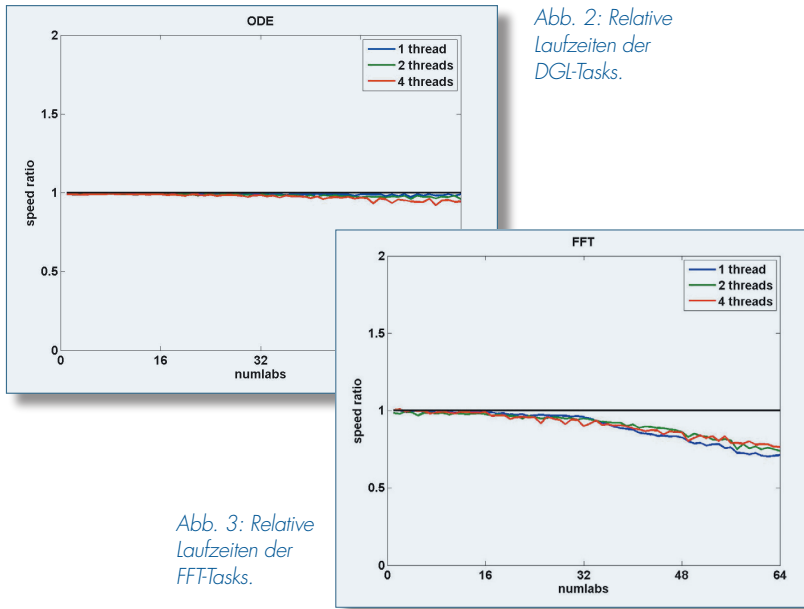


Abb. 2: Relative Laufzeiten der DGL-Tasks.

Abb. 3: Relative Laufzeiten der FFT-Tasks.

Der DGL-Task (Abb. 2) ist typisch für viele MATLAB-Tasks: Er arbeitet mit interpretiertem M-Code, wiederholten Funktionsaufrufen, einem moderaten Datenaufkommen, mäßig vielen arithmetischen Operationen je Schritt des DGL-Lösers und vielen Einzelschritten. Dieser Task zeigt sogar bis hin zu 64 Labs die ideale Effizienz. Jeder Einzelkern kann alle Berechnungen für ein Lab erledigen. Die Speicheranforderungen sind hier nicht dominierend. Mehr als einen Thread einzusetzen macht keinen Sinn, weil dem Task keine Multithreading-fähigen Bibliotheken zur Verfügung stehen. Genau genommen ist auch nicht ersichtlich, welchen Sinn Multithreading in diesem Fall überhaupt hätte.

Beim FFT-Task (Abb. 3) beträgt die Vektorlänge  $n = 2^{20}$ . Seine Komplexität,  $n \cdot \log(n)$ , lässt erkennen, dass für jedes Vektorelement nur eine Hand voll arithmetischer Operationen durchgeführt wird. Für 16 Labs ist die Effizienz des Tasks nahezu ideal, verschlechtert sich aber bei mehr als 16 Labs rapide, weil der Speicher die Kerne nicht schnell genug mit Daten bedienen kann. Die Zeit für die Berechnung von 64 Tasks durch 64 Labs ist um etwa 40 % länger als die, die ein Lab für einen Task benötigt. Mehrere Threads machen auch hier keinen

Unterschied, denn die verwendete FFT-Bibliothek unterstützt kein Multithreading.

Die blaue Linie im LR-Diagramm (Abb. 4) lässt erkennen, dass in diesem Fall ein Thread je Lab bis hinauf zu 64 Labs recht effizient, wenn auch nicht ganz ideal ist. Die zur Berechnung von 64 Tasks in 64 Labs benötigte Zeit ist nur um etwa 6 % höher als die, die ein Lab für einen Task braucht. Die Ordnung der Matrix ist  $n = 1000$ . Der Speicherbedarf,  $n^2$ , ist zwar in

etwa so hoch wie beim FFT-Task, aber die Komplexität der Ordnung  $n^3$  legt nahe, dass jedes Element viele Male wiederverwendet wird. Der zu Grunde liegende Faktorisierungsalgorithmus aus LAPACK nutzt den Cache sehr effizient, weshalb sich die Tatsache, dass es nur 16 primäre Speicherbereiche gibt, nicht negativ auf die Rechenzeit auswirkt.

Die grünen und roten Linien im LR-Diagramm zeigen, dass der Einsatz von zwei oder vier Threads je Lab von Vorteil ist, solange das Produkt aus der Anzahl der Threads und der Anzahl der Labs nicht größer ist als die Anzahl der Kerne. Mit dieser Einschränkung laufen zwei Threads je Lab etwa um 20% schneller als einer und vier Threads je Lab sind sogar um 60% schneller. Bei größeren Matrizen fällt dieser Prozentsatz sogar noch höher aus, bei kleineren dagegen geringer. Bei mehr als 64 Threads – also mehr als 32 Labs, die je 2 Threads nutzen, oder mehr als 16 Labs mit je 4 Threads – wird Multithreading kontraproduktiv.

Die Ergebnisse für den Sparse Task (Abb. 5) sind am wenigsten typisch und wohl auch die überraschendsten. Die Blaue Linie zeigt wieder, dass man bereits mit nur

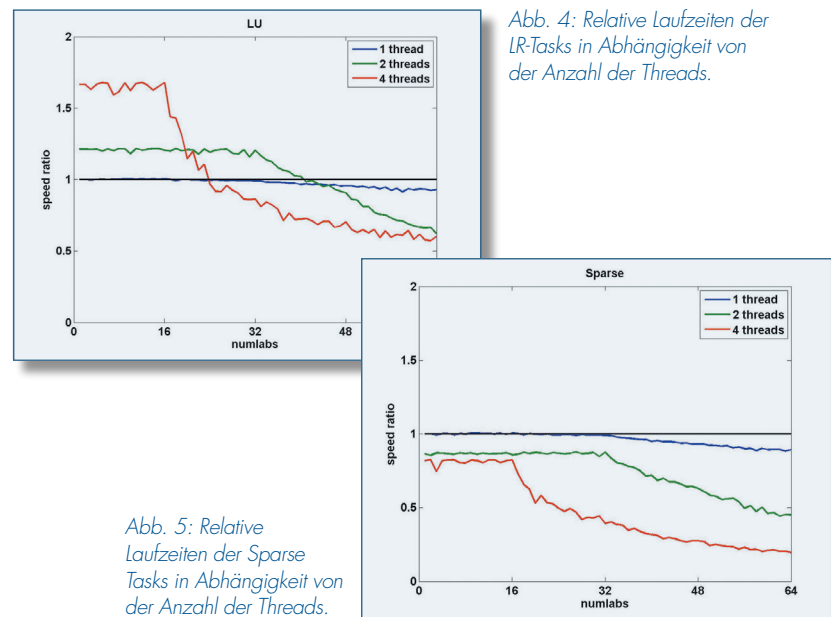


Abb. 4: Relative Laufzeiten der LR-Tasks in Abhängigkeit von der Anzahl der Threads.

Abb. 5: Relative Laufzeiten der Sparse Tasks in Abhängigkeit von der Anzahl der Threads.

einem Thread je Lab recht effizient arbeitet. Die roten und grünen Linien machen aber deutlich, dass Multithreading hier immer kontraproduktiv ist – zumindest für die gewählte Matrix. CHOLMOD, ein von Tim Davis entwickelter supernodaler Cholesky-Löser für dünnbesetzte Matrizen, wurde erst kürzlich in MATLAB eingeführt, allerdings bevor wir uns Gedanken um Multithreading gemacht haben. Der Algorithmus schaltet von dünnbesetzten Datenstrukturen, bei denen die BLAS (Basic Linear Algebra Subroutines) nicht genutzt werden, auf dichtbesetzte

Datenstrukturen und Multithreading-BLAS um, sobald die Zahl der Fließkommaoperationen für die von Null verschiedenen Matrizelemente eine bestimmte Schwelle übersteigt. Der supernodale Algorithmus ist vor allem für hochgradig rechteckige Matrizen geeignet und es scheint, als könne BLAS mit solchen Matrizen nicht besonders gut umgehen. Wir müssen uns also sowohl CHOLMOD als auch die BLAS in Zukunft noch einmal genauer ansehen, um ihre Zusammenarbeit zu verbessern. ◀◀

---

## Weitere Informationen

- CHOLMOD Linear Equation Solver  
[www.cise.ufl.edu/research/sparse/cholmod](http://www.cise.ufl.edu/research/sparse/cholmod)
- Cleve's Corner Collection  
[www.mathworks.de/res/cleve](http://www.mathworks.de/res/cleve)

## Quellen

### WEBSITE

[www.mathworks.de](http://www.mathworks.de)

### TECHNISCHER SUPPORT

[www.mathworks.de/support](http://www.mathworks.de/support)

### ONLINE USER COMMUNITY

[www.mathworks.de/matlabcentral](http://www.mathworks.de/matlabcentral)

### DEMOS

[www.mathworks.de/products/demos](http://www.mathworks.de/products/demos)

### SCHULUNGEN

[www.mathworks.de/training](http://www.mathworks.de/training)

### PRODUKTE VON DRITTANBIETERN

[www.mathworks.de/connections](http://www.mathworks.de/connections)

### WELTWEITE KONTAKTINFORMATION

[www.mathworks.de/contact](http://www.mathworks.de/contact)

### E-MAIL

[info@mathworks.de](mailto:info@mathworks.de)



©2007 The MathWorks, Inc. MATLAB, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, SimBiology, SimHydraulics sowie xPC TargetBox sind eingetragene Warenzeichen und SimEvents ist eine Handelsmarke von The MathWorks, Inc. Andere Produkt- oder Markennamen sind Handelsbezeichnungen oder eingetragene Warenzeichen der jeweiligen Eigentümer.