R | R | Z | N

Leibniz Universität Hannover

Gottfried Wilhelm Leibniz Universität Hannover
Regionales Rechenzentrum für Niedersachsen
Lehrgebiet Rechnernetze

Bachelor-Arbeit
im Studiengang Informatik (B. Sc.)

Benchmarking und Validierung der MPI-Funktionalität eines Rechenclusters

Verfasser:	Holger Kettler
Erstprüfer:	Prof. Dr.-Ing. C. Grimm
Zweitprüferin:	Prof. Dr.-Ing. G. von Voigt
Betreuer:	M. Sc. R. Gröper
Datum:	03. März 2009

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwandt habe.

Holger Kettler

Einleitung

Im Rahmen dieser Arbeit soll die Leistungsfähigkeit eines Rechenclusters mit Tests, sogenannten Benchmarks, ermittelt werden. Für die Durchführung und Einstufung des Ergebnisses dienen die Vorgaben der Organisation *Top 500.*

Neben einer Gesamtleistung aus einem Top 500 konformen Test und dem daraus abgeleiteten Ranking, sollen die Skalierbarkeit des Clusters sowie die Leistungsunterschiede durch alternative Netzwerktechnik dargestellt werden. Ein Test der MPI-Funktionalität erfolgt implizit unter hoher Last über die Durchführung des Benchmarks.

Das Kapitel der Grundlagen in dieser Arbeit soll die Motiviation für einen Benchmark zeigen und einen Eindruck von den komplexen Rahmenbedingungen vermitteln, die eine einfache Einstufung der Ergebnisse beeinflussen können. Theoretische Hintergründe für das weitere Verständnis des zweiten Teils sollen hier ebenfalls diskutiert werden. Durch das Ziel, eine Gleitkommaleistung für die Top 500 Liste zu ermitteln, ist die Verwendung des LINPACK-Benchmark bereits festgelegt. Die Herkunft des Programms und die Funktionsweise werden in der Arbeit erläutert. Im praktischen zweiten Teil werden verschiedene Konfigurationen mit dem Benchmark ausgeführt, Ergebnisse betrachtet und Erklärungen anhand der theoretischen Grundlagen aus dem ersten Teil gesucht. Am Ende steht die Gesamtleistung und das erwartete Ranking in der Top 500.

Inhaltsverzeichnis

I Grundlagen

1 Theorie hinter einem Benchmark

1.1 Gründe für einen Benchmark

*Ein **Benchmark** (engl. Maßstab) oder Benchmarking bezeichnet eine vergleichende Analyse mit einem festgelegten Referenzwert.*

Wikipedia

Eine Erklärung [5] für die Herkunft dieses Wortes könnte die Praxis eines Tischlers sein, Markierungen (*marks*) in seine Werkbank (*bench*) zu schnitzen, um die Länge von Stuhlbeinen zu kontrollieren. Ungleiche Beinlängen lassen sicherlich die Qualität und den Absatz des Endproduktes sinken. Nun handelt es sich bei Computersystemen nicht um massive Gegenstände, als vielmehr um Werkzeuge, deren Leistung am Ende auf dem mehr oder minder guten Zusammenspiel seiner diversen Bestandteile fußt. Im Prinzip müsste man die Leistung ohne äußere Beeinflussung, wie etwa dem Betriebssystem, messen. Das ist jedoch wenig realistisch und auch nicht praxistauglich. Man würde eher versuchen, durch simple Vergleichsmessungen Erkenntnisse zu gewinnen. Die Gesamtleistung dieser als Einheit funktionierenden Einzelteile ist ein Maßstab, der gewisse Auswirkungen unterschiedlicher Konfigurationen von Hard- oder Software verhältnismäßig gut widerspiegeln kann. Ebenso ist der Vergleich ganzer Einheiten, sprich: Computersystemen, auf Basis dieser Gesamtleistung möglich. So können Benchmarks helfen, Konfigurationen zu vergleichen, Fehlkonfigurationen zu finden oder einfach Kaufentscheidungen zu erleichtern. So hat ein Computer-Benchmark in diesem Punkt zumindest etwas mit dem in der Wirtschaft kursierenden Begriff des Benchmarking gemein, als dass auch hier eine vergleichende Optimierung angestrebt werden kann. Allerdings existiert wohl noch der sportliche Aspekt, für jede Leistung mindestens einen Test zu haben, der sie messen kann.

1.2 Maßangaben

Sicherlich lässt sich die Leistung eines Computersystems nicht so einfach darstellen, wie etwa die einer Luftpumpe. Ersteres beschränkt sich traditionell meist auf den Durchsatz an Instruktionen des Prozessors und die Speicherzugriffsleistung. Leistungen der Grafikkarte beispielsweise, werden nicht berücksichtigt[1]. Eine Zählung von

[1] Außer in speziellen Multimedia-Benchmarks, wie z.B. *3D-Mark*. Mittlerweile existieren aber brauchbare Toolkits, die die speziellen GPUs für Berechnungen nutzen.

"Millionen Instruktionen pro Sekunde" (MIPS) hat sich wegen der unterschiedlichen Architekturen, zum Beispiel komplexe CISC und einfach RISC Befehle, als wenig brauchbar herausgestellt. Außerdem resultieren Maßnahmen optimierender Compiler in schnellerer Ausführungszeit, aber geringerer Anzahl von Rechenoperationen und damit widersprüchlich in geringere MIPS. Andere Komponenten wie Massenspeicher oder Verbindungsnetzwerke können bei der Leistungsmessung von CPU und Speicher ebenso entscheidenden Einfluss haben, wie Software, die einfach schlecht mit den Ressourcen umgeht, auch wenn sie, oberflächlich betrachtet, nicht im Rampenlicht stehen. Mit zunehmender Geschwindigkeit bei der Verarbeitung steigt im selben Maße auch die Anforderung an die Zuliefergeschwindigkeit der Daten. Können die Daten nicht schnell genug geliefert werden, reduziert sich die Geschwindigkeit der schnellen Verarbeitungseinheit effektiv auf die des langsameren Mediums.

Im technisch-wissenschaftlichen Bereich ist es gewöhnlich die von einem Rechner bewältigte *Gleitkommaleistung* (engl. *floating point performance*), welche von Interesse ist. Gleitkommazahlen[2] sind die approximative Darstellung reeller Zahlen in einem Computer. Und die mit ihnen ausgeführten Operationen pro Sekunde (*flops - floating point operations per second*) sind dann ein mögliches Maß für eine Leistung. Es ist nicht unbedingt eindeutig definiert, welche Gleitkommaoperationen bei der Leistungsangabe in *flops* berücksichtigt werden. Zumeist sind dies jedoch elementare, arithmetische Funktionen (Addition, Multiplikation etc.), wie sie im Befehlssatz der *Floating-Point-Unit* (FPU) moderner CPUs implementiert sind. Möglich wären auch Zuweisungen, Vergleiche oder Typ-Konversionen. Die Behandlung von Gleitkommazahlen und Gleitkommaberechnung belastete in den Anfängen nur den Hauptprozessor, bis eine eigenen Hardware - zunächst als eigener Chip, später als Untereinheit der CPU selbst - dafür optimiert wurde. FPUs besitzen oft breitere Register und verfügen über hardwareseitig optimierte Methoden oder Lookup-Tabellen, um ihre Aufgabe effizient zu erledigen.

Jede dieser Funktionen selbst kann jedoch nicht einfach als einzelne Instruktion gezählt werden, da immer zusätzlicher Overhead (zum Beispiel das Laden aller Argumente aus dem Speicher) durch die begrenzte Anzahl von Registern ensteht, auf denen die CPU letztendlich arbeiten kann. Jedoch besitzen superskalare[3] Rechner mehrere unabhängige Funktionseinheiten. Sie arbeiten damit auf Befehlsebene parallel, da Anweisungen durch Ausnutzung der Nebenläufigkeit in Teilanweisungen zerlegt werden können. Somit führt ein Prozessor nicht nur eine Instruktion pro Takt durch, sondern mehrere Teile mehrerer Instruktionen gleichzeitig und steigert damit den Gesamtdurchsatz. Eine Zerlegung vereinfacht die Operation und lässt höhere Bearbeitungsgeschwindigkeit zu. Die Funktionseinheiten sind über eine Pipeline verbunden, welche wiederum unabhängig von anderen Pipelines dafür Sorge trägt, dass Aufgaben verteilt und wieder zusammengesetzt werden. Am Ende eines Taktes stünde dann optimalerweise in jeder dieser Pipelines ein fertiges Gleitkommaresultat zur Verfügung. Die theoretisch erreichbare Spitzenleistung (*peak performance*) lässt so rechnerisch ein Vielfaches der Taktfrequenz zu. Diese ist nur unter optimalen Bedingungen zu erreichen und oft Gegenstand marketingtechnischer Angaben. Trotzdem mag es mitunter aufschlußreich sein, die gemessenen Werte in ein vergleichendes, prozentuales Verhältnis zur *peak performance* zu stellen. Der in der Arbeit benutzte LINPACK-Benchmark[7] misst die Gleitkom-

[2] Im Gegensatz zur Festkommazahlen wird explizit die variable Positition des Kommas gespeichert, um aufwändige Skalierung bei Überläufen zu vermeiden.

[3] Der Begriff superskalar (*multiple issue*) ist eine "nachträgliche" Modifikation des Begriffs skalar. Skalare Architekturen sind fähig, einen Befehl pro Taktzyklus auszuführen.

maleistung und sollte nicht weniger als 80% des Peaks erreichen. Ein Abfall auf unter 50% könnte hier auf ein Problem hindeuten, wie etwa die Benutzung ineffektiver Bibliotheken (siehe dazu Absatz 3.2.2).

Beispiel: Eine CPU mit einem Takt von 750 MHz besitze zwei Gleitkommaeinheiten: Einen Addierer und einen Multiplikator. In jedem Takt kann nur eine der beiden Einheiten ein Ergebnis vorweisen, also ist die *peak-performance*:

$$R_{peak} = \frac{1 \text{ Operation}}{1 \text{ Takt}} \cdot 750 \text{ MHz} = 750 \text{ MFlop/s}$$

Hersteller und CPU Typ	*cycle-time* [MHz]	*operations per cycle*	*peak-performance* [MFlop/s]
Intel Pentium III	750	1	750
Sun UltraSparc II	400	2	800
AMD Athlon	1400	2	2800
Intel Pentium 4	2530	2	5060
Intel Xeon Quadcore	2667	4 (x4)	42672

Tabelle 1.1: Theoretische *peak-performances* einiger CPUs [DON08]. Pipelining und Multicore-Architektur verändern das Ergebnis entsprechend. Die letzte Zeile entspricht einer CPU, wie sie auf dem für die Arbeit benutzten Cluster Verwendung findet. Der Wert wäre nochmal zu verdoppeln, da jeweils zwei Sockel pro Motherboard vorhanden sind.

Mit Messungen der Gleitkommaleistung ist keine Gesamtaussage über eine Leistung aller möglichen Real-Anwendungen zu treffen, wie es beispielsweise die *Standard Performance Evaluation Corporation* (SPEC[13]) mit Ihren gemittelten Einzelleistungen verfolgt. SPEC liefert gemeinhin auch keine *flops*, sondern eine Leistung relativ zu einer Referenzmaschine. Mit LINPACK werden Operationen (das Lösen linearer Gleichungssysteme) strapaziert, die im technisch-wissenschaftlichen Umfeld sehr häufig und in großer Zahl auftreten und daher für die Leistung von Programmen ein typisches, zeitkritisches Element bedeuten. Die dazu herangezogenen Funktionssammlungen (Bibliotheken) sind von den eigentlichen Anwendungen meist entkoppelt, da sie sich sehr gut zur architekturnahen Optimierung eignen. Etwaige Eigenheiten des Systems können berücksichtigt und besondere Leistungsmerkmale ausgenutzt werden.

1.3 Parallelisierung

If you were plowing a field, what would you rather use? Two strong oxen or 1024 chickens?

Seymour Cray, 1925-1996

Die Bewältigung der Daten im Makrokosmos von Computern spaltete sich irgendwann in zwei Lager: Entweder man setzte weiterhin auf leistungsfähigere, monolithische Architektur (Vektorrechner[4] Cray-1, 1976 oder Cyber 205) oder man benutzte eine

[4]Die Geschwindigkeit dieser Computer liegt weder in ihrer Taktfrequenz noch in der Zusammenschaltung von mehreren CPUs begründet. Stattdessen besitzt die CPU mehrere Vektor-Register, die im Fall von Cray

größere Menge identischer Hardware und einheitliche Vernetzung und betrieb diese arbeitsteilend. Dieser Versuch galt dennoch lange als unwirtschaftlich gegenüber den traditionellen Supercomputern. 1983 hatte die Firma DEC ersten wirklichen Erfolg mit dem VAXCluster, welcher noch mit spezieller Kommunikationshardware ausgerüstet war. Erst ab 1996 erlangte eine Alternative zu teuren Computersystemen unter dem Namen *Beowulf-Cluster*[14] Bekanntheit. Die Verbreitung von Clustern mit preisgünstiger "Off-Shelf-Hardware[5]" fand zunehmend Anklang, weil die benutzten Arbeitsplatzrechner mittlerweile ein beachtliches Leistungspotential bei gutem Preisverhältnis vorweisen konnten. *Beowulf-Cluster* setzen die Software-Architektur *Parallel Virtual Machine* (PVM[11]) und die Kommunikationssoftware *Message Passing Interface* (MPI[12] und Absatz 3.2.1) ein.

Die horizontale Skalierung von Hardware wirft natürlich das Problem des effektiven Pipelinings - diesmal die Verteilung der Daten auf einzelne Knoten eines Computerkollektivs - wieder auf: Wie gut lassen sich gewisse Vorgänge parallelisieren? Sogenannte Parallelrechner bewältigen große Aufgaben, indem sie sie in vielen kleinen Teilaufgaben bearbeiten. Abgesehen von der Herausforderung, das Grundproblem in disjunkte Aufgaben zu zerteilen, kann nicht davon ausgegangen werden, dass die Teilprobleme auch unabhängig voneinander lösbar sind. Ein Austausch von Zwischenergebnissen kann unter beliebigen Teilmengen der Prozesse notwendig sein. Prozesse rechnen auf CPUs, die sich nicht einmal alle im selben Gehäuse befinden müssen. Das Senden und Empfangen von Nachrichten untereinander tritt also als wichtiger Faktor beim parallelen Rechnen hinzu. Der Zugriff auf die gesamten Daten eines Clusters ist sicherlich mit mehr Kosten verbunden, als es bei einem traditionellen Supercomputer der Fall wäre. Man kann und sollte jedoch zwei Arten von Datenaustausch unterscheiden: Lokal im Hauptspeicher und entfernt mit einem anderen Knoten.

Systeme, mit mehr als einem Prozessor pro Knoten, nutzen den lokalen Speicher teilweise gemeinsamen und greifen dabei auf einen einheitlichen Adressraum zu. Bei dieser Art gemeinsamen Speichers (*shared memory*) müssen gleichzeitige Schreibzugriffe auf Speicherzellen kontrolliert und Cache-Kohärenz eingehalten werden. Solche Multiprozessor-Maschinen werden *symmetrisch* (SMP) genannt, wenn alle CPUs mit gleicher Geschwindigkeit auf den Speicher zugreifen können. Aus Preisgründen werden auch mehrere identische Kerne[6] auf einem Chip verbaut, ohne den SMP-Gedanken zu verändern. Asymmetrische Mehrkernprozessoren, zum Beispiel IBM Cell, beherbergen mehrere, unterschiedlich leistungsfähige Architekturen (Haupt- und Coprozessoren) gleichzeitig. Preiswerte SMP-Systeme kommunizieren über einen Bus, der nur einer CPU den Zugriff exklusiv erlaubt. Andere Varianten ordnen einzelnen Prozessoren eigene Speichermodule zu, auf die diese schneller zugreifen können und gestatten auch zu einem diskreten Zeitpunkt mehreren CPUs Speicherzugriffe zu "ihren" Modulen. Ist die Geschwindigkeit des Zugriffs auf eigene und fremde Module allerdings unterschiedlich, so spricht man von *non uniform memory access* (NUMA) Architekturen, wie zum Beispiel die des AMD Dual-Opteron.

Gemeinsam genutzter Speicher erfordert jedoch umso aufwändigere Technik, je höher die Anzahl der konkurrierenden CPUs liegt. Auch ist die Größe zu bewältigender Probleme, die in den lokalen Hauptspeicher passen, durch dessen technische Limitierung (und Kosten) begrenzt. Um die Anzahl der Prozessoren weiter zu steigern, muss teil-

64 Werte auf einmal aufnehmen können. Mit mehreren Vektorregistern können innerhalb eines Taktzyklus Instruktionen auf 64 Werten gleichzeitig ausgeführt werden.

[5] Bedeutet: "von der Stange".

[6] Dabei handelt es sich um komplette CPU-Replikate bis auf den Bus.

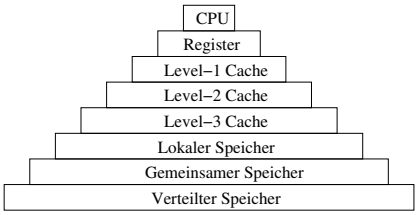

Abbildung 1.1: Speicherhierarchie moderner PC-Architekturen. Von unten nach oben wird der Speicher schneller, dafür aber teurer und kleiner.

weise auf gemeinsamen Speicher verzichtet werden. Ein Rechenknoten, bestehend aus CPUs und Speicher, kann dabei neben den lokalen Ressourcen über eine gesonderte Verbindung entfernt mit anderen Rechenknoten - speziell mit dessen Prozessen und so indirekt mit dessen Speicher - kommunizieren. Der Flaschenhals bei verteiltem Speicher (*distributed memory*) ist in der Regel die Technik zur Datenübermittlung zwischen den Computern. Das Verbindungsnetzwerk ist sehr viel langsamer als der Zugriff von CPU auf lokalen Speicher. Im Gegenzug kann jedoch für Probleme der gesamte zur Verfügung stehende Speicher aller Einzelrechner genutzt werden.

Die Kombination von *shared* und *distributed memory* lässt sich in einem Cluster aus SMP-Knoten betreiben. Die Zeiten bei Kommunikation über den Bus sind gegenüber der aufwändigen Verteilung über das Netzwerk erheblich niedriger. Dabei rangiert *shared memory* wegen des Verbindungsnetzwerks noch deutlich hinter dem lokalen Hauptspeicher bei exlusiver Nutzung. Abgesehen davon ist es immer günstiger, Speicherzugriffe zu vermeiden und stattdessen mehr Zeit mit “nützlichen” Rechnungen zu verbringen, bevor Austausch mit anderen Prozessen notwendig wird. Da die Größe des Speichers immer begrenzt ist, werden ungenutzte Objekte auf die nächste Ebene verschoben, also ausgelagert. Diese bietet für gewöhnlich mehr Platz, ist dafür aber auch meist deutlich langsamer. Muss auf bestimmte Daten zugegriffen werden, werden nacheinander von oben nach unten alle Ebenen abgefragt. Und das kostet Zeit. Um den Griff auf niedrigere Speicherlevel in der Hierarchie (Abb. 1.1) zu vermeiden, sollte ein Algorithmus Informationen so lange wie möglich vorhalten - also den Grad der Wiederbenutzung erhöhen. Sind Daten erst einmal im Register einer CPU oder im schnellsten Cache, sollten sämtliche Operationen, die diese Daten benötigen, durchgeführt werden, bevor sie wieder in den langsamen Hauptspeicher ausgewiesen werden müssen. Die schnellsten Speicher einer CPU sind aber für gewöhnlich auch immer die kleinsten. Feingranularere Aufteilungen des Hauptproblems passen in kleinere Datenpakete und lassen sich durch ihren gesunkenen Grad an Komplexität effizienter berechnen. Sie steigern aber durch die erhöhte Frequentierung und den niedrigen Wiederbenutzungswert die Interaktion mit anderen Prozessen. Gröbere Aufteilung hingegen lässt eine CPU längere Zeit mit nützlicher Arbeit verbringen, bevor ein Datenaustausch benötigt wird. Idealerweise ist der Hauptspeicher weitestgehend mit Daten befüllt. Er stellt gewissermaßen die unterste noch zu tolerierende Schranke für die Verzögerung beim Datenzugriff dar. Dennoch ist die gröbere Einteilung insgesamt erstrebenswerter, da so die für Berechnungen notwendige Zeit schneller wächst, als der Kommunikationsaufwand.

1.4 Beschleunigung und Effizienz

Der allgemeine *speedup* ist definiert durch das Verhältnis der Ausführungszeit von serieller zu paralleler Abarbeitung mit p Prozessoren als

$$S(p) = \frac{T_{seq}}{T(p)}.$$

Eine Anwendung, die auf p Knoten verteilt wird, erledigt die Arbeit im Vergleich zur sequentiellen Ausführung also in Zeit t/p. Mit den bisherigen Überlegungen ist es einleuchtend, dass auch die Verteilung über das Netzwerk zur parallelen Verarbeitung auf mehreren Computern nicht verlustfrei im Sinne von "reinem" Datenaustausch funktionieren wird. Die Arbeit muss koordiniert und schließlich gleichmäßig verteilt werden. Das kostet einerseits Ressourcen, andererseits sind dieses Lastverteilungen nicht immer ideal, so dass Kommunikation oder ungleicher Arbeitsaufwand zu Wartezeiten (hier zu $o(p)$ zusammengefasst) der Prozessoren führen. Je nachdem, wie gut der Algorithmus mit der Problemgröße skaliert, kann sich das Verhältnis von Rechenzeit zu IO-Wartezeit dennoch ab einem bestimmten Punkt als günstig herausstellen und scheinbar paradoxerweise "mehr Arbeit" die Effektivität steigern. Im besten Falle einer durchdachten Verarbeitung von Daten sorgt das Caching für einen superlinearen Speedup ($S \geq p$), wenn die Speicherzugriffszeit den limitierenden Faktor darstellt.

Massive Parallelrechner sind Systeme, in denen alle CPUs gleichzeitig Operationen mit identischer Mächtigkeit durchführen können. Paralleles Rechnen ist gut dazu geeignet, regelmäßig aufgebaute Datensätze effizient zu bearbeiten. Die Summe der Einzelleistungen wird aber trotzdem selten erreicht. Ein Aspekt ist die Parallelisierbarkeit des Programms selbst: Nicht alle Anweisungen lassen sich gleichzeitig ausführen.

Amdahl's Gesetz 1967

Nach Amdahl[7] hängt der Geschwindigkeitszuwachs durch Parallelisierung vor allem vom sequentiellen Anteil des Problems ab. Dieser begrenzt die maximale Beschleunigung als obere Schranke. Sei σ die Zeit, die ein Programm im sequentiellen Teil verbringt und π das Pendant des parallelisierbaren Teils. Dann ergibt sich Amdahl's Speedup [GA67] bei einer Ausführung auf p Prozessoren zu

$$S(p) = \frac{1}{a + \frac{1-a}{p}} \leq \frac{1}{a}, \tag{1.1}$$

mit einem relativen Anteil

$$a = \frac{\sigma}{\sigma + \pi}$$

nicht-parallelisierbarer Anweisungen. Diese Formel besagt nun, dass die Zeit für parallele Ausführung beliebig klein werden kann (durch Erhöhung der Ressourcen), der sequentielle Anteil aber immer bleibt, wie er ist. Im sequentiellen Anteil enthalten ist die Arbeit, parallele Bearbeitung zu koordinieren und kann - so der Schluss - daher nie verkleinert werden. Das bedeutet, dass ein sequentieller Anteil von 10% den Speedup auf max. 10 begrenzt, egal wie viele Prozessoren für die parallele Verarbeitung zur

[7] Gene Myron Amdahl, USA, Computerarchitekt und Hi-tech-Unternehmer.

Verfügung stehen. Beispielsweise führt jede triviale Aufspaltung von Matrix Daten zur parallelen Berechnung früher oder später zu Amdahls Beschränkung. In anderen Worten: Solange die Berechnungen nicht unabhängig voneinander ausführbar sind, bringt das Hinzufügen von Prozessoren ab einem gewissen Punkt keine zusätzliche Leistung mehr.

Karp-Flatt-Metrik 1990

Wenn man Amdahls Formel nach a umstellt erhält man die Karp-Flatt[8]-Metrik[KF90]

$$a = \frac{1/S(p) - 1/p}{1 - 1/p}. \tag{1.2}$$

Diese Metrik gibt Hinweise darauf, ob die Leistung von einer ungleichen Lastverteilung oder von Kommunikationskosten dominiert wird. Weder Amdahl noch Gustafson (siehe unten) haben solchen Overhead berücksichtigt. Mit dieser Abhängigeit von p können eventuell Schlüsse gezogen werden, welche die Auswertung von Benchmarks unterstützen.

Gustafson's Gesetz 1988

Amdahl kam seinerzeit zu dem Schluss, dass sich massive Parallelität nicht lohne, da der serielle Anteil niemals zu elimieren sei. Wenn man den Ansatz etwas ändert und statt konstante Problemgröße mit höherem Ressourcenaufwand in immer kürzeren Zeiten zu lösen, eher wachsende Problemgrößen bei konstanter Laufzeit betrachtet, kommt man zum *skalierten Speedup* von Gustafson[9][JG88]

$$S(p) = a + p(1-a) = p + (1-p) \cdot a. \tag{1.3}$$

Die Formel vergleicht die parallele Ausführungszeit mit der Entsprechenden auf einer sequentiellen Maschine - nicht andersrum, wie Amdahl es tut. Daher kommt Gustafson bei steigender Problemgröße und damit einem angenommenen Verschwinden des sequentiellen Anteils auf einen linearen Zusammenhang von p und $S(p)$. Die hypothetische[10] Laufzeit auf einer sequentiellen Maschine würde linear steigen. Der Speedup erreicht wie erwünscht p, wenn n gegen Unendlich geht.

Im Allgemeinen wird also die Effizienz mit zunehmender Anzahl von Prozessoren bei konstanter Problemgröße abnehmen, mit steigender Problemgröße dagegen zunehmen. Wenn das parallele System skalierbar ist, dann kann die Effizienzminderung bei Prozessorvermehrung durch eine Erhöhung der Problemgröße abgefangen werden. Das bedeutet mehr Ressourcen ohne die damit verbundenen Nachteile. Bei einer günstigen Granularität verkürzen mehr verfügbare Recheneinheiten natürlich die Laufzeit, die minimale Laufzeit sollte dabei aber kostenoptimal sein. Skalierbare parallele Systeme

[8] Alan H. Karp und Horace P. Flatt.

[9] John L. Gustafson, Computerwissenschaftler und Geschäftsmann, hauptsächlich bekannt durch seine Arbeit im Bereich High-Performance-Computing, CEO bei Massively Parallel Technologies, Inc.

[10] Die Problemgröße kann so groß sein, dass kein seq. Rechner sie mehr ausführen könnte.

sind kostenoptimal bei geeigneter Wahl von p und Problemgröße N, das heißt die Kosten[11] wachsen nicht stärker als ein konstanter Faktor ($\theta(1)$) im Vergleich zur besten sequentiellen Lösung.

1.4.1 Isoeffizienz

Um die Effizienz konstant zu halten, muss also das Problem N im richtigen Verhältnis zu p wachsen. Sei W die Laufzeit, die der beste sequentielle Algorithmus benötigt und $T_0(W,p)$ die Zusammenfassung der Verluste durch Kommunikation oder ungleiche Lastverteilung bei paralleler Berechnung. Dann ist

$$\begin{aligned} \text{die parallele Laufzeit } T_p &= \tfrac{W+T_0(W,p)}{p}, \\ \text{der Speedup } S &= \tfrac{W}{T_p} &&= \frac{W\cdot p}{W+T_0(W,p)}, \\ \text{und die Effizienz } E &= \tfrac{S}{p} &&= \frac{W}{W+T_0(W,p)} = \frac{1}{1+T_0(W,p)/W}. \end{aligned}$$

Bei gegebener Effizienz E bedeutet dies

$$W = \tfrac{E}{1-E}\cdot T_0(W,p) = CT_0(W,p). \tag{1.4}$$

Die Problemgröße N ist eine Funktion von W und das heißt, N muss (mindestens) so schnell wachsen, dass die sequentielle Laufzeit genauso schnell wächst wie die Gesamtzeit der parallelen Verluste. Gleichung 1.4 ist die Isoeffizienz-Funktion. Die Laufzeit ist dabei nicht zwangsläufig proportional zu N. Die Bearbeitungszeit einer Matrix mit dem Gauß'schen Eliminiationsverfahren beispielsweise wächst kubisch mit der Problemgröße N.

1.5 Verbindungsnetzwerke

Die unterschiedliche Bandbreite von Netzwerken spielt hier eine untergeordnete Rolle, als dass es nicht um Durchsatz sondern um die Latenz der Übermittlung geht. Einzelne Verbindungsaufbauten können nicht zusammengefasst werden. Der bisher betrachtete parallele Overhead besteht zum großen Teil aus den kumulierten Idle-Zeiten von Prozessoren durch Verzögerung bei der Übertragung. Häufig vorkommende Techniken im HPC Bereich sind Myrinet oder Infiniband, wobei Letzteres sich am Markt langsam durchsetzen konnte. Weiterhin ist traditionelles Ethernet anzutreffen. Eine Gigabit-Variante wird in Clustern wohl am häufigsten verbaut, aber erst die 10 GBit Version wird eine erneute Alternative zu Infiniband darstellen. Infiniband, eine serielle Bustechnik über Kupfer, kann, im Gegensatz zu Gigabit-Ethernet, eine deutlich latenzärmere ($< 2\mu s$) Kommunikation aufweisen, da der Protokollstack im Gegensatz zu TCP/IP auf die Netzwerkhardware ausgelagert wird. Außerdem übernimmt die Technik selbst die Umsetzung von virtuellen in physikalische Adressen und greift per DMA auf den

[11] Das Produkt aus Laufzeit und Anzahl der Prozessoren ist ein Maß für die durchgeführte Arbeit.

Hauptspeicher zu. Ethernet stellt durch die hohe Signallaufzeit oft den limitierenden Faktor bei der Skalierung dar. Infiniband nimmt in der Top 500 nur den zweithöchsten Anteil an Interconnects hinter Ethernet ein.

Die in Clustern zu bevorzugende latenzarme Übertragungstechnik begünstigt im speziellen MPI Kollektiv-Funktionen, wie `Bcast`[12] und `Scatter`[13].

1.6 Unterschiedliche Compiler

Unterschiedliche Compiler erzeugen unterschiedlichen Maschinencode. Das trifft insbesondere dann zu, wenn man die Optimierungsstufen nutzt und zusätzliche *Flags* setzen möchte. Da die Art und Tiefe der Optimierungen variieren, sollen hier drei für die Arbeit genutzte Parameter kurz angerissen werden:

Compiler-Optimierung `-O{1-3}` ist ein Schalter, der gewöhnlich viele andere Optimierungsversuche des Compilers steuert. Der Compiler versucht hierbei, den Code nach bestem Wissen so umzustrukturieren, dass die Ausführungszeit sich minimiert. Zum Beispiel werden Speicherzugriffe vorgezogen oder verzögert, schnellere (Maschinen-)Anweisungen substituiert, das Paging verbessert oder Register besser genutzt.

Loop-Unrolling `-funroll-loops` oder `-Munroll` sind Flags, die dazu dienen, Schleifen teilweise oder sogar komplett aufzulösen. Die Anweisungen innerhalb der Schleife werden ausgeführt, ohne dass jedes Mal nach den Anweisungen eine Prüfung der Schleifenbedingung und ein Sprung zum Schleifenbeginn erfolgen muss [DLP03].

No-Frames `-fomit-frame-pointer` oder `-Mnoframe` sind Flags, um die sogenannten Frame-Pointer zu verwerfen, wenn sie nicht benötigt werden. Auf diesem Wege wird ein zusätzliches Register frei. Das Frame-Pointer-Register wird benutzt um auf Parameter für Funktionen oder lokale Variablen zuzugreifen.

Optimierungen finden auch im Bezug auf die Hardware statt, für die übersetzt wird. Besondere Eigenschaften, wie die Anzahl der Prozessor-Register, werden dabei beachtet. Möglicherweise erzeugt der Compiler auch langsameren Code, als ohne Optimierungen. Das kann an höheren Cache-Ladezeiten liegen, die sich erst bei häufigerem Zugriff positiv auswirken. Compiler Optimierungen sind ein weites Feld und umfangreich dokumentiert. Sie stellen für sich genommen schon wieder eine eigene Klasse des Benchmarkings dar und die Auswirkungen sind oft ebenso empirischer Natur. Sicherlich beeinflusst ausgiebiges Compiler-Tuning bei der Übersetzung von Bibliotheken die erzielte Leistung der aufsetzenden Anwendung zum Teil in erheblichem Maße. Für die praktischen Tests sollen jedoch der frei erhältliche Gnu Compiler und der kommerzielle Portland Group Compiler mit den genannten Standardoptimierungen (`-O3`) gegenübergestellt werden. Dabei werden sowohl der Benchmark als auch die kritischen Bibliotheken einheitlich übersetzt, sowie vorkompilierte Versionen von OpenMPI benutzt. Der Intel C Compiler stand aus Lizenzgründen nicht zum Test zur Verfügung[14].

[12] Ein MPI-Prozess schickt gleiche Daten an andere Prozesse aus seiner Gruppe.
[13] Wie zuvor, nur mit unterschiedlichen Daten.
[14] Wobei Intel-Compiler-optimierte Bibliotheken benutzt werden konnten (Absatz 3.2.2).

2 Die Top 500

Seit 1986 existiert eine Liste der schnellsten Computersysteme mit Ihren Leistungswerten. Anfangs in einer jährlich publizierten Statistik, seit 1993 erscheint sie aufgrund der zunehmenden Verbreitung paralleler Systeme durch die eigens gegründete Organisation *TOP500* zweimal im Jahr (Juni und November) abwechselnd in Deutschland und den USA. Die Anfänge machten seinerzeit ausschließlich Vektorrechner. Das Ranking erfolgt durch Ermittlung der Gleitkommaleistung. Die Werte basieren dabei auf denen der Hersteller der Systeme. Der Benchmark besitzt eine lange Historie, was ihn einzigartig macht. So sind Vergleiche von 1993 bis heute möglich.

Da der Top 500 Liste das LINPACK-Ergebnis zugrunde liegt, rücken andere Leistungsmerkmale des parallelen Systems in den Hintergrund. Außerdem muss der Benchmark auf dem System ausführbar sein, was zu einer Verfälschung der Rangliste durch Ausnahmesysteme führte, die es mit ihrer reinen Gleitkommaleistung leicht auf diese schaffen würden. Allerdings beabsichtigt LINPACK ausdrücklich nicht einen generellen Leistungseindruck vom System zu ermitteln. In dieser, für Naturwissenschaften wichtigen Eigenschaft der Gleitkommaleistung soll vielmehr ein Mechanismus existieren, der diese Daten mit derselben arithmetischen Präzision erhebt und damit vergleichbar macht. Man mag das Lösen linearere Gleichungssysteme als den kleinsten gemeinsamen Nenner beim Supercomputing sehen. Selbst wenn dabei andere Algorithmen zum Einsatz kommen und die Systeme eine andere Struktur aufweisen, sind die Ergebnisse doch bedingt übertragbar. Ein weiterer Vorteil für die meist öffentlich finanzierten Supercomputer ist, dass man auch Laien die Kenngröße “Gleitkommaberechnungen pro Sekunde” gut erklären kann. Dies fällt den PR-Abteilungen bei Speicherbandbreite und -latenz oder gar bei mehrdimensionalen Benchmark-Ergebnissen deutlich schwerer.

Auf der Liste angegeben ist - soweit verfügbar - zusätzlich die maximale Problemgröße N_{max}, bei der die höchste Leistung R_{max} erzielt wurde. Außerdem werden die theoretische Leistung des Systems R_{peak} (*peak performance*) und die Systemgröße $N_{1/2}$ angegeben, bei der die Gleitkommaleistung auf $R_{max/2}$ abfällt.

System	Cores	Processor	R_{max}	N_{max}
BladeCenter QS22/LS21	129600	PowerXCell 8i	1105000	2329599
Cray XT5 QC	150152	AMD x86_64	1059000	4712799
SGI Altix ICE 8200EX	51200	Intel EM64T	487005	2300760

Tabelle 2.1: Internationale Top 3 vom November 2008

Rank	System	Cores	Processor	R_{max}	N_{max}
11	Blue Gene/P Solution Forschungszentrum Jülich	65536	PowerPC 450	180000	1766399
23	Power 575 RZG/Max-Planck-Gesellschaft	6720	Power6	92980	-
44	Altix 4700 Leibniz Rechenzentrum	9728	Intel IA-64	56520	1583232

Tabelle 2.2: Deutsche Top 3 vom November 2008

Abbildung 2.1: Spannweiten der Top-500 Liste seit 1993
[Quelle: http://www.top500.org]

Jahr	Minimum R_{max}	Minimum EM64T	Cores/Takt
Jun 06	2026	2050	400@3200
Nov 06	2736	2747	698@3200
Jun 07	4005	4087	542@3000
Nov 07	5929	5229	1344@1866
Jun 08	8996	8996	1488@3000

Tabelle 2.3: Einstiegsleistungen in die Top-500 Liste der letzten Jahre. Es wird das allgemeine Minimum betrachtet, sowie der langsamste Vertreter der EM64T Familie mit Angabe von Core-Anzahl und Taktfrequenz. Das maximale Problem, mit dessen Angabe man auf den zur Verfügung gestandenen Gesamtspeicher schließen kann, ist leider bei beinahe allen dieser Systeme nicht aufgeführt worden.

3 Der LINPACK Benchmark

3.1 Herkunft

Das ***Lin****ear Algebra* ***Pack****age* war zunächst eine Numerik-Bibliothek für die Lösung linearer Gleichungssysteme. Mittlerweile ist LINPACK in diesem Bereich durch die Weiterentwicklung LAPACK abgelöst worden. Der Name hat sich aber im Bereich des damit betriebenen Benchmarkings gehalten.

Der *LINPACK User's Guide* wurde 1979 zusammen mit dem Quellcode eines in Fortran geschriebenen Benchmarks veröffentlicht. Dieses Programm sollte einen Eindruck von der Gleitkommaleistung eines Rechners geben und hatte die Problemgröße N=100 fest vorgegeben. Die meisten der Gleitkommaoperationen in LINPACK werden dabei in einer extra Funktionssammlung, der *Basic Linear Algebra Subprograms* (BLAS, Absatz 3.2.2) ausgeführt, welche kontinuierlich aufgerufen wird. Eine Modifikation des Programms war für die Leistungsermittlung manuell nicht erlaubt - lediglich Optimierungen seitens des Compilers. Die Größe war seinerzeit durch den verfügbaren Speicher begrenzt (10000 Elemente bei 64 Bit Genauigkeit[1]) und stellte ein "hinreichend großes Problem" dar. Dieser Benchmark ist insofern unbrauchbar für moderne Prozessoren, als dass die Ausführungszeiten zu klein für die mitgelieferte Zeitmessungsmethode geworden sind. Hier muss die Methode `SECOND` manuell durch eine mit höherer Auflösung ersetzt werden. Später wurden LINPACK-Benchmarks mit einer LINPACK-Version durchgeführt, die die Problemgröße von N=1000 vorgab, aber durchaus manuell die LR-Zerlegung und Lösungsroutine für lineare Gleichungen zu optimieren oder gar ersetzen gestattete. Die Anzahl der zur Lösung notwendigen Schritte musste dabei natürlich beachtet werden. Andere Namen für das Programm waren *Toward Peak Performance* oder *Best Effort*. Multiprozessor Implemetierungen waren ebenfalls erlaubt.

Für paralleles Rechnen durfte die Schleife um die `SAXPY`-Methode[2] nun parallelisiert, die Matrix in Subpanels verteilt werden. Weitere Ausführungen dazu unter Absatz 3.3. Das Ganze mündete schließlich im *High Performance Benchmark* (HPL), welcher die Problemgröße zu variieren erlaubte, um dem Leistungspotential der aufkommenden Gattung von Massiv-Parallel-Systemen gerecht zu werden. Die Regeln waren die folgenden: Es sollen lineare Gleichungen mit einer beliebigen Methode bei variabler Anzahl Unbekannter gelöst werden und dabei ist die Ausführungszeit zu messen. Der Algorithmus muss $2n^3/3 + 2n^2$ Operationen verwenden um die Gleitkommaleistung zu

[1] Auf einer Cray bedeutet dies' Single-Precision, auf IBM dagegen bereits Double-Precision.

[2] Skalar Alpha X + Y, ein generischer Name für Skalarmultiplikation mit Vektoraddition. Das "S" bedeutet eigentlich *single precision*.

bestimmen. Das Residuum ist dabei geringer als $N\varepsilon$, mit ε der Maschinengenauigkeit[3].

Bekannt geworden ist der LINPACK hauptsächlich durch seine Verwendung bei der Erstellung der Top-500 Liste. Mittlerweile existiert der *High-Performance LINPACK* (HPL) in Version 2.0 für Maschinen mit verteiltem Speicher und ist beinahe ausschließlich in der Programmiersprache C geschrieben.

3.2 Voraussetzungen

Im Wesentlichen benötigt der HPL zwei zusätzliche Dinge: Eine MPI-Bibliothek und eine (optimierte) Version von BLAS. Die Installation des HPL wird über ein Makefile gesteuert, in welcher die Pfade zu diesen Ressourcen manuell einzutragen sind. Das betrifft im wesentlichen die MPI-fähigen C und Fortran Compiler, sowie die statischen oder dynamischen BLAS Bibliotheken.

3.2.1 Message Parsing Interface

Für die Kommunikation unter den Prozessen auf einem Parallel-Rechner kommt häufig eine Implementierung des *Message Parsing Interface* (MPI) zum Einsatz. Der MPI-Standard[4] legt Syntax und Semantik für C, C++ oder Fortran Interfaces fest, um den Datenaustausch von parallelisierten Prozessen für den Gesamtfortschritt zu regeln. MPI beeinflusst einen nicht zu vernachlässigenden Anteil am Kommunikationsoverhead und sollte - ebenso wie es für die leistungskritischen Linear Algebra Funktionen gilt - auf das jeweilige System optimiert sein. Es existieren eine ganze Reihe von Umsetzungen; Die Bekanntesten sind wohl die freie Softwarebibliothek MPICH, welche zuerst den MPI-Standard 1 umsetzte, und LAM/MPI, einem der Vorgänger des comunity-geleiteten OpenMPI. Letzteres implementiert MPI-1 und Teile von MPI-2. OpenMPI unterstützt die in HPC-Clustern gängigen Protokolle wie TCP/IP/Ethernet, Myrinet, Infiniband oder Shared-Memory um Informationen auszutauschen. Dabei nutzt es im Allgemeinen alle vorhandenen Techniken auch gleichzeitig aus. Zum Beispiel die Kommunikation zwischen lokalen Prozessen eines Rechners über Shared-Memory, Datenaustausch zwischen Knoten per Infiniband und Kontrollinformationen noch über TCP/Ethernet.

OpenMPI 3.0 steht zur Zeit als Pre-Release zur Verfügung und ist nicht zu verwecheln mit OpenMP, einer Multi-Prozessor Programmierschnittstelle, die Parallelisierung auf Thread- bzw. Schleifenebene lokal im *shared-memory* betreibt. Hybride Systeme mit MP und MPI Programmcode sind jedoch nicht selten bei modernen Parallelrechnern anzutreffen.

3.2.2 Basic Linear Algebra Subroutines

Die BLAS stellen den leistungskritischen Unterbau für den LINPACK Benchmark, da die meisten Gleitkommarechnungen mit ihnen ausgeführt werden. Diese Biliothek

[3] IEEE standard double precision $= 2^{-53}$.

[4] Message Passing Interface (MPI) ist ein Standard[6], der den Nachrichtenaustausch bei parallelen Berechnungen auf verteilten Computersystemen beschreibt. Er legt dabei eine Sammlung von Operationen und ihre Semantik, also eine Programmierschnittstelle fest, aber kein konkretes Protokoll und keine Implementierung.

stellt Standardroutinen für Vektor- und Matrix-Operationen zur Verfügung, welche in verschiedenen Klassen (von Dongarra als sogenannte *level* bezeichnet) in Anlehnung an die Dimension der Vektorberechnung aufgeteilt ist. Die Unterscheidung hat den Versuch der Optimierung der Speicherreferenzierung als Ursache. Während der Faktorisierung einer Matrix beispielsweise müssen Blöcke gelesen, aktualisiert und zurückgeschrieben werden. Und zwar im selben Maße wie die Anzahl der Gleitkommaoperationen und ohne sichtbar hohen Anteil an Wiederverwendung. Es wird deutlich, dass das Verhältnis von Operation zu Speicherzugriff sich erhöhen muss, wenn man eine Steigerung der Leistung erzielen möchte. Superskalere Rechner leiden hier besonders unter der großen Anzahl von Datenbewegungen und Aktualisierungen. Die drei Level der BLAS versuchen nun, diese unterschiedlichen Anforderungen durch Anpassung der Speicherreferenzierung zu adressieren: In Level 1 BLAS werden ausschließlich Skalar-, Vektor- und Vektor-Vektor-Operationen, in Level 2 BLAS Matrix-Vektor- und Level 3 BLAS Matrix-Matrix-Operationen durchgeführt. Ab Level 2 steigt der Grad der Wiederbenutzung, da nun mehrere Operationen mit vorgehaltenen Daten ausgeführt werden. Level 3 ist bekannt als dasjenige mit der besten Cache-Optimierung. Viele Algorithmen der numerischen linearen Algebra können effektiv auf Level 2 Operationen umgeschrieben werden. Diese Technik kommt vor allem Vektorrechnern zugute. Auf parallelen Systemen hat sich die blockweise und damit Matrix-Matrix-Abarbeitung von Level 3 jedoch als effektiver erwiesen. Ausführungen distinkter Blöcke und die darin ausgeführten Level 1 Operationen können jeweils parallelisiert werden - Blöcke werden für Wiederbenutzung komplett im Cache bzw. lokalen Speicher gehalten. Die rekursive LR-Zerlegung hat sich als leistungsstärker erwiesen, als die Blockmethode[5], da sie weniger Speicherzugriffe benötigt. Es existiert zudem die Parallel-BLAS, welche statt serieller Routinen in parallelen Threads arbeitet.

Die Art und Weise dieser Optimierungen machen klar, dass eine genaue Kenntnis der unterliegenen Hardware-Struktur notwendig ist. So existieren maschinenspezifische Versionen von BLAS, sowieso verschiedene Ansätze der Optimierung. Zwei seien nachstehend kurz hervorgehoben:

Automatically Tuned Linear Algebra Software

Die ATLAS-Bibliothek ist eine portable Implementation von BLAS und einigen LAPACK Routinen und kann auf Wunsch auch einen parallelen Ansatz per Threading verfolgen. Beim Übersetzen dieser Bibliothek werden verschiedene Tests durchgeführt, um auf empirische Weise eine schnellstmögliche Realisierung zu finden. Dieses stellt den interessanten Versuch einer automatischen Selbst-Adaption und Optimierung dar. ATLAS ist mit die leistungsfähigste Umsetzung aller BLAS-Varianten und kann auf vielen Architekturen mit den herstelleroptimierten Bibliotheken mithalten[WPD00].

Intel Kernel Math Library

Die Imkl beeinhaltet unter anderem auch Pakete für Lineare Algebra und ist für Intel-Architekuren optimiert. Eine derart hardwarenahe Implementierung beinhaltet viel manuelles Feintuning und wird aus diversen Gründen vom Hardware-Hersteller gepflegt. Von diesen Bemühungen profitieren natürlich andere Bereiche der Softwareentwicklung ebenso.

[5] Hier ist ein Algorithmus gemeint, nicht die blockweise Aufteilung der Matrix in Teilmatrizen.

3.3 Der Algorithmus von LINPACK

Das LINPACK Paket bestand ursprünglich aus einer Sammlung von Fortran Funktionen um lineare Gleichungssysteme $A \cdot x = b$ mittels Gauß'schem Eliminationsverfahren zu lösen. Durch Dekomposition wird das ursprüngliche Problem - eine randomisierte Matrix A der Dimension N, in ein Produkt von (Teil-)Matrizen zerlegt. Diese lassen sich durch ihre einfache Struktur leichter manipulieren um das originäre Problem zu lösen. Dabei handelt es sich um N^2 lineare Gleichungen, gebildet aus der vollbesetzten und quadratischen Matrix $A \in \mathbb{R}^{N \times N}$ und den Vektoren $x, b \in \mathbb{R}^N$. Die Koeffizientenmatrix A wird per LR-Zerlegung in die obere linke L und die untere rechte Dreiecksmatrix R faktorisiert, so dass $A \cdot x = (L \cdot R) \cdot x = L \cdot (R \cdot x) = b$. Anschließend wird durch Vorwärtssubstitution der Vektor $y = R \cdot x$ berechnet, der $L \cdot y = b$ erfüllt und durch Rückwärtssubstitution in $y = R \cdot x$ die Lösung ergibt. Der Aufwand für die LR-Zerlegung beträgt $2/3 \cdot N^3$ Operationen. Die Vor- und Rückwärtssubstitutionen benötigen zusätzlich noch $2N^2$ Operationen. Insgesamt arbeitet der Algorithmus also in $O(N^3)$. Der Aufwand - und damit abzuschätzen, die Zeit - wächst also etwa kubisch mit der Problemgröße.

Durch leichte Änderung an der sequentiellen Ausführung können die Ergebnisse in der ursprünglichen Matrix A selbst gespeichert werden. Da diese bereits vollbesetzt erzeugt wurde, muss kein zusätzlicher Speicher alloziert werden. Der verfügbare Speicher (die Summe aller lokalen Speicher M_i in einem Verbund) begrenzt die Anzahl der vorgehaltenen Matrix-Elemente mit doppelter Genauigkeit (8 Byte) und kann bestimmt werden durch

$$N \approx \sqrt{\frac{0.9 \cdot p \cdot M_i}{8}}. \tag{3.1}$$

Je nach Hauptspeicherbelegung durch das Betriebssystem, welche für einen Benchmark möglichst gering ausfallen sollte, können mehr oder weniger als die hier veranschlagten 90% zur Berechnung der Matrix genutzt werden. Durch vorhergehende Betrachtungen ist bekannt, dass der Algorithmus bessere Ergebnisse erzielt, je größer das Problem ist. Zu hohe Werte zwingen jedoch das Betriebssystem dazu, Speicher auszulagern, was die Benchmarkergebnisse verfälscht.

Um das Problem parallel zu lösen, muss die Matrix A jedoch zuvor auf die einzelnen Knoten sinnvoll verteilt werden. Dazu wird sie logisch in $(N/N_B) \times (N/N_B)$ Teilmatrizen $A_{i,j}$ (sogenannte *panels*) zerlegt, mit N_B der konfigurierbaren Blockgröße. Die Arbeit wird den verfügbaren Prozessoren zyklisch zugewiesen, indem die Teilmatrizen zeilen- oder spaltenweise auf ein Gitter angeordnet werden, in dem jedes Feld einem Prozessor entspricht (Abb. 3.1). Die blockweise Verarbeitung kommt modernen Speicherarchitekturen mit Hierarchie entgegen, da Hauptspeicherzugriffe (cache miss) bei Standardimplementationen mit zeilen- oder spaltenweiser Abarbeitung zu häufig Daten benötigen, die sich nicht im Cache befinden. Gerade bei großen Matrizen ist die Zeilenlänge dabei ein Problem. Kleine Blockgrößen begünstigen die Lastverteilung, sind aber häufiger auszutauschen und haben entsprechend ihrer Größe nur einen geringen Wiederverwendungswert (siehe Abschnitt 1.3). Von sehr großen Blöcken wird in der HPL Dokumentation abgeraten. Die optimale Größe ist natürlich bedingt durch das Rechenzeit-zu-Kommunikationsoverhead-Leistungsverhältnis des jeweiligen Systems.

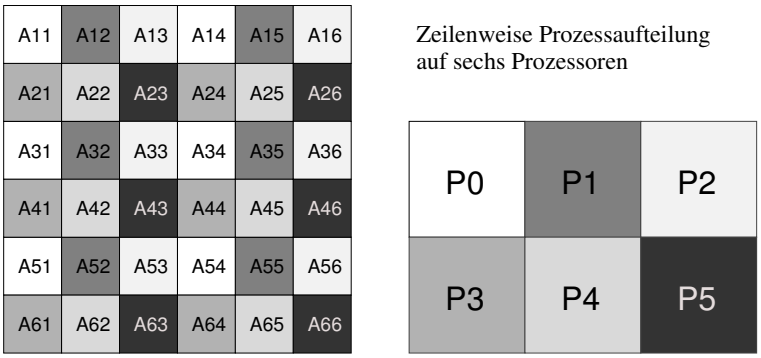

Abbildung 3.1: Zyklische (reihenweise) Aufteilung der Teilblöcke $A_{i,j}$ auf das Prozessorgitter, hier mit $P \times Q = 2 \times 3$. Ein Teilblock hat dabei die Dimension N_B

.

Um numerisch stabil zu sein, wird partiell pivotisiert, das heißt die Zeilen werden so vertauscht, dass das aktuelle Diagonalelement in der Spalte das Maximum darstellt. Bei jeder Iteration der Hauptschleife werden N_B Zeilen/Spalten faktorisiert. So ein Block wird in der HPL Dokumentation *panel* genannt. Die Panel-Faktorisierung geschieht rekursiv mit einem Matrix-Matrix basierten Algorithmus, der in jedem Schritt N_{div} neue Teilprobleme erzeugt, bis eine Mindestanzahl N_{Bmin} an Spalten vorhanden und so die maximale Rekursionstiefe erreicht ist. Anschließend wird ein Matrix-Vektor-Algorithmus das *Mindest*problem lösen. Für beide dieser Algorithmen kann aus drei numerisch äquivalenten Verfahren gewählt werden: *right-looking*, *left-looking* oder *crout*. Abbildung 3.2 veranschaulicht die Arbeitsweise dieser Verfahren etwas, um eine ungefähren Eindruck zu vermitteln.

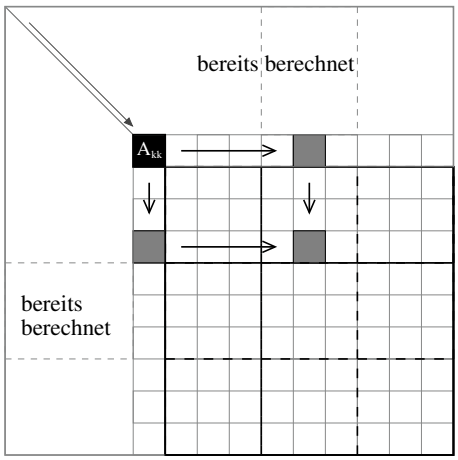

Abbildung 3.2: In jedem Schritt der Hauptschleife wird ein diagonaler Block A_{kk} LR-faktorisiert. Eine Zeilen-Pivotisierung muss für bereits berechnete und unbehandelte Submatrizen $A_{ik}, i \neq k$ weitergegeben werden. Außerdem müssen alle Prozesse, die Submatrizen $A_{ik}, i > k$ speichern, über die Änderungen informiert werden. Im Beispiel können Änderungen von A_{kk} an $A_{k+5,k}$ und $A_{k,k+3}$ weitergereicht werden, welche wiederum für eine Änderung in $A_{k+5,k+3}$ sorgen.

Bei *crout* sind alle Diagonalelemente der Matrix R mit eins vorbelegt. Der Unterschied

zur *right-looking* Variante besteht unter anderem darin, dass Folge-Submatrizen nur über Zeilenvertauschungen informiert und nicht transformiert werden. Die genaue Darstellung der Techniken soll hier nicht weiter angesprochen werden. Im Wesentlichen differieren sie in der Art, welche anderen Prozesse über Aktualisierungen informiert werden müssen. Während der Panel-Faktorisierung erfolgt die Kommunikation entlang der Spalten, danach entlang der Zeilen des Prozessgitters. Es liegen teils deutliche Leistungsunterschiede zwischen den einzelnen Prozessgittern. In einem SMP-System ist es ratsam, die Blöcke so aufzuteilen, dass Nachbarn im selben lokalen Speicher liegen, sprich: Eine Multiprozessor-Maschine profitiert am stärksten von aufeinander folgenden Daten in einer Reihe dadurch, dass die zugehörigen Prozesse nicht über das langsame Verbindungsnetzwerk sprechen müssen. Die spaltenweise Zuteilung von Aufgaben zu Prozessoren, bringt immerhin noch einen kleinen Vorteil dadurch, dass während der Faktorisierung Information spaltenweise ausgetauscht werden. Sollten die Daten weder in Reihe noch spaltenweise im Hauptspeicher einer SMP-Maschine liegen, so entfällt der lokale Vorteil komplett und alle Prozesse reden über das Protokoll des Verbindungsnetzwerkes miteinander.

Ferner ist daher auch einzusehen, dass auf einer SMP-Maschine möglichst soviele Prozesse gestartet werden, wie sie der Anzahl der Prozessoren entspricht. Dabei gibt es neben mehrfachen MPI-Prozessen noch die Möglichkeit, nur einen zu starten und eine parallelisierende BLAS-Version zu nutzen. Die PBLAS-Threads können dabei von den unterschiedlichen Cores bearbeitet werden. Mehr verspricht in diesem Fall allerdings der Komfort von lokalem Datentransfer durch mehrere MPI-Prozesse. Es existiert dann nämlich ein Szenario, in dem **nicht** alle Prozesse identischen Aufwand haben um Daten auszutauschen.

In Bezug auf das Netzwerk bevorzugen[4] sternförmige Topologien, wie zum Beispiel über einen Ethernet-Switch, in etwa gleich große P und Q mit $Q \geq P$. In einem ringförmigen Netzwerk skaliert HPL besser mit einem flachem Prozessgitter (a$\times b$, mit $b \gg a$). Unter recht allgemeinen Annahmen zur verwendeten Hardware kann man zeigen [DLP03], dass die vom HPL-Benchmark aufgewendete Zeit für eine Problemgröße N etwa

$$t = \gamma \frac{2N^3}{3PQ} + \beta \frac{N^2(3P+Q)}{2PQ} + \alpha \frac{N((N_B+1)\log P + P)}{N_B} \tag{3.2}$$

beträgt. Die erster Konstante γ ist die inverse Gleitkommaleistung[6] des Knotens. In diesem Teil spielt das Gitter keine Rolle. Der Zeitbedarf steigt kubisch mit der Problemgröße. Für die Latenz α zeichnet sich maßgeblich die Zeilenanzahl P verantwortlich, allerdings fällt der Teil von der Größe nicht so sehr ins Gewicht, wie der Mittlere. Die zweite Konstante β bezeichnet die inverse Bandbreite bei der Kommunikation zwischen den Knoten. Dieser Teil der Formel sorgt wegen N^2 für hohe Zeitstrafen bei asymmetrischem Gitter. Durch eine Wahl der Spaltenanzahl $Q \approx 2P$, bzw. $P \approx Q \approx \sqrt{p}$ bei p Prozessen lässt sich dieser Aufwand jedoch minimieren. Durch Umstellen von 3.2 lässt sich außerdem zeigen, dass bei konstant gehaltener Speichernutzung pro Prozessor $N^2/(PQ)$, der Beta-Term ebenfalls konstant bleibt. Damit wird ausgedrückt, dass HPL nicht nur hinsichtlich Berechnungsmenge sondern auch in Bezug auf das Kommunikationsvolumen skalierbar ist.

[6] Genauer: Eine Gleitkommaoperation auf Level 3 Matrix-Matrix.

Die Benachrichtigungen über Zeilentausch oder die Aktualisierung der Matrizen an die jeweiligen Prozesse werden durch sogenannte *Broadcasts* verteilt. HPL kennt auch hier mehrere Varianten, von denen zwei in Abb. 3.3 gezeigt werden.

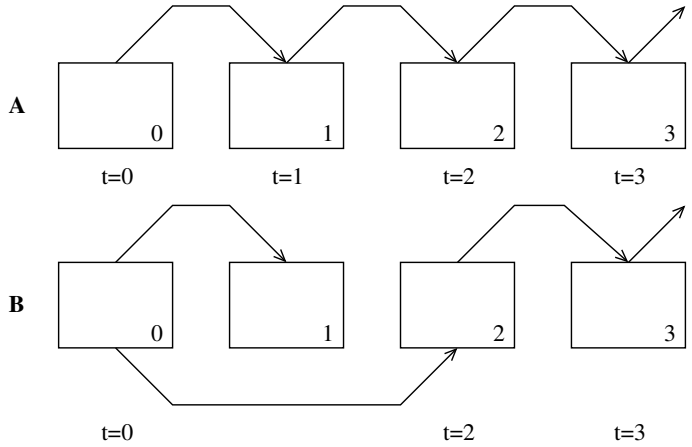

Abbildung 3.3: A ist das klassische **Increasing-ring** Verfahren, welches allerdings vom Senden von Prozess 1 abhängig ist. In Verfahren B wird A dahingehend verbessert, dass Prozess 1 jetzt nur noch empfängt und Prozess 0 zweimal sendet. Diese Methode ist laut HPL Dokumentation so gut wie immer besser, wenn nicht sogar die beste Methode aller Alternativen. Alle *modified* Versionen befreien den Prozess-Nachfolger von der Last des Sendens, sonst hätte dieser Matrix-Aktualisierungen zu empfangen und auch zu senden. Eine Beschreibung der anderen Methoden ist unter [1] zu finden

.

HPL ist in der Lage mit *lookahead* zu arbeiten, um kritische sequentielle Abläufe zu beschleunigen. Wenn zum Beispiel das k-te Panel faktorisiert wurde und schließlich sendet, ist der dringendste Folgeschritt die Faktorisierung und der Broadcast des (k+1)-ten Panels. Dieser Schritt kann beschleunigt werden, wenn die Faktorisierung sofort bei Beginn der Aktualisierung durchgeführt wird. Lookahead verbraucht mehr Speicher, da alle Panels vorgehalten werden müssen, dessen Spalten sich in der *lookahead pipe* befinden.

Da der Algorithmus in dieser Form die immer gleiche Anzahl von Schritten in Abhängigkeit von der Problemgröße (hier Zeilen oder Spalten, da die Matrix quadratisch angelegt wurde) durchläuft, kann anhand der verstrichenen Zeit eine Gleitkommaleistung festgestellt werden:

$$R = \frac{2N^3/3 + 2N^2}{t}. \tag{3.3}$$

Ist das Gleichungssystem gelöst, so berechnet HPL noch die Abweichung, indem das Ergebnis in die Matrix eingesetzt wird. Die Berechnung der skalierten Residuen beeinflusst nicht die Gleitkommaleistung und kann per Konfigurationsdatei abgeschaltet werden, um etwas Zeit zu sparen.

3.3.1 Die Parameterdatei

Der HPL wird über eine Textdatei[7] konfiguriert, welche im recht starren Aufbau an Fortran erinnert. Meist besteht eine Option aus zwei Zeilen: Die erste bestimmt die Anzahl der Durchläufe, die Zweite den jeweiligen Wert dieser Option. So wird der Benchmark dann mit allen Kombinationen mehrdeutiger Werte aufgerufen und ausgeführt. Diese Möglichkeit spiegelt den prinzipiellen Gedanken hinter LINPACK wieder: Es ist oft nicht vorhersehbar, welche Option die Leistung begünstigt oder abschwächt, daher bleibt "ausprobieren" das probate Mittel. So kann man folgern, ob reihen- oder spalteweise Anordnung der Matrix der gegebenen Speicherarchitektur entgegen kommt, vermuten, ob das empfohlene Broadcast-Verfahren tatsächlich das schnellste ist, aber nicht wirklich wissen, welches der drei zur Auswahl gestellten Faktorisierungsverfahren besser abschneiden wird.

Um eine optimale Konfiguration zu finden, mit der dann später die Leistungsbenchmarks gefahren werden, sind eine Reihe dieser semi-automatischen Testläufe notwendig (Abschnitt 4.2). Die Ergebnisse sind umso verlässlicher, je größer das Problem ist. Bei einer Kombinationsvielfalt im fünfstelligen Bereich ist der Zeitaufwand jedoch immens. In den Testläufen erwies sich die Suche aber glücklicherweise als sehr unkompliziert in Bezug auf die Beeinflussung der Werte untereinander. Es können bis zu 20 Werte pro Zeile angegeben werden, wenn die Option es zulässt. Diese müssen nicht zwangsläufig unterschiedlich sein. So ist zum Beispiel eine mehrfache Wiederholung eines Laufes mit sonst unveränderten Optionen möglich, indem einfach die Problemgröße ein paar Mal hintereinander geschrieben wird. Die Läufe können so einfach gemittelt werden. Im Folgenden soll eine kurze Erklärung der einzelnen Optionen (Abb. 3.3.1) gegeben werden. Wenn von Empfehlungen die Rede ist oder Abschätzungen gegeben werden, so bezieht sich dieses immer auf die in der HPL-Dokumentation[2][4] ausgesprochenen Hinweise.

5.-6. Zeile: Diese Zeilen legen die Problemgröße selbst fest. Letzteres ist die Anzahl der Zeilen/Spalten der erzeugten Matrix, welche faktorisiert werden soll.

7.-8. Zeile: In Zeile 8 stehen die zu verwendenden Blockgrößen. Eine gute Einstiegsbasis sind Blockgrößen im Interval [32...256]. Diese Einstellung betrifft sowohl die Granularität der Teilmatrizen, als auch die Datendistribution.

9. Zeile: In dieser Zeile wird die Abbildung der Prozesse auf die Koeffizientenmatrix festgelegt. Die zeilenweise Behandlung kommt SMP-Systemen entgegen, da die Bandbreite zwischen möglichen MPI-Prozess-Paaren nicht für alle gleich ist (siehe Abschnitt 3.3). Besteht das parallele System nur aus homogenen Knoten mit einem Prozessor, ist diese Option irrelevant.

10.-12. Zeile: In diesen Zeilen wird die Dimensionierung des Prozessgitters festgelegt. Das Produkt legt somit die Anzahl der erzeugten Prozesse fest. Dabei gilt die Festlegung Zeilen$\times$Spalten. Das Ergebnis der Multiplikation der beiden Werte muss nicht immer für alle Läufe identisch sein.

13. Zeile: Diese reelle Zahl gibt die Grenze der zu tolerierenden Abweichung beim Einsetzen des Ergebnisses in die originäre Matrix an. Während der Optimierungsphase kann die Überprüfung des Residuums durch einen negativen Wert

[7] Üblicherweise *HPL.dat*. Das Binary kennt keine Kommandozeilenoptionen mehr und so kann der Name der Datei nur vor dem Übersetzen geändert werden.

```
1     HPLinpack benchmark input file
2     Innovative Computing Laboratory, University of Tennessee
3     HPL.out      output file name (if any)
4     6            device out (6=stdout,7=stderr,file)
5     2            # of problems sizes (N)
6     78000 220000  Ns
7     1            # of NBs
8     256          NBs
9     0            PMAP process mapping (0=Row-,1=Column-major)
10    1            # of process grids (P x Q)
11    16           Ps
12    24           Qs
13    16.0         threshold
14    1            # of panel fact
15    2            PFACTs (0=left, 1=Crout, 2=Right)
16    1            # of recursive stopping criterium
17    2            NBMINs (>= 1)
18    1            # of panels in recursion
19    4            NDIVs
20    1            # of recursive panel fact.
21    2            RFACTs (0=left, 1=Crout, 2=Right)
22    1            # of broadcast
23    1            BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
24    1            # of lookahead depth
25    1            DEPTHs (>=0)
26    2            SWAP (0=bin-exch,1=long,2=mix)
27    256          swapping threshold
28    0            L1 in (0=transposed,1=no-transposed) form
29    0            U  in (0=transposed,1=no-transposed) form
30    1            Equilibration (0=no,1=yes)
31    8            memory alignment in double (> 0)
```

Abbildung 3.4: Die Konfigurationsdatei HPL.DAT für den HPL-Benchmark (hier mit Zeilennummern), wie sie beispielsweise in einem Lauf (zwei Durchgänge mit unterschiedlicher Problemgröße) verwendet werden kann.

umgangen werden. Ist der reskalierte Fehlerwert größer als der *threshold*, wird der Test als nicht bestanden gekennzeichnet, wenngleich das Ergebnis wahrscheinlich korrekt ist. Ein Lauf sollte als fehlerhaft eingestuft werden, wenn der Wert ein paar Größenordnungen über 1 liegt, beispielsweise 10^6. Die Zahl 16 sollte die meisten Fälle abdecken.

14.-21. Zeile: Die Parameter dieser Zeilen sind in Abschnitt 3.3 erklärt. Hier werden die Faktorisierungs-Algorithmen für das Panel (PFACT) und die Rekursion (RFACT) gewählt. Zu jedem dieser Schritte existieren drei numerische Verfahren. Das Panel wird solange in N_{div} Teile geteilt, bis nicht weniger als N_{Bmin} Spalten im aktuellen Teilproblem vorhanden sind. Eine klassische Rekursion wäre mit $N_{div} = 2$ und $N_{Bmin} = 1$ gegeben.

22.-23. Zeile: Zur Weiterleitung von Ergebnissen implementiert der HPL-Benchmark insgesamt sechs verschiedene Broadcast Varianten. Die Empfehlung ist hier das *modified increasing-ring* (1rM). Verfahren.

24.-25. Zeile: In Zeile 25 kann die Tiefe des *lookahead* eingestellt werden. Ein Wert von 0 bedeutet, dass kein *lookahead* benutzt wird. Bei einem größeren Wert als 1 ist es eher unwahrscheinlich, dass er die Leistung erhöht.

26.-27. Zeile: In diesen Zeilen wird der Algorithmus zur Vertauschung von Zeilen/-Spalten festgelegt. Zur Auswahl stehen zwei Methoden: Die *long* Variante sollte bei großen Problemen die erste leistungsmäßig übertreffen. Es ist aber möglich, beide im *mix* (Option 2) zu benutzen. Dieses ist theoretisch die beste Wahl. Die Größe des *threshold* sollte der der Blockgröße entsprechen.

28.-31. Zeile: In den letzten Zeilen der Datei finden sich ein paar Details, die unter Umständen noch für das eine oder andere Prozent mehr Leistung gut sind, wenn alle anderen Einstellmöglichkeiten ausgereizt sind. Die Standardwerte sollten hier für gewöhnlich ausreichen.

Die Ausgabe eines HPL-Laufes ist pro Zeile eindeutig bestimmt. In ihr finden sich alle Angaben zur Konfiguration wieder. Die erste Spalte ist dabei nach fixen Stellen kodiert (Abb. 3.5). Der folgende Lauf erreichte auf 16 ($P \times Q$) Kernen mit einer Blockgröße NB von 256 und einer Problemdimension N von 62000 etwa 83 GFlop/s bei einer Laufzeit von etwa 31 Minuten:

```
================================================================================
T/V                N      NB     P     Q       Time          Gflops
--------------------------------------------------------------------------------
WR11R4R2       62000     256     4     4    1913.78       8.302e+01
================================================================================
```

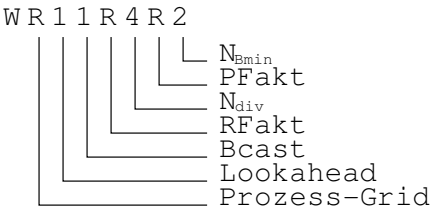

Abbildung 3.5: Die Bedeutung des Ausgabe-Codes von HPL. Das *W* steht nicht für eine Konfigurationsoption. Hier wird das Prozess-Gitter reihenweise aufgeteilt, eine *lookahead*-Tiefe von 1 benutzt, jeweils der *right-looking*-Algorithmus für die Faktorisierung gewählt und das Panel pro Schritt in 4 Teile geteilt, bis nicht weniger als 2 Spalten übrig sind.

II Praxis

4 LINPACK auf dem Cluster

4.1 Testumgebung

4.1.1 Hardware

Die Benchmarks werden auf dem Rechencluster des D-Grid-Integrationsprojekt am RRZN/L3S durchgeführt. Dieser besteht aus insgesamt 60 Knoten, welche zu beliebigen Teilmengen zu Queues zusammengefasst werden. Für gewöhnlich ist der Knoten mit der Nummer 60 nur als Login-Rechner aktiv. Während der Arbeit wurden von hier aus die Jobs in die Queues geschickt, es standen also effektiv zwischenzeitlich 59 Rechenknoten für den LINPACK-Benchmark zur Verfügung.

Jeder Knoten beherbergt zwei Sockel mit je einer Quad-Core Einheit von Intel des Typs Xeon-MP (Core) X5355 (Modell 6, Stepping 7, "Clovertown") im Intel-64 Modus (EM64T). Eine einzelne Core hat eine Taktfrequenz von 2.667 GHz und kann vier Gleitkommaoperationen pro Takt durchführen. Der Speicherbeträgt je Knoten 16 GByte. Es stehen also insgesamt 480 Cores und 960 GByte Speicher zur Verfügung. Ohne Login-Knoten sind es noch 59 SMP-Systeme mit 472 Cores und 944 GByte Arbeitsspeicher. Die CPUs verfügen über die 128 Bit breiten Befehlssatzerweiterungen *SSE/2* für Gleitkommaoperationen[1].

Die theoretische Spitzenleistung einer CPU ist 42.672 GFlop/s. Damit erreichen 59 Knoten 5.035 TFlop/s. 58 Knoten kämen noch auf einen **Peak von 4.95 TFlop/s**, welches die angepeilte Marke sein soll, da sich für die Zahl 472 (Cores in 59 Knoten) mit 8×59 nur ein relativ flaches Prozessgitter findet.

Im für den Benchmark bestimmten Cluster stehen zwei Interconnects in Stern-Topologie zur Verfügung: TCP/Ethernet mit einem GBit/s und 4-Kanal Infiniband mit 20 GBit/s.

4.1.2 Software

Das Betriebsystem ist *Scientific Linux CERN SLC release 4.7 (Beryllium)* mit Distributionskernel Version 2.6.9-78.0.1.EL.cernsmp.

An weiterer Cluster-Software sind der *Maui Cluster Scheduler 3.2.6p19* und der *PBS/TORQUE Resource Manager 2.1.6* installiert. *OpenMPI* ist in Version 1.2.4 vorhanden und jeweils für den *GNU C Compiler 4.1.2* und *Portland Group C Compiler 7.1-4* verfügbar. Alle Komponenten exklusive des PGCC sind Open Source Software.

[1] AMD erhielt seinerzeit die Lizenz für die Nutzung der Erweiterung *SSE2* von Intel im Tausch gegen die Lizenz für die 64-Bit Erweiterung *AMD64* (*EM64T*).

4.2 Ermitteln optimaler Parametern

Die Läufe wurden jeweils mit einer Problemgröße von 16000 Zeilen durchgeführt. Es ergaben sich teilweise bis zu 20MByte große Textdateien mit Output, dessen relevante Zeilen nach Leistung sortiert werden mussten. Um die Anzahl der Permutationen in Grenzen zu halten, wurden teilweise Tuning-Läufe durchgeführt, die nur einzelne Optionen variierten. Hier sollten zuerst die sogenannten Daumenregeln oder Empfehlungen der HPL Dokumentation überprüft werden. Es wurde der Übersichtlichkeit wegen auf die Ausgabe der einzelnen Testläufe an dieser Stelle verzichtet und nur die Erkenntnisse mit Begründung aufgelistet:

Der Broadcast wurde auf zwei Knoten mit jeweils einem Prozess ermittelt, um Shared-Memory Effekte zu vermeiden. Der *modified increasing-ring* konnte sich zusammen mit der unmodifizierten Variante bei beiden Netzwerktypen gegenüber den vier anderen Varianten klar durchsetzen. Die Wahl fiel hier auf den etwas besseren und in der Dokumentation bevorzugten 1rM-Broadcast.

Das Prozessgitter ist in symmetrischen Aufteilung und bei reihenweiser Abarbeitung in den Tests am effektivsten. Gefolgt vom spaltenweisen Prozess-Mapping (Parameter PMAP). Die flachen Gitter fielen deutlich zurück. Die im praktischen Test ermittelten Beobachtungen stimmen mit der theoretischen Vorbetrachtung (3.3) überein. In den Läufen wird also ein maximales P mit $P < Q$ und $P \cdot Q$ gleich Anzahl verfügbarer Cores gesetzt.

Bei den Faktorisierungsmethoden konnte in ausführlichen Tests kein reproduzierbares Ergebnis ermittelt werden, welches den eindeutigen Sieger festlegte. Es wurde daher das häufigste Auftreten in den hundert schnellsten Kombinationen bestimmt. Gewinner dieser Auszählung ist bei der Panel- sowie Rekursiv-Faktorisierung der *right-looking*-Algorithmus.

Die Rekursionsschritte haben sich ebenfalls nicht besonders sichtbar auf das Ergebnis ausgewirkt. Die Kombination, jedes Panel zu vierteln, bis mindestens noch zwei Spalten im aktuellen Panel vorhanden sind, stand sehr knapp an der Spitze und wird deshalb verwendet.

Die Blockgröße wurde im Interval [1...1024] getestet. Die höchste Leistung wurde mit dem Wert 256 erzielt. Ein Test mit einem Vielfachen - wie in der HPL Dokumentation angeregt - konnte kein besseres Ergebnis hervorbringen.

Der Lookahead wurde eingeschaltet und erreichte bei 1 bessere Ergebnisse als ausgeschaltet. Ab 2 kehrte sich die Beobachtung ins Gegenteil um.

4.3 Wahl des Setups

Bei einigen Tests wurde sich ein deutlicheres Ergebnis erhofft, aber keines gefunden. Die getroffene Wahl entbehrt also teilweise einer Rechfertigung, bis auf die Tatsache, dass sie empirisch nicht schlechter ist, als eine beliebige andere. Letztendlich fand sich folgende Einstellung: **WR11R4R2** (siehe Abb. 3.5) mit $N_B = 256$.

4.3.1 PBS/Torque

Der Header jeder Batch Datei, mit der ein Lauf an die Queue übergeben wurde, hatte jeweils folgende Charakteristika:

```
#PBS -q dgipar -j oe -o run.out
#PBS -l nodes=4:ppn=2
#PBS -l pmem=8000MB
#PBS -l walltime=12:00:00
```

Die Queue für parallele Verarbeitung hat den Namen *dgipar*. Sämtliche Ausgaben sollen in eine Datei geschrieben werden (Parameter -j oe -o run.out). Die zweite Zeile legt die Anzahl der verwendeten Knoten und deren zu benutzenden Core-Anzahl fest. Leider unterscheidet der Scheduler hier nicht wirklich nach nodes, so dass obige Zeile ohne weitere Maßnahmen wohl auf einem Knoten ausgeführt worden wäre. Erst durch Zeile drei, die jedem dieser Prozesse einen Maximalwert an Speicher garantiert, werden nur noch zwei Prozesse pro Knoten gestartet, da sie wegen ppn $\cdot$ pmem die verfügbaren 16 GB allozieren. Die walltime zu setzen ist notwendig, weil die Testläufe teilweise über das Default-Zeitlimit des Queuemanagers hinaus aktiv waren.

5 Benchmarking

The following performance data should not be taken too seriously.

Hinweis im Benchmark-Report der Top 500

5.1 Ergebnisse und Beurteilung

Nach den ersten Testläufen stellte sich ziemlich schnell heraus, dass sich mit der auf dem System vorgefunden Linear-Algebra-Bibliothek nur sehr schlechte Leistung erzielen lies. Der HPL-Benchmark lieferte ein Single-Core Ergebnis von 476,9 MFlop/s, das entspricht knapp einem Prozent des theoretischen Peaks. Daher wurde für weitere Läufe auf Vergleiche mit dieser BLAS-Variante verzichtet und stattdessen gegen ATLAS und Intel MKL getestet. Da diese den kritischen Anteil an der Leistung des Benchmarks stellen, musste ATLAS jeweils einmal mit dem Gnu Compiler und dem Portland Group Compiler übersetzt werden. Die Kombination HPL und ATLAS mit Gnu Compiler wird im Folgenden kurz B gcc-atlas, das PG Compiler Pendant kurz C pgcc-atlas genannt. Weiterhin konnte kein Unterschied bei der Verwendung mit der IMKL in Bezug auf die Übersetzung des HPL gemacht werden, so dass hier schlicht die GNU Version benutzt wurde (kurz A gcc-imkl). Bei der Lösung eines linearen Gleichungssystems gelingt es nicht, das Problem in mehrere unabhängige Teilprobleme zu zerlegen. Die einzelnen Instanzen müssen miteinander kommunizieren. Wie sehen nun die Effekte in der Praxis aus, die in den vorhergehenden Kapiteln diskutiert wurden? Um die Auswirkung zusätzlicher Rechenknoten und des Verbindungsnetzwerkes (insbesondere auch TCP/Ethernet gegenüber Infiniband) zu betrachten, wurde der Benchmark für verschiedene Prozessgitter und Problemgrößen auf dem Cluster durchgeführt. In Abbildung 5.1 sind die verschiendenen Gittergrößen von einem einzelnen Prozess (Single Core) bis hin zu 48 Knoten (384 Cores) mit der erreichten Gleitkommaleistung für die drei Kombinationen A, B und C graphisch dargestellt. Die maximale Dimension, die einem einzelnen Knoten zugemutet wurde, betrug 38.000. Ab einem Gitter von 4×4 - das entspricht zwei Knoten - kam außer neuen CPUs auch mehr Speicher hinzu, welcher verteilt für das gemeinsame Problem benutzt werden konnte. Man kann erkennen, dass die Größe der gelösten Probleme monoton mit der Anzahl der verfügbaren Knoten anwächst. Die Probleme sind ab einer gewissen Größe auch immer schneller vom System lösbar, wenn mehr Knoten vorhanden waren. Es ist auffällig, dass große Gitter mitunter bei kleinen Problemgrößen sehr schlecht abschneiden. Erst ab ca. 4000 - 20.000 Zeilen sind mehr Knoten auch immer schneller. Das sind die praktischen Auswirkungen der Granularität (siehe Abschnitt 1.3), bei welcher das Verhältnis von Rechenleistung zu Kommunikationskosten deutlich wird. Ein kleines

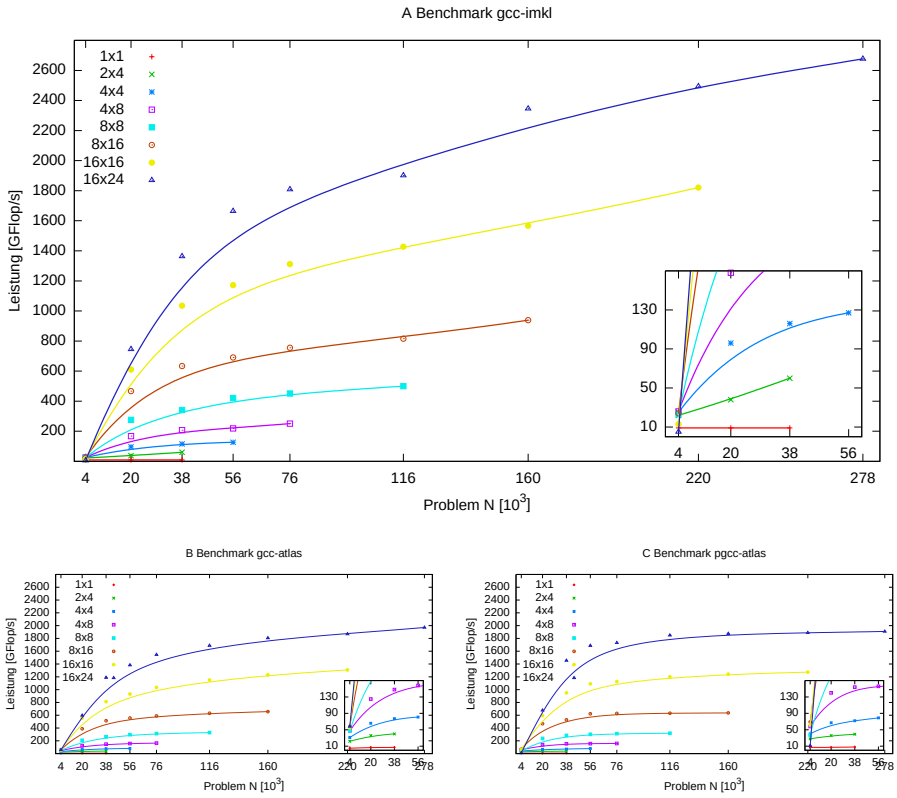

Abbildung 5.1: Die ermittelten Gleitkommaleistungen der Kombinationen als Funktionen der Problemgröße mit unterschiedlichen Prozessgittern

Problem verursacht anteilsmäßig einen zu hohen Overhead im Bezug zur reinen Nutztätigkeit der CPUs, wenn es verteilt werden muss. Wie man der Graphik entnehmen kann, sind nicht alle Prozessgitter im gleichen Maße günstig für eine bestimmte Problemgröße. Am Ende der Kurve ist, am deutlichsten zum Beispiel bei Graph A (Abb. 5.1 oben bei 16×24,116.000), ein gewisses “einpendeln” von gcc-imkl zu sehen. Die abflacherende Kurve in Graph C (Abb. 5.1 unten rechts) lässt vermuten, dass die Kombination pgcc-atlas ihr Maximum an Leistung schneller mit weniger Knoten erreicht, als die beiden anderen. Der skalierte Speedup wächst kaum noch (Tab. 5.1). Das Plus an CPU Leistung durch mehr Knoten scheint keine positiven Auswirkungen mehr hervorzubringen. So kann Kombination pgcc-atlas in dieser Konfiguration lediglich den Status Quo aufrecht erhalten. Sehr wahrscheinlich muss hier die Problemgröße stärker wachsen, als es der Zugewinn an Speicher durch mehr Knoten (derzeit 16 GB) zulässt. Bei Graph A ist dagegen erkennbar, dass eine steilere Kurve am Ende auf ein noch vorhandenes Potential bei gcc-imkl hindeutet. Nach Gustafson’s Gesetz (Abschnitt 1.3) wäre in den drei Fällen ein skalierter Speedup von 384 möglich, wenn N nur hinreichend groß würde. Die Kurve erreicht bei 58 Knoten immerhin einen Wert von 318. Das Maximum ergäbe, legt man vereinfachend die Steigung der letzten beiden Punkte zugrunde, in etwa eine Matrixdimension von 365.000, also einen Speicherbedarf von knapp 992 GB ($\widehat{=}$17.1 GB/Knoten für Daten; Mit der 90% Regel müssten etwa

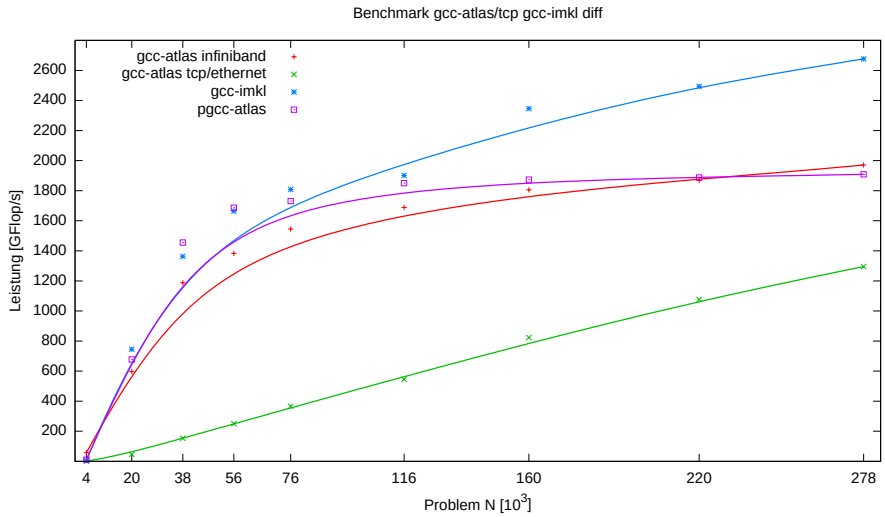

Abbildung 5.2: Vergleich des 16×24 Prozessgitters mit den verschiedenen algebraischen Bibliotheken.

3 GB/Knoten nachgerüstet werden). Diese Rechnung ist grob überschlagen und es ist natürlich anzunehmen, dass der Leistungsgewinn nicht verlustfrei sein wird. Insgesamt betrachtet, hat Kombination gcc-imkl deutlich besser abgeschnitten, als gcc-atlas oder pgcc-atlas. Optimierungen vom Hardware-Hersteller sind in diesem Fall effizienter, als eine selbstoptimierende Version der Linearen Algebra Bibliothek.

Der Einfluss der Verluste durch ungleiche Lastverteilung oder Kommunikationsaufwand ist schon mehrfach erwähnt worden. Jedoch ist durch die Isoeffizienz-Formel bekannt, dass eine effektivere Auslastung bzgl. der Rechenzeit kompensierend wirkt. In der Tat fällt die Effizienz bei konstanter Problemgröße mit dem Anwachsen der Prozesse signifikant ab (Abb. 5.3 oben). Es steht zwar mehr Rechenleistung zur Verfügung, jedoch müssen die Knoten die Arbeit über das relativ langsame Verbindungsnetzwerk miteinander koordinieren. Wenn die Problemgröße in ausreichendem Maße mitwächst, bleibt die Effizienz konstant (Abb. 5.3). Bemerkenswert ist, dass pgcc-atlas auch bei konstantem N eine relativ gute Effizienz halten kann. Der gcc-imkl hingegen verhält sich analog zu den vorherigen Beobachtungen insofern, als dass der Nutzanzeil der Prozessoren leicht abfällt, da die Problemgröße nicht ausreichend mitwachsen kann. Das

Problemgröße	t_{seq}' 1×1 [s]	t_{par} 16×24 [s]	Speedup
38000	4519	25	180
160000	337327	1456	231
220000	876919	3759	233
278000	1769399	7503	235

Tabelle 5.1: Der skalierte Speedup für 48 Knoten bei Kombination pgcc-atlas. Die Werte für die sequentielle Laufzeit sind hypothetisch, da sie nicht mehr auf dem System ausführbar wären. Für die Berechnung wurde ein vereinfachtes kubisches Verhältnis von Zeitbedarf zu Matrixdimension angenommen.

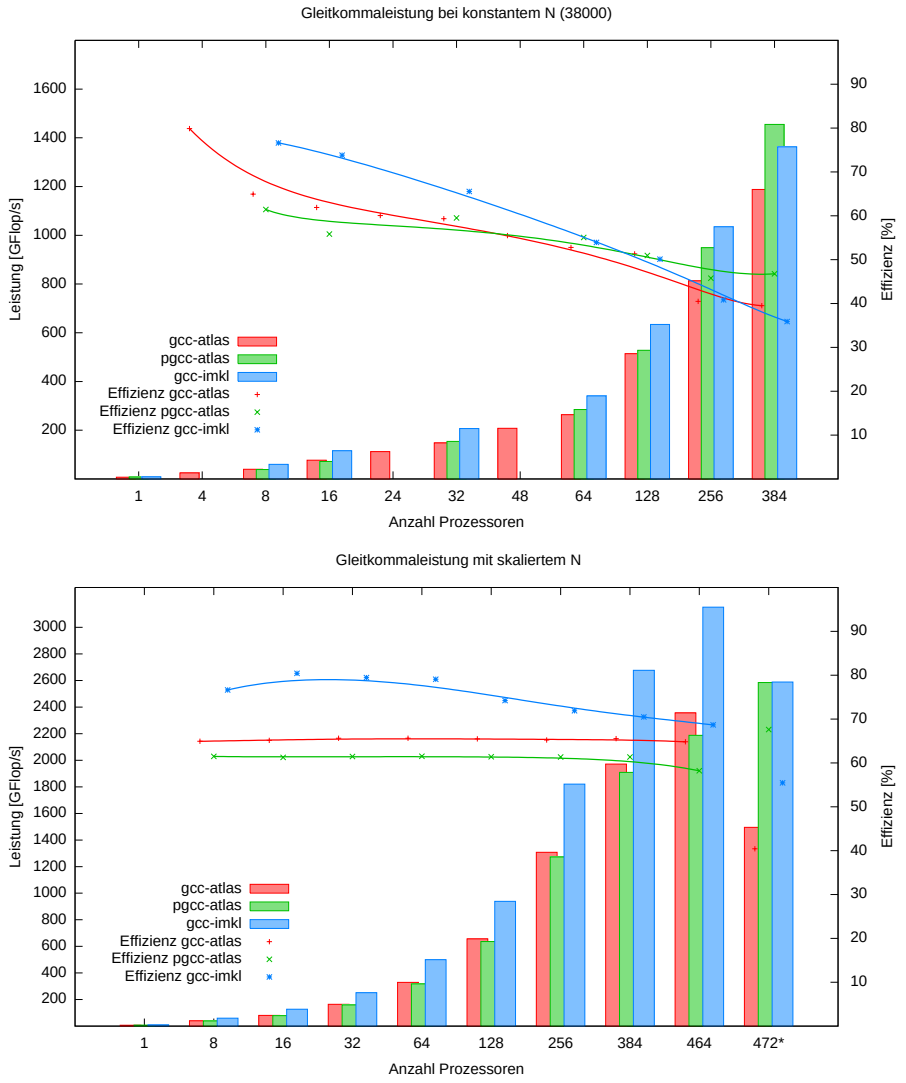

Abbildung 5.3: Die Leistung der Kombinationen im Vergleich zur Effizienz bei konstanter (oben) und skalierter Problemgröße (unten). In der skalierten Version tauchen noch zwei zusätzliche Konfigurationen auf: 464 und 472 Prozessoren. der letzte Wert ist nicht bei der Effizienz berücksichtigt worden, da er durch seine Prozessgitteraufteilung nicht die optimale Voraussetzungen erfüllt.

Verhalten des gcc-imkl ist leicht einzusehen, wenn man eine effektivere Verarbeitung der Teilprobleme auf den einzelnen Knoten berücksichtigt, welche die reine Nutzzeit der Berechnungen senkt und dadurch das Verhältnis zu den Verlusten entsprechend ungünstig beeinflusst. Beim Speedup (Abb. 5.4) zeigt der gcc-imkl seine Abhängigkeit von der Problemgröße ebenso deutlich. Er skaliert bei konstanter Größe sogar am schlechtesten. Die Beschleunigung bei wachsendem N hingegen ist die stärkste. Trotzdem kann in diesem Fall die Isoeffizienz nicht erreicht werden und das bedeutet, dass

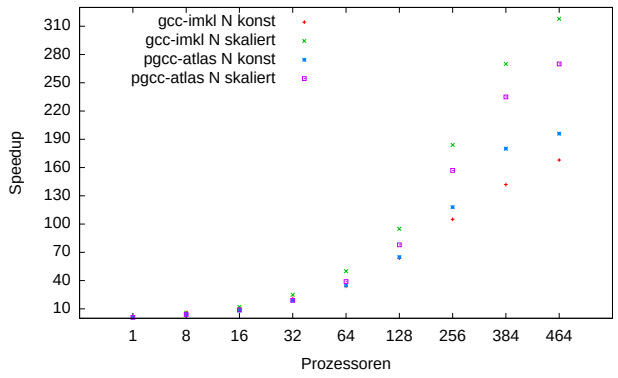

Abbildung 5.4: Die einfachen und skalierten Speedups von gcc-imkl und pgcc-atlas. Bei der äquidistanten Darstellung ist zu berücksichtigen, dass sich ab 256 Knoten die Anzahl nicht weiter verdoppelt, sondern nur um 128 und nochmal 80 Cores ansteigt. Das "abflachen" des Speedups ist also mitunter graphisch bedingt.

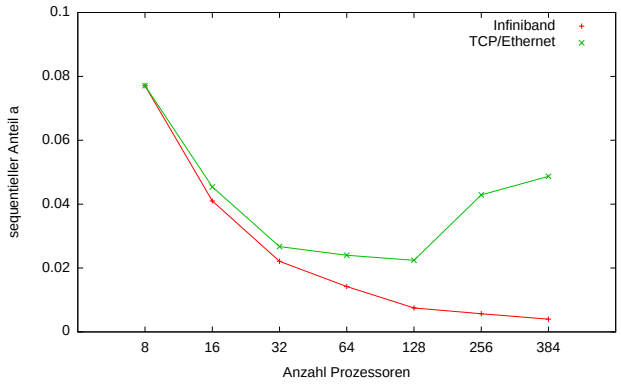

Abbildung 5.5: Die Karp-Flatt-Metrik für Infiniband und TCP/Ethernet. Die Umstellung von Amdahls Gesetz liefert empirisch einen Hinweis darauf, ob Kommunikation oder Lastverteilung die Verluste dominieren. Mit der Berücksichtigung des Overheads liegt diese Formel quasi zwischen Amdahl und Gustafsons Aussage.

in Fall gcc-imkl das Cluster nicht optimal skaliert.

Für den Lauf mit größtmöglichem Problem standen 59 Knoten zur Verfügung. Da sich in jedem Gehäuse acht Cores befinden, ist so nur ein ungünstiges, da relativ flaches, Prozessgitter von 8×59 möglich. In Abbildung 5.3 (unten) ist der Leistungsabfall gegenüber der 464 Core Variante mit einem Knoten weniger deutlich zu erkennen. Einzig die Kombination pgcc-atlas scheint aus den zusätzlichen Ressourcen (acht CPUs und 16 GB Speicher) sogar noch einen Vorteil herauszuholen. Dieser Fall illustriert sehr schön, dass Rechenleistung nicht gleich Rechenleistung sein muss. Erst die Erweiterung um vier zusätzliche Rechner unter Einbeziehung des Login-Knotens lässt wieder ein besseres Prozessgitter von 12×32 zu (die Problemgröße stiege dabei auf 351488). Erzwingt man für die MPI-Prozesse die Kommunikation über das TCP/Ethernet Netzwerk, so sind deutliche Leistungseinbußen zu verzeichnen (Abb. 5.2). Das liegt einerseits am geringeren Durchsatz von nur einem GBit/s, welcher bei anwachsender Zahl von Knoten an seine Grenzen stieß[1]. Aber vielmehr am vergleichsweise hohen Anteil von Verzögerung pro Verbindung, die sich aufsummierten. Während das latenzarme Infiniband anscheinend besser in der Lage ist, die hohe Anzahl von Prozess-Kommunikation zu bewältigen und einen sehr steilen Anstieg der Kurven bei kleinen Problemgrößen zur Folge hat, bewegt sich TCP/Ethernet beinahe linear. Das Anwachsen des empirisch-sequentiellen Anteils in der Karp-Flatt-Metrik (Abb. 5.5) bestätigt,

[1] Auf dem Backplane des Switches war zwischenzeitlich das fünffache zu messen.

dass die Kommunikation und nicht eine schlechte Lastverteilung die parallelen Reibungsverluste dominiert. Der sequentielle Anteil sollte, wie bei Infiniband zu sehen, bei höherem Grad an Parallelisierung immer weniger ins Gewicht fallen. Die Prozessoren verbringen sonst einen zunehmenden Anteil ihrer Zeit mit dem Warten auf neue Daten. Die Ansprüche an die Zulieferung kann das Netzwerk somit nicht erfüllen.

Alle zugrundeliegenden Ergebnisse für Kapitel 5 sind auf der beigelegten DVD im Verzeichnis `Ergebnisse/Benchmarks` einzusehen.

5.2 Ranking in der Top 500

Aufgrund nonoptimaler Bedingungen für das 59 Knoten Cluster, konnte die maximale Gleitkommaleistung nur auf 58 Knoten gemessen werden. Das endgültige Ergebnis lautet wie folgt:

Procs	Proc. Frequency	Proc Family	R_{max}	R_{peak}	N_{max}	$N_{1/2}$
464	2666	EM64T	3152	4949	310016	56000

Die letzte Aufrüstung des Clusters fand im Dezember 2007 statt. Bei Erscheinen der Liste im November 2007 lag aber bereits die Einstiegsgrenze bei 5.929 TFlop/s (Tabelle 2.3). Tabelle 5.2 zeigt eine ungefähre Prognose des Rankings. Im Juni 2007 findet sich ein etwa vergleichbares Infiniband System von SGI, allerdings mit 544 Cores des Typs Xeon 53xx (Clovertown) mit je 2666 MHz. Es erreicht bei einer Problemgröße von 250880 einen Wert von 4.468 TFlop/s. Die Verteilung in der Top 500 Liste liegt um

Halbjahr	Bestmöglicher Rang
Nov 2006	374
Jun 2007	534
Nov 2007	**777**
Jun 2008	1075

Tabelle 5.2: Ein Ranking für den Zeitraum nach Juni 2007 ist nicht mehr möglich, da die Listen jeweils bei Platz 500 enden und die Mindestanforderungen bereits zu hoch waren. So kann nur eine grobe Einschätzung anhand der Verschiebung nach unten durch neu aufgenommene System gegeben werden. Dann käme das getestete Cluster immerhin auf Platz 777 im relevanten Zeitraum.

einen Wert von 1024-2048 Cores und ist anteilsmäßig an stärksten mit EM64T Architektur besetzt. Ein Eintrag in die Liste ist unter diesen Umständen also nur mit deutlich höheren Kosten zu erbringen.

III Fazit

6 Verlauf des praktischen Teils

6.1 Probleme

Der zeitintensivste Teil waren sicherlich die empirischen Leistungstests, um eine optimale Konfiguration zu finden. Mehr Optionen lassen ein besseres Feintuning zu, erhöhen aber auch den Zeitaufwand dafür immens. Denn Tests mit kleinen Problemgrößen sorgen für große Schwankungen in den Resultaten, da sich durch die kurzen Ausführungszeiten externe Einflüsse, wie zum Beispiel des Betriebssystems, deutlicher auswirken. Das macht eine Festlegung auf diese Weise nicht möglich. Viel Zeit wurde also im Laufe der Arbeit damit verbracht, Konfigurationen zu ändern, Queues zu füllen und Endergebnisse zu vergleichen. Es ist allerdings schwierig, diesen Aufwand sinnvoll in dieser Arbeit aufzuführen.

Die Durchführung der Benchmarks gestaltete sich nach ein paar Anlaufschwierigkeiten ohne größere Schwierigkeiten. Teilweise wurden jedoch Knoten mit Arbeit überlastet, was nicht immer nachvollziehbar war. Grund war oft eine zu knapp bemessene Marge für das Betriebssystem. Problemgrößen, die auf einigen Knoten bearbeitet werden konnten, sorgten auf anderen wiederum für so starkes Swapping, dass der Knoten nicht mehr ansprechbar war. Die Problemgröße sollte in jedem Fall größtmöglich sein, um die Ausbeute zu maximieren. Dieses *trial and error* Verfahren führte im Laufe der Arbeit zu eher vorsichtigen Einstellungen, um nicht weitere Abstürze zu provozieren.

Ein weiteres Hindernis stellte die aktive Produktionsumgebung des Clusters dar. Die Queue konnte nicht beliebig erweitert werden, um andere Nutzer nicht zu beeinträchtigen. Glücklicherweise wurde es mir ermöglicht, die Ressourcen immer weiter zu beanspruchen und am Ende sogar kurzfristig alle 59 Knoten einzusetzen.

Das Zeitfenster für den Test mit der größten Matrix war eng gesteckt und leider konnten, bedingt durch Hardwareausfälle, nicht alle Tests mehrfach oder in gewünschter Konfiguration ausgeführt werden. So war es in der Zeit nicht möglich, ausreichend sicherzustellen, ob nicht doch noch ein paar Prozent mehr Leistung möglich gewesen wären. Der Kurvenverlauf der `gcc-imkl` Variante lässt Raum für Spekulationen, in denen Konfigurationsveränderungen sich unter Umständen anders auswirken, als auf *gcc-atlas*. Die Festlegung der Parameter wurde aussschließlich mit `atlas` durchgeführt. Die Intel Version kam erst im Verlauf der Arbeit hinzu, als sie durch tiefere Beschäftigung mit der Materie eine interessante Alternative wurde. Ziel der Arbeit sollte ursprünglich nicht sein, sämtliche BLAS-Implementationen auf ihre Leistung zu vergleichen. Aber für die Ermittlung des höchsten Wertes erschien die Verwendung der `imkl` sehr sinnvoll. Es wurde von mir angenommen, dass die parallele BLAS Version schlechter abschneiden würde, als mehrere MPI-Prozesse. Denn im ersten Fall müsste durch das

Threading die Zahl der MPI-Prozesse reduziert werden, da sonst ein Ressourcenmangel, also Overhead durch Paging, enstehen würde.

Die Queue für sequentielle Jobs (zum Beispiel *boinc*) gab immer wieder Anlass für Zeitverzögerungen. Die Arbeitsverteilung schien sich nicht an Gegebenheiten der parallelen Queue zu orientieren und so konnte nie vorhergesagt werden, ob und wann mehrere dieser Jobs auf einen Knoten delegiert wurden, der bereits im Benchmarking aktiv war. Das stellt unter normalen Bedingungen kein Problem dar, ist aber im speziellen Fall einer Leistungsmessung natürlich ein klares Abbruchkriterium.

6.2 Einschätzung

Nachdem die Theorie recht gut zu den praktischen Ergebnissen passte, lässt sich eine vorsichtige Abschätzung über eine mögliche Aufrüstung geben.

Der kleinste vorhandene Hauptspeicher in einem Knoten ist für diese Anwendung verteilten Speichers das schwächste Glied. Es macht also keinen Sinn (abgesehen von sequentiellen Jobs) Knoten mit mehr als 16 GB Hauptspeicher dem Cluster hinzuzufügen. Nähme man nun etwa 40.000 Euro in die Hand, so entspräche das wahlweise 16 GB zusätzlichem Speicher für die bisherigen 60, oder 15 neue Knoten in gewohnter Ausstattung. Das maximale Problem wäre in der ersten Variante deutlich größer (507T zu 380T). Variante zwei wird jedoch von den 120 zusätzlichen Kernen profitieren.

Aus den bisherigen Betrachtungen kann man für das Szenario die folgenden Schlüsse ziehen: Wenn die Probleme gleich bleiben, wird das Plus an Prozessoren für eine schnellere Bearbeitung und damit kürzere Wartezeiten sorgen, auch wenn die Effizienz dabei erheblich sinken würde. Tendenziell liegt aber die Befürchtung nahe, dass die Gesamtleistung wieder durch eine *zu* geringe Problemgröße ausgebremst wird.

Es ist wichtig, sich das Ziel zu stecken, das man hier erreichen möchte. Wenn das Cluster weiterhin haupsächlich durch sequentielle Jobs mit hoher Laufzeit beansprucht wird, sind einzelne Kerne zweifelsfrei die bessere Wahl. Dann reichen auch die 16 GB pro Knoten aus. Sollen zukünftig andere Probleme, wie Wetterdaten etc. verarbeitet werden, sollte der Gesamtspeicher in einem gewissen Maße mit der Anzahl der Knoten mitwachsen, sonst ist die Anschaffung unwirtschaftlich. Sind größere Probleme vorgesehen, so kommt man um zusätzlichen Arbeitsspeicher sowieso nicht herum.

Literaturverzeichnis

[DLP03] J.J. DONGARRA, P. LUSZCZEK, A. PETITET. *The LINPACK Benchmark: Past, Present and Future. Concurrency and Computation.* (2003)

[DON08] J.J. DONGARRA. *Performance of Various Computers Using Standard Linear Equations Software.* (2008)

[DBMS79] J.J. DONGARRA, J.R. BUNCH, C.B. MOLER, AND G.W. STEWART. *LINPACK User's Guide.* (1979)

[WPD00] R.C. WHALEY, A. PETITET, J.J. DONGARRA. *Automated Empirical Optimizations of Software and the ATLAS project.* (2000)

[BM06] H. BAUKE, S. MERTENS. *Cluster Computing.* (2006)

[HK09] H.G. KRUSE. *Leistungsbewertung bei Computersystemen.* (2009)

[GA67] G.M. AMDAHL. *The validity of the single processor approach to achieving large scale computing capabilities.* (1967)

[KF90] A.H. KARP, H.P. FLATT. *Measuring Parallel Processor Performance.* (1990)

[JG88] J.L. GUSTAFSON. *Reevaluating Amdahl's Law.* (1988)

[1] *HPL Algorithm*
http://www.netlib.org/benchmark/hpl/algorithm.html

[2] *HPL Tuning*
http://www.netlib.org/benchmark/hpl/tuning.html

[3] *HPL Scalability Analysis*
http://www.netlib.org/benchmark/hpl/scalability.html

[4] *HPL Frequently Asked Questions*
http://www.netlib.org/benchmark/hpl/faqs.html

[5] http://de.wikipedia.org/wiki/Benchmark

[6] http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html

[7] http://www.netlib.org/linpack/

[8] http://www.netlib.org/benchmark/hpl/

[9] http://www.intel.com/cd/software/products/asmo-na/eng/219771.htm

[10] http://sourceforge.net/project/showfiles.php?group_id=23725

[11] http://www.csm.ornl.gov/pvm/

[12] http://www-unix.mcs.anl.gov/mpi/

[13] http://www.spec.org/

[14] http://www.beowulf.org/

A Anhang

A.1 Installationsanleitung

Im Folgenden wird die recht einfache Installation des HPL Benchmarks kurz beschrieben. Die Ähnlichkeiten zu Version mit `pgcc` sind marginal und werden daher hier nicht noch einmal extra aufgeführt. Der Quelltext für Version 1.0a und 2.0 kann unter [8] runtergeladen werden. Für die Arbeit wurde Letztere benutzt. Die Konfiguration für HPL wird manuell in einem Makefile durchgeführt. Dazu kann nach dem Auspacken des Archivs eine der mitgelieferten Dateien im Verzeichnis *setup* ins Hauptverzeichnis des HPL kopiert und entsprechend inhaltlich angepasst oder umbenannt werden. Da hier die Pfade zu den verwendeten BLAS eingetragen wird, müssen diese zuvor installiert werden.

HPL mit GCC und IMKL

Eine *non-commercial* Version für Linux kann unter [9] heruntergeladen werden. Es wird entweder eine Lizenzdatei oder ein Lizenzcode benötigt, der nach Einverständniserklärung und Angabe einer E-Mail-Adresse zugeschickt wird. Nach dem Auspacken muss lediglich das Shell-Skript `install.sh` ausgeführt und den Anweisungen gefolgt werden. Hier wird unter anderem auch der Installationspfad angegeben. Da sämtliche Dateien schon vorkompiliert sind, besteht die Installation hauptsächlich aus Kopierarbeiten.

HPL mit GCC und ATLAS

Der Quelltext kann unter [10] heruntergeladen werden. Nach dem Auspacken ist manuell das Installationsverzeichnis anzulegen und von dort aus das `configure` des entpackten Archivs aufzurufen. Mit dem üblichen `-prefix=...` kann der Zielpfad angepasst werden. Anschließend wird durch `make; make install` der Übersetzungsvorgang und die Installation angestoßen.

Sind diese Sachen erfolgreich erledigt worden, kann nun der HPL selbst übersetzt werden. Nach dem Kopieren und Umbenennen eines Makefiles (je eines für `gcc-imkl` und `gcc-atlas`) sind darin verschiedene Pfade anzugeben, die im Folgenden aufgeführt werden. `Pgcc` Anweisungen sind jeweils kommentiert.

```
# ----------------------------------------------------------------
# - Platform identifier ------------------------------------------
# ----------------------------------------------------------------
ARCH         = Linux.imkl
#pgcc# ARCH  = Linux.pgcc.atlas
```

Diese Option bekommt einen selbstgewählten Namen. Sinnvoll ist hier der letzte Teil des Makefile. Mit diesem Namen werden diverse Unterverzeichnisse angelegt und die kompilierten Dateien erzeugt. Es lassen sich also beliebig viele Versionen eines HPL-Binary erzeugen, wenn unterschiedliche Namen gewählt werden. In diesem Fall soll der HPL gegen die Imkl gelinkt werden.

```
# ----------------------------------------------------------------
# - HPL Directory Structure / HPL library ------------------------
# ----------------------------------------------------------------
TOPdir       = /home/uhed0035/software/hpl-2.0
INCdir       = $(TOPdir)/include
BINdir       = $(TOPdir)/bin/$(ARCH)
LIBdir       = $(TOPdir)/lib/$(ARCH)
HPLlib       = $(LIBdir)/libhpl.a
```

Hier ist unbedingt das `TOPdir` anzupassen, welches für gewöhnlich den Ort des entpackten HPL Quellcodes darstellt. Sonst wird der Übersetzungsvorgang fehlschlagen.

```
# ----------------------------------------------------------------
# - Message Passing library (MPI) --------------------------------
# ----------------------------------------------------------------
MPdir        = /home/software/mpi/gcc4/openmpi-1.2.4
#pgcc# MPdir = /home/software/mpi/pgcc/openmpi-1.2.4
MPinc        = -I$(MPdir)/include
mPlib        = $(MPdir)/lib64/libmpi.so
```

Unterschiede gibt es nur bei Wahl des Compilers. Die Variable `MPdir` wird weiter unten für die Pfade zu den Binaries nocheinmal benötigt.

```
# ----------------------------------------------------------------
# - Linear Algebra library (BLAS or VSIPL) -----------------------
# ----------------------------------------------------------------
LAdir        = /home/uhed0035/intel/mkl/10.1.0.015
LAinc        = -I$(LAdir)/include
LAlib        = $(LAdir)/lib/em64t/libmkl_intel_lp64.a \
               $(LAdir)/lib/em64t/libmkl_em64t.a \
               $(LAdir)/lib/em64t/libguide.a -lpthread

#pgcc# LAdir = /home/uhed0035/software/libatlas
#pgcc# LAinc = -I$(LAdir)/include
#pgcc# LAlib = $(LAdir)/lib/libcblas.a \
               $(LAdir)/lib/libatlas.a
```

Hier sind der Installationspfad zur IMKL bzw. ATLAS einzutragen.

```
HPL_OPTS     =
#pgcc# HPL_OPTS = -DHPL_CALL_CBLAS
```

Bei Benutzung des `pgcc` muss HPL auf die C-Version der BLAS hingewiesen werden. Für IMKL kann diese Variable leer bleiben. Um detaillierte Timing Informationen zu bekommen, wäre noch der Parameter `DHPL_DETAILED_TIMING` möglich. Allerdings schadet die Unterlassung auch nicht.

```
# ----------------------------------------------------------
# - Compilers / linkers - Optimization flags -----------------
# ----------------------------------------------------------
CC           = $(MPdir)/bin/mpicc
CCNOOPT      = $(HPL_DEFS) -O0
CCFLAGS      = $(HPL_DEFS) -fomit-frame-pointer -O3 \
               -funroll-loops -W -Wall
LINKER       = $(MPdir)/bin/mpif77
LINKFLAGS    = $(CCFLAGS)

#pgcc# CC           = $(MPdir)/bin/mpicc
#pgcc# CCNOOPT      = $(HPL_DEFS) -O0 -Kieee
#pgcc# CCFLAGS      = $(HPL_DEFS) -Mnoframe -O3 -Munroll
#pgcc# LINKER       = $(MPdir)/bin/mpif77
#pgcc# LINKFLAGS    = $(CCFLAGS) -Mnomain
```

Compiler und Flags können hier angepasst werden. HPL benötigt jedoch einen MPI-fähigen Übersetzer und daher sind die entsprechenden OpenMPI für `gcc` oder `pgcc` hier mit Pfad anzugeben.

Schließlich kann der HPL mit dem Kommando `make arch={ARCH}` übersetzt werden. Nach erfolgreichem Vorgang ist das Binary unter `bin/{ARCH}` mitsamt einer Beispielkonfigurationsdatei zu finden.

A.2 Durchführung eines Benchmarks

Wegen der Benutzung einer Queue wird eine Batchdatei benutzt, um entsprechende PBS-Parameter zu setzen. Im Folgenden wird so eine minimale Datei erläutert.

```
#PBS -q dgipar -j oe -o mein.lauf.out
#PBS -l nodes=2:ppn=8
#PBS -l pmem=2000MB
#PBS -l walltime=36:00:00
```

Über die Bedeutung des Headers wurde schon in Abschnitt 4.3.1 geschrieben.

```
source /etc/profile
wdir=~/software/hpl-2.0/bin/Linux.imkl/bench/mein.lauf
```

Die Systemumgebung muss wegen bestimmter Standardpfade eingebunden werden. Ebenfalls wird hier die Variable `wdir` für spätere Benutzung auf das Arbeitsverzeichnis des Prozesses gesetzt. Ohne diese Angabe würde HPL die Konfigurationsdatei nicht finden und abbrechen.

```
module load openmpi_gcc4
#pgcc# module load openmpi_pgcc
```

Je nach Compiler ist hier das entsprechende Modul zu laden. Sonst schlägt das `mpiexec` Kommando am Schluss dieser Datei fehl.

```
mca="-mca btl openib,self,sm"
### mca="-mca btl ^tcp"
mpiopt="-wd $wdir"
```

OpenMPI lässt sich per `-mca` Parameter umfangreich konfigurieren. In diesem wird mitgeteilt, dass Infiniband, *loopback* auf den eigenen Prozess und *shared memory* zur Kommunikation benutzt werden darf. Die kommentierte Zeile schließt beispielsweise TCP/Ethernet explizit aus, was unterm Strich dasselbe bedeutet.

```
mpiexec -v $mca $mpiopt ${wdir}/xhpl
```

Nun muss ein MPI-Prozess erzeugt werden. Die Angabe der Prozesse entfällt bei Verwendung der Queue. Sie steht dafür im Header (siehe oben).

Diese Batchdatei kann nun per `qsub mein.lauf.batch` in die Queue gegeben werden. Im Arbeitsverzeichnis ist nach Beendigung des Laufes auch die im Header angegebene Logdatei zu finden. Da die Konfigurationsdatei von HPL nicht geändert werden kann, solange ein Lauf noch in der Warteschlange ist, empfiehlt sich ein Unterverzeichnis pro Job. Die Variable `wdir` ist ensprechend anzupassen, damit die richtige `HPL.dat` benutzt wird.

Beispiele dieser Dateien finden sich auf der beigelegten DVD.