**Prof. Dr. R. Loogen, Dipl. Inform. Jost Berthold**
**PHILIPPS-UNIVERSITÄT MARBURG**
**Fachbereich Mathematik und Informatik**

# Bulk Synchronous Programming und LogP

# Seminar zur Parallelen Programmierung

# Simone Geisel

**2005-11-16**

Valiant, L.G., 1990. A Bridging Model for Parallel Computation, Communications of the ACM, 33:8, August 1990, pages 103-111

Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K.E., Santos, E., Subramomian, R., and von Eicken, T. 1993. LogP: Towards a Realistic Model of Parallel Computation, Proceedings of the Fourth ACM SIGPLAN Symposium on Principles an Practice of Parallel Programming, May 1993, pages 1-12.

Gianfranco Bilardi, Kieran T Herley, Andrea Pietracaprina, Geppino Pucci, Paul Spirakis: BSP vs LogP, ACM SPAA 96, Padua Italy 1996.

## 1. Abstract

In diesem Artikel werden zwei Modelle für parallele Berechnungen vorgestellt, zum einen das BSP Modell (Bulk Synchronous Programming) und zum anderen das LogP Modell. Beide haben das Ziel analog zum von Neumann Modell eine Brücke zwischen Software und Hardware für parallele Berechnungen zu schlagen. Sie sollen als Basis für die Entwicklung schneller, portabler, paralleler Algorithmen dienen und den Maschinendesignern Richtlinien bieten. Nach einer ausführlichen Beschreibung des BSP und des LogP Modells folgt eine quantitative Gegenüberstellung, bei der mit Kreuzsimulationen gearbeitet wird. Die Analyse ist von asymptotischer Natur und kommt zu dem Schluss, dass beide Modelle innerhalb des Bandweiten-Latenzzeit Rahmens für das Modellieren paralleler Berechnungen sehr eng verwandte Varianten sind, dennoch gibt es leichte Präferenzen für das BSP Modell.

## Inhaltsverzeichnis



## 2. Einführung

Mit dem BSP und dem LogP Modell sind zwei Modelle für parallele Berechnung entworfen worden, die das Ziel haben als Standard Modelle bei der Entwicklung von paralleler Software und Hardware zu dienen. Dadurch soll die Herstellung von allgemeingebräuchlichen Maschinen und transportabler Software ermöglicht werden. Der Vorwurf an die bisher entwickelten Algorithmen lautet, dass sie unpraktisch sind und künstliche Faktoren ausnutzen wie Null-Kommunikationsverzögerung oder unendliche Bandweite oder dass sie zu speziell sind und nur auf bestimmte Topologien passen. Beispielsweise geht die PRAM davon aus, dass alle Prozessoren synchron arbeiten und Interprozessorkommunikation kostenlos ist.

Zuerst wurde 1990 das BSP Modell entworfen. Ausgehend davon wurde LogP drei Jahre später entwickelt. In BSP sind die fundamentalen Primitiven globale Barrierensynchronisation und das Routen von beliebigen Nachrichtenmengen. LogP stellt einen etwas eingeschränkteren Message-Passing Mechanismus zur Verfügung mit einer Kapazitätsbedingung und hat keine explizite Synchronisation. Daher bietet BSP intuitiv eine überzeugendere Abstraktion für Algorithmen Design und Programmierung, während LogP mehr Kontrolle über die Maschinenressourcen bietet.
Die Gegenüberstellung der beiden Modelle besteht aus zwei Teilen. Der erste besteht aus einer Messung des Slowdowns, der entsteht, wenn ein Modell das andere simuliert. Das bedeutet, dass ein LogP Computer das BSP Modell simuliert und umgekehrt. Der zweite Vergleich bezieht sich auf die Leistung der beiden Modelle auf der gleichen Hardware. Hierfür werden Hardware Plattformen betrachtet, die als Punkt-zu-Punkt Netzwerke modelliert werden können.
Dieser Artikel gliedert sich folgendermaßen. Kapitel 3 widmet sich der Vorstellung des BSP Modells, was die Definition des BSP Computers und die Einführung der für die Laufzeit relevanten Parameter beinhaltet. In Kapitel 4 wird anhand eines Beispiels über Broadcasting und Matrixmultiplikation das BSP Modell verdeutlicht.
Kapitel 5 befasst sich mit der Einführung von LogP, dessen Hauptparametern und einigen Einschränkungen, die das Modell macht. Im anschließenden Kapitel werden die Beispiele Broadcast und der Butterfly Algorithmus erläutert. In Kapitel 7 findet der Vergleich beider Modelle statt, der aus Kreuzsimulationen besteht. Darauf folgen eine Schlussbemerkung und das Literaturverzeichnis.

## 3 BSP

### 3.1 Das BSP Modell

Das BSP Modell wurde 1989 von Valiant entwickelt. Das BSP Modell der parallelen Berechnung oder der BSP Computer ist definiert als die Kombination von drei Elementen:

1. Einer Anzahl von Komponenten **p**, von denen jede Prozess- oder Speicherfunktionen durchführt,
2. Einem Router, der Nachrichten Punkt zu Punkt zwischen Komponentenpaaren ausliefert und
3. Anlagen für die Synchronisation aller oder einer Teilmenge der Komponenten, die in regulären Intervallen durchgeführt wird.

Ein BSP Computer besteht also aus einer Menge von Prozessoren, bei denen jeder seinen eigenen Speicher besitzt, ein distributed-memory Computer.
Eine Berechnung besteht aus einer Sequenz von **Superschritten**, der Länge $\mathbf{L_{super}}$. In einem Superschritt führt jeder Prozessor lokale Berechnungen aus und empfängt und sendet Nachrichten mit folgenden Einschränkungen: Die Berechnung darf nur mit Daten durchgeführt werden, die sich zu Beginn des Superschritts im lokalen Speicher des Prozessors befinden und ein Prozessor kann in einem Superschritt höchstens h Nachrichten empfangen und höchstens h Nachrichten senden. Dieses Kommunikationsmuster wird **h-Relation** genannt. Nachrichten, die zu Beginn eines Superschritts gesendet werden, können nur im nächsten Superschritt verwendet werden, auch wenn sie vor dem Ende des Superschritts am Zielprozessor ankommen. Nach $L_{super}$ Zeiteinheiten wird ein globaler Check durchgeführt um zu bestimmen, ob der Superschritt von allen Komponenten vollendet wurde. Wenn alle fertig sind, fährt die Maschine mit dem nächsten Superschritt fort. Ansonsten wird für den unvollständigen Superschritt die nächste Periode von $L_{super}$ Zeiteinheiten bereitgestellt. Für die Analyse von parallelen Algorithmen werden die Berechnungs- und die Kommunikationsphase getrennt. In der Praxis findet diese Trennung im Superschritt aber nicht statt.

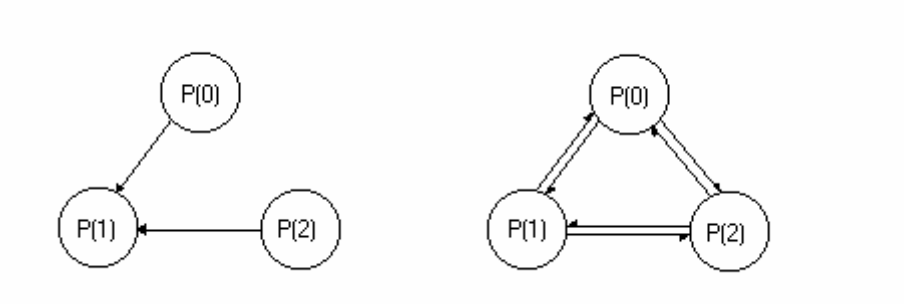


Bild 1: zwei Möglichkeiten einer 2- Relation

### 3.2 Die Laufzeitberechnung

Die Laufzeit eines Superschritts berechnet sich aus der **Bandweite *g***, der **Latenzzeit *ℓ*** und der maximalen Anzahl an **lokalen Operationen *w***, die von einem der Prozessoren durchgeführt werden:

$$T_{super} = w + g*h + \ell =: L_{super}$$

*g* ist der Basisdurchlauf der Nachricht durch den Router, wenn er in ständigem Gebrauch ist.
*ℓ* sind die Startup und Latenzzeitkosten, die auch eine obere Grenze für die globale Barrierensynchronisation sein müssen (wenn $w = 0$, $h = 0$). Ursprünglich tauchte der Parameter *ℓ* in der Berechnung nicht explizit auf, sondern *g* wurde soweit erhöht, dass er die Zusatzkosten abschätzen konnte. Darum wird er in dem Beispiel zum BSP Modell nicht berücksichtigt, er wird aber später im Vergleich mit LogP auftauchen.
Der Parameter *h* hat in allen Superschritten denselben Wert, wodurch Wartezeiten für einige Prozessoren entstehen, wenn sie weniger als *h* Nachrichten senden oder empfangen. Genauso kann es passieren, dass Prozessoren in der Berechnungsphase warten müssen, weil sie weniger als *w* Operationen durchführen.

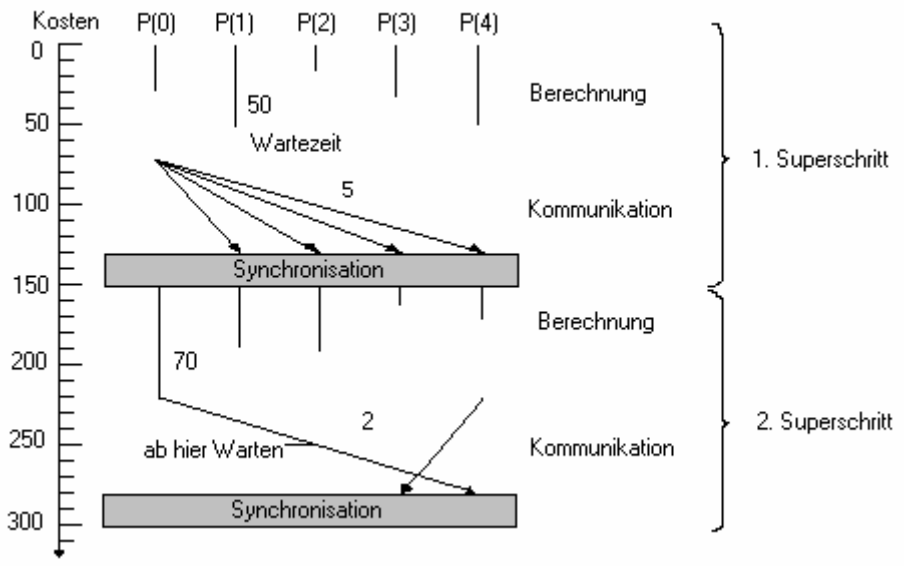


Bild2: Für $\ell = 20$, $g = 15$, $p = 5$, $h = 4$, $w = 70$ ist die Länge der Superschritte $T = 70 + 15*4 + 20 = 150$. Weil im 1. Schritt die Berechnungszeit nur 60 Zeiteinheiten beträgt und im 2. Schritt die Kommunikation nur $2*15 = 30$ dauert, müssen die Prozessoren warten.

Ein Algorithmus im BSP Modell wird in Superschritte zerlegt, in denen die Wörter, die im aktuellen Superschritt gelesen werden, alle das letzte Mal in dem vorhergehenden Superstep verändert wurden. In einem Superschritt der Periodizität $L_{super} = T$ können T/2 lokale Operationen und $\lceil T/2g \rceil$ – Relationen realisiert werden. Die Parameter einer Maschine sind

deswegen $L_{super}$, *g* und p. Jeder Algorithmus hat außerdem einen Parameter n, die Problemgröße. Die Komplexität eines Algorithmus kann auf verschiedene Weise durch Terme aus diesen Parametern ausgedrückt werden. Wir werden parallele Algorithmen beschreiben, in denen das Produkt aus Zeit und Prozessor die Anzahl der Operationen für Berechnungen nur um eine feste multiplikative Konstante übersteigt, unabhängig von $L_{super}$, *g*, p und n, vorausgesetzt $L_{super}$ und *g* sind unter kritischen Werten. In solchen „optimalen" Algorithmen gibt es immer noch einige Richtungen für mögliche Verbesserungen, nämlich in den multiplikativen Konstanten ebenso wie in den kritischen Werten von *g* und $L_{super}$.

## 4. Beispiele

### 4.1 Matrixmultiplikationen

Als ein einfaches Beispiel für dicht synchronisierte Algorithmen, die gut geeignet für direkte Implementierung sind, betrachten wir die Multiplikation zweier n x n Matrizen A und B und benutzen den Standardalgorithmus für p <= $n^2$ Prozessoren. Jedem Prozessor wird das Subproblem der Berechnung einer $n/\sqrt{p}$ x $n/\sqrt{p}$ Submatrix des Produktes zugewiesen. Zuerst muss jeder Prozessor Daten, die $n/\sqrt{p}$Zeilen von A und die $n/\sqrt{p}$Spalten von B, empfangen. Dann muss jeder Prozessor $w = 2n^3/p$ Additionen und Multiplikationen durchführen und $h = 2n^2/\sqrt{p} <= 2n^3/p$ Nachrichten empfangen. Wenn jeder Prozessor $2n^2/\sqrt{p}$ Nachrichten Übermittlungen zusätzlich macht, wird die Laufzeit sicherlich nur durch einen konstanten Faktor verändert. Glücklicherweise werden nicht mehr als diese Anzahl von Übermittlungen benötigt, selbst wenn die Elemente einfach von der Quelle repliziert werden. Dies kommt daher, dass wenn die Matrizen A und B am Anfang gleichmäßig über die p Prozessoren verteilt sind, $2n^2/p$ Elemente bei jedem, und jeder Prozessor jedes seiner Elemente $\sqrt{p}$ mal repliziert und diese zu den $\sqrt{p}$ Prozessoren sendet, die diese Einträge benötigen, die Anzahl der Übertragungen pro Prozessor $2n^2/\sqrt{p}$ sein wird. Das ist ein Beispiel dafür, dass konkurrierende Zugriffe, wenn die Zugriffsmultiplizierende *h* genügend klein ist, effizient implementiert werden können durch Replizieren der Daten in der Quelle. Aus $L_{super} = hg + w$ folgt, dass die optimale Laufzeit $O(n^3/p)$ erreicht wird, vorausgesetzt $g = O(n/\sqrt{p})$ und $L = O(n^3/p)$.

### 4.2 Broadcast

Ein Fall, in dem es ineffizient wäre, mehrere Zugriffe durch Replikation in der Quelle zu realisieren, ist Broadcasting. Hierbei muss ein Prozessor Kopien einer Nachricht zu jedem der n Speicherpositionen schicken, die gleichmäßig zwischen den p Komponenten verteilt sind. Das Versenden einer Kopie zu jeder der p Komponenten kann in $\log_d$ (p) Superschritten erreicht werden durch Ausführen eines logischen d – näre Baums (für d = 2 binär). In jedem Superschritt überträgt jeder Prozessor, der beteiligt ist, d Kopien zu verschiedenen Komponenten. Der Zeitaufwand hierfür beträgt dg $\log_d$ (p). Wenn n/p – 1 weitere Kopien in jeder Komponente gemacht werden, kann Optimalität, hier Laufzeit von O(n/p), erreicht werden, wenn $d = O((n/(gp \log p))\log(n/(gp \log p)))$ und $L_{super} = O(gd)$.

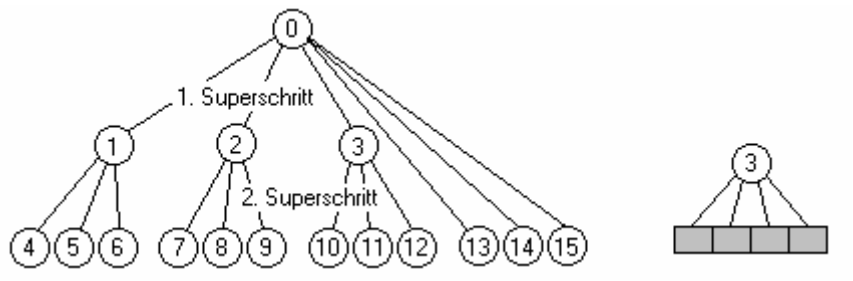

Bild 3: Broadcast Tree für n = 80, p = 16, d = 3, g = 4
$T = dg \log_d (p) + n/p - 1 = 2*3*4 + 80/16 - 1 = 28$ Die Knoten sind mit den Prozessor IDs beschriftet, die Linien verbinden kommunizierende Prozessoren. n/p – 1 = 4 Kopien werden je Komponente erstellt Die 4 grauen Kästchen stellen die Speicher-positionen dar.

Eine Operation, die schwieriger als Broadcasting ist, ist paralleles Prefix. Gegeben ist $x_1,...x_n$ und zu berechnen ist $x_1$ o $x_2$ o ... o $x_i$ für 1<=i<=n für eine assoziative Operation o. Der jetzige rekursive Standardalgorithmus dafür ist mit d –närer anstelle binärer Rekursion und gibt exakt die gleiche Beschränkung wie die zuvor geforderte für Broadcasting her.

## 5. LogP

### 5.1 Das LogP Modell

Ebenso wie das BSP Modell geht auch das LogP Modell von einem Multiprozessor mit verteiltem Speicher aus, in dem die Prozessoren über Punkt-zu-Punkt Nachrichten kommunizieren.
Das LogP Modell charakterisiert eine parallele Maschine anhand folgender Parameter:

**P**: die Anzahl der **Prozessor** und Speicher Module. Wir setzen die Dauer einer lokalen Operation als eine Zeiteinheit fest und nennen sie Zykel.
**G**: die **Kommunikationsbandweite**, definiert als das minimale Zeitintervall zwischen zwei aufeinaderfolgenden Sendevorgängen eines Prozessors bzw. Nachrichtenempfängen von einem Prozessor.
**L**: eine obere Grenze für die Kommunikationsverzögerung oder **Latenzzeit**, die durch die Übertragung einer Nachricht, die ein Wort enthält, von ihrer Quelle zu ihrem Ziel verursacht wird.
**o**: die **Kommunikationskosten**, definiert als die Zeitlänge, die ein Prozessor mit dem Empfangen oder Senden einer Nachricht beschäftigt ist. Während dieser Zeit kann der Prozessor keine andern Operationen durchführen.

Weiterhin wird angenommen, dass das Netzwerk nur eine begrenzte Kapazität hat, weil mehr Nachrichten um die Ressourcen kämpfen. Es gibt einen Sättigungspunkt, ab dem die Latenzzeit stark ansteigt. Unterhalb dieses Punktes ist die Latenzzeit unempfindlich gegenüber der Anzahl der Nachrichten. Deshalb gibt es in LogP die Kapazitätsbedingung, dass zu einem Zeitpunkt höchstens $\lceil L/2G \rceil$ Nachrichten von einem oder zu einem Prozessor im Netzwerk unterwegs sein dürfen. Wenn ein Prozessor eine Nachricht senden will und damit die Schranke überschreiten würde, muss er warten. Bevor ein Prozessor senden kann, muss er also auf die Akzeptanz seiner Nachricht warten. Mit dieser Schranke kann garantiert werden, dass alle Nachrichten zur Zeit t = L übertragen sind.
Der Faktor ½ wurde eingeführt um einige möglicherweise unvorgesehene Konsequenzen zu vermeiden. Zum Beispiel sei G = 1 und zum Zeitpunkt t = 0 senden L Prozessoren simultan zum gleichen Prozessor eine Nachricht. Weil das Ziel nur eine Nachricht zu einem Zeitpunkt erhalten kann, muss eine Nachricht ankommen für jedes t = 1,2,...,L. Um zur Zeit t = L die Übertragung abzuschließen muss es für jeden Prozessor j und jedes L-Tupel von Prozessoren $i_1, i_2,... i_L$ ein $i_h$ geben, von dem j in einem Zeitschritt erreicht werden kann. Dies ist sicherlich eine strenge Performance Bedingung und schwer zu unterstützen auf einer Maschine.
Das Modell ist asynchron, das heißt, dass die Prozessoren asynchron arbeiten und eine Nachricht benutzen können, sobald sie ankommt. Das Modell fördert das Planen der Berechnung und das Überlappen von Berechnung und Kommunikation, innerhalb der Grenzen der Netzwerkkapazität. Diese Grenze unterstützt auch balancierte Kommunikationsmuster, bei dem kein Prozessor mit ankommenden Nachrichten überflutet

wird. Mit Hilfe des LogP Modells lassen sich leicht Berechnungs- und Kommunikationspläne erstellen.
Unter LogP benötigt das Lesen einer entfernten Speicherposition die Zeit 2L + 4o.

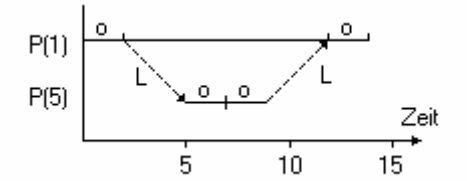


Bild 4: Prozessor 1 will Daten aus dem lokalen Speicher von Prozessor 5 lesen. Dazu schickt er erst eine Anfrage per Nachricht und dann werden ihm die Daten gesendet. Für L = 3 und o = 2 dauert dies 14 Zykel.

## 6. Beispiele

### 6.1 Broadcast

Zunächst wird der Broadcasts eines einzelnen Datums von einem Prozessor zu P-1 anderen gezeigt. Die Hauptidee ist einfach: alle Prozessoren, die ein Datum erhalten haben, senden es so schnell wie möglich weiter, wobei sie sicherstellen, dass kein Prozessor mehr als eine Nachricht erhält. Hierbei wird wie im BSP Modell mit einer Baumstruktur gearbeitet. Die Quelle des Broadcasts beginnt die Übermittlung des Datums zur Zeit 0. Das erste Datum gelangt in das Netzwerk zur Zeit o, benötigt L Zykel um sein Ziel zu erreichen und wird vom Knoten empfangen zur Zeit L + 2o. In der Zwischenzeit hat die Quelle die Übermittlung zu anderen Prozessoren initiiert zur Zeit g, 2g,... angenommen g >= o, jeder von ihnen handelt als Wurzel eines kleineren Broadcast Baumes. Wie in Bild 5 zu sehen, ist der optimale Broadcast Baum für p Prozessoren unbalanciert. Dabei ist zu beachten, dass die Prozessorkosten der sukzessiven Übermittlung die Auslieferung von früheren Nachrichten überlappt. Die Knoten müssen am Ende des Algorithmus Leerlaufzyklen durchlaufen, während die letzten Nachrichten in der Übermittlung sind.

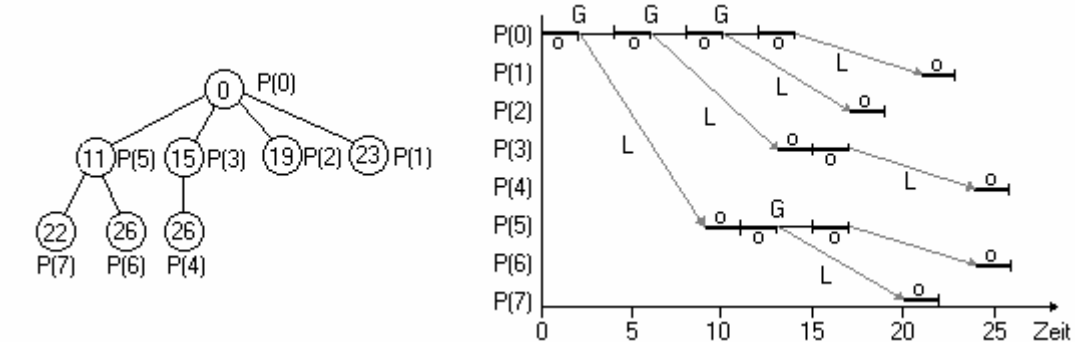


Bild 5: Links ein optimaler Broadcast Baum für P = 8, L = 7, G = 4, o = 2 und rechts die Aktivitäten eines jeden Prozessors im Laufe der Zeit. Die Nummern in den Knoten geben den Zeitpunkt an, an dem der Prozessor das Datum empfangen hat und mit dem Weitersenden beginnen kann. Der letzte Wert wird zum Zeitpunkt 26 empfangen.

### 6.2 Butterfly Algorithmus

Broadcast war nur ein einfaches Problem, um die Parameter besser zu verstehen. Nun widmen wir uns einem schwereren Beispiel, der schnellen Fourier Transformation. Diese kann mit Hilfe des Butterfly Algorithmus berechnet werden, der am besten anhand seines Berechnungsgraphen beschrieben werden kann. Der n-Input Butterfly, mit $n = 2^k$, ist ein gerichteter azyklischer Graph mit n(log n +1) Knoten, die in n Zeilen mit log n+1 Spalten angeordnet sind. Bild 6 zeigt einen 8 –Input Butterfly.

Die Knoten in Spalten 0 sind die Problem Inputs und die in Spalte log n sind die Ergebnis Outputs der Berechnung. Jeder Knoten, der kein Input Knoten ist, repräsentiert eine komplexe Operation, deren Dauer auf eine Zeiteinheit festgesetzt wird. Das Implementieren des Algorithmus auf einem parallelen Computer entspricht dem Verteilen der Butterfly Knoten auf die Prozessoren. Dieser Entwurf bestimmt den Berechnungs- und den Kommunikationsplan.

Ein natürlicher Entwurf ist der ersten Zeile des Butterflys den ersten Prozessor zu zu weisen, der zweiten Zeile den zweiten Prozessor und so weiter. Dies wird als zyklischer Entwurf bezeichnet. In diesem Entwurf benötigen die ersten log n/P Spalten der Berechnung nur lokale Daten, wobei die letzten P Spalten eine entfernte Referenz für jeden Knoten benötigen. Eine Alternative wäre die ersten n/P Zeilen auf den ersten Prozessor zu platzieren, die nächsten n/P Zeilen auf den zweiten und so weiter. In diesem Block Entwurf benötigt jeder der Knoten der ersten log P Spalten entfernte Daten für seine Berechnung, während die letzten (log n)/P Spalten nur lokale Daten benötigen. In beiden Entwürfen braucht jeder Prozessor n * log n/P Zeit für die Berechnung und (G*n/P + L) log P Zeit für die Kommunikation, vorrausgesetzt G >= 2o.

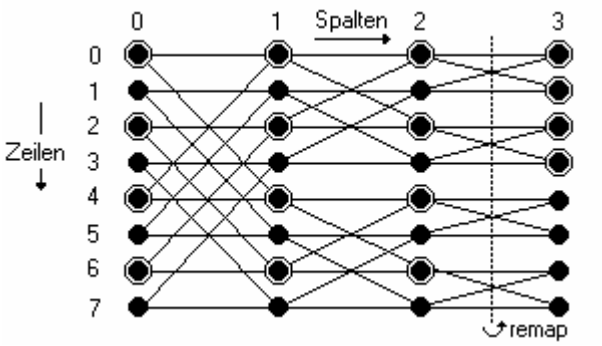


Bild 6: Ein 8-Input Butterfly mit P = 2. Die Knoten, die dem 0. Prozessoren zugewiesen sind, sind umkringelt.

Weil die Berechnungen am Anfang des zyklischen Entwurfs und die Berechnungen am Ende des Block Entwurfs komplett lokal sind, wurde ein hybrider Entwurf entwickelt, der auf den ersten log P Spalten zyklisch ist und blockartig auf den letzten log P. Tatsächlich führt ein Wechsel zwischen zyklischem und blockartigem Entwurf zu einem Algorithmus, welcher einen einzelnen „all-to-all“ Kommunikationsschritt zwischen zwei völlig lokalen Berechnungsphasen hat. Bild 6 zeigt die Knoten Zuweisung für den Prozessor 0 für einen 8-Input FFT mit P = 2 mit dem Hybrid – Entwurf; der Wechsel geschieht zwischen der 2 und der 3. Spalte.

Die Berechnungszeit beim Hybrid Entwurf ist dieselbe wie bei einfacheren Entwürfen, aber die Kommunikationszeit ist geringer um den Faktor Log P: jeder Prozessor sendet $n/P^2$ Nachrichten zu jedem anderen, was nur $G(P-1)n/P^2 + L = G(n/P - n/P^2) + L$ Zeit benötigt. Die absolute Zeit ist innerhalb eines Faktor (1+G/log n) optimal, was zeigt, dass dieser Entwurf das Potential zum fast perfekten Speedup für große Probleme hat. Für den Kommunikationsschritt im Hybrid Entwurf werden hier zwei Möglichkeiten vorgestellt. Ein einfacher Plan würde einfach jeden Prozessor Daten senden lassen, beginnend mit der Zeile, in der er das erste Mal senden muss, bis runter zur letzten am Ende. Aber dann würde jeder Prozessor zuerst Daten zu Prozessor 0 schicken und dann alle zu Prozessor 1 und so weiter.

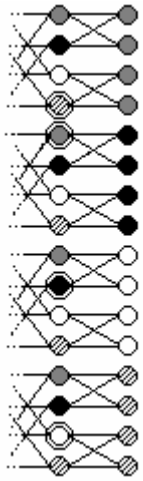

Bild7: Bei P = 4 und 16-Input Butterfly bekommt Prozessor 0 mit dem einfachen Plan Nachrichten von 3 anderen Prozessoren. Nach der versetzten Sendeanordnung beginnen die umkringelten Prozessoren und senden alle zu verschiedenen Empfängern

Dies wird in Bild 7 dargestellt. Alle bis auf L/2G Prozessoren würden am Anfang warten müssen. Ein besserer Plan ist es die beginnenden Zeilen versetzt anzuordnen: Prozessor i startet mit seiner $i*n/P^2$ -ten Zeile und fährt fort bis zur letzten und beginnt dann von oben.

### 6.3 Implementierung des FFT Algorithmus

Um die Vorhersage der Analyse zu verifizieren implementierten wurde der hybride Algorithmus auf einem CM-5 Multiprozessor implementiert und die Leistung des Algorithmus in den drei Phasen gemessen: (I) Berechnung in einem zyklischen Entwurf, (II) Wechsel der Daten, Remapping, und (III) Berechnung im Block Entwurf. Bild 8 zeigt die Bedeutung des Kommunikationsplans: die drei Kurven zeigen die Berechnungszeit und die Kommunikationszeit für die zwei Kommunikationspläne. Im einfachen Plan benötigt das Remapping mehr als 1,5 mal so lang wie die Berechnung, wohingegen mit der versetzten Anordnung nur 1/7-mal so lang.

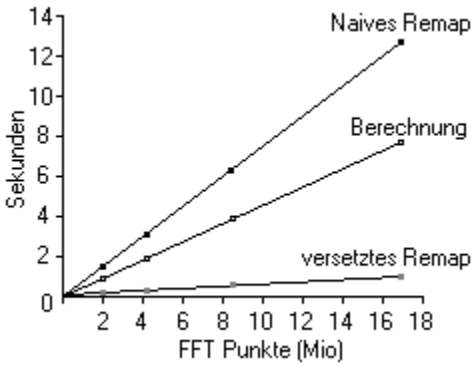


Bild 8: Ausführungszeit für einen FFT mit variierender Größe auf einem 128 Bit Prozessor CM-5

Die zwei Berechnungsphasen umfassen nur lokale Operationen und sind Standard FFTs. Bild 9 zeigt die Berechnungsrate über eine Reihe von FFT Größen ausgedrückt in MFlops/Prozessor. Zum Vergleich, ein CM-5 Sparc Knoten erreicht nur ungefähr 3,2 MFlops mit dem Linpack Benchmark. Dieses Beispiel zeigt einen passenden Vergleich der relativen Wichtigkeit von Cache Effekten, die wir hier vernachlässigen wollen, und Kommunikationsbalance, was andere Modelle ignorieren. Der Leistungsabfall des lokalen FFTs von 2,8 MFlops auf 2,2 MFlops taucht auf, wenn die Größe des lokalen FFTs die Cache Kapazität übersteigt. Für große FFTs erleidet die zyklische Phase, die einen großen FFT enthält, mehr Cache Interferenz, das heißt die gewünschten Daten sind nicht im Cache, als die Block Phase, die viele kleine FFTs löst.

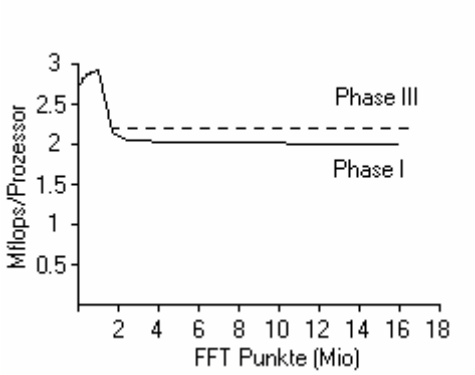


Bild 9: Die Berechnungsrate pro Prozessor für die zwei Berechnungsphasen des FFT in Mflops

### 6.4 Quantitative Analyse

In der quantitativen Analyse soll die Laufzeit des implementierten Algorithmus vorausgesagt werden. Mit Hilfe der Berechnungszeit, die in Bild 9 dargestellt wird, kann die Länge eines Zykels festlegt werden. Diese Zeiteinheit entspricht der Dauer der komplexen Mehrfachadditionen des FFTs, die pro Prozessorknoten im Butterfly durchgeführt werden. Bei einem Mittelwert von 2,2 MFlops und 10 Floating-Point Operationen pro Butterfly entspricht ein Zykel 4,5 µs. In vorherigen Experimenten auf der CM-5 haben wir berechnet, dass $o \approx 2\mu s = 0{,}44$ Zykel, und auf einem ungeladenen Netzwerk, $L \approx 6\mu s = 1{,}3$ Zykel. Für G werden aufgrund der Bisectionsbandweite von 5MB/s pro Prozessor für Nachrichten mit 16 Byte Daten und 4 Byte Adresse $4\mu s = 0{,}88$ Zykel berechnet. Zusätzlich dauert es ungefähr $1\mu s = 0{,}22$ Zykel für lokale Berechnungen pro Datenpunkt für das Laden oder das Speichern. Analysen der gestaffelten Remapphase sagen voraus, dass die Kommunikationszeit $n/p^* \max\{1\mu s+2o,G\} + L$ ist. Für unsere Parameterwerte ist die Übermittlungsrate durch die Bearbeitungszeit und die Kommunikationskosten beschränkt, weil die Bandweite kleiner ist ($1\mu s+2o=5\mu s > 4\mu s = G$). Diese detaillierte quantitative Analyse der Implementierung zeigt, dass der hybride FFT Algorithmus Entwurf fast optimal auf der CM-5 ist. Die Berechnungsphasen sind rein lokal und die Kommunikationsphase ist Kostenbeschränkt,

obwohl die Prozessoren die ganze Zeit 100% ausgelastet sind. Leistungsverbesserungen in der Implementierung sind sicherlich möglich, aber sie haben keine Auswirkungen auf den Algorithmus selbst.

## 7. Vergleich beider Modelle

Da das LogP Modell zwei Parameter besitzt, die das BSP Modell nicht explizit berücksichtigt, müssen nun kleine Modifikationen vorgenommen werden. Das BSP Modell wird weiterentwickelt und erhält nun einen eigenen Latenzzeit Parameter $\ell$ und im LogP Modell wird o durch G abgeschätzt, für G >= o wird G auf G + o <= 2G gesetzt.
Die erste Simulation befasst sich mit der Messung des Slowdowns, der entsteht, wenn man ein Modell auf dem Computer des anderen simuliert, d.h. auf einem Rechner, der die Erfordernisse des jeweiligen Modells erfüllt. In der zweiten Simulation werden die beiden Modelle nacheinander auf der gleichen Maschine implementiert und für beide der Latenzzeit und der Bandweiten Parameter gemessen.

### 7.1 Simulation von LogP auf BSP

Zunächst wird die Simulation eines beliebigen LogP Programms auf einer BSP Maschine betrachtet. Dies lässt sich einfach realisieren, weil die BSP Maschine höhere Anforderungen erfüllt als das LogP Modell erfordert, beispielsweise enthält sie bereits einen Synchronisationsmechanismus und kann beliebig viele Nachrichten durch das Netzwerk leiten. Außerdem gibt es noch einige Einschränkungen an das auszuführende LogP Programm um den ursprünglich geplanten Gebrauch des LogP Modells zu zeigen. Es wird vorausgesetzt, dass in den LogP Programmen kein Prozessor warten muss und jede Nachricht nach spätestens L Zeiteinheit ankommt. Weiterhin darf jeder LogP Prozessor höchstens 1 Nachricht alle G Zeitschritte senden oder empfangen und die Kapazitätsbedingungen müssen eingehalten werden. Unter diesen Voraussetzungen wird folgende Simulation durchgeführt.
Der i-te BSP Prozessor imitiert die Aktivitäten des i-ten LogP Prozessors und benutzt seinen eigenen lokalen Speicher um die Inhalte des lokalen Speichers seines LogP Gegenstücks zu repräsentieren. Die Simulation wird als eine Sequenz von Zykeln durchgeführt, jede von ihnen simuliert die Effekte von L/2 aufeinanderfolgenden LogP Schritten. Ein Zykel wird ausgeführt mittels eines einzelnen BSP Supersteps und besteht aus zwei Perioden. In der ersten Periode überlappt jeder BSP Prozessor den Empfang der Nachrichten, die zu ihm während des vorherigen Zykels geschickt wurden, mit der lokalen Berechnung und der Nachrichten Generierung, die im LogP Programm für den aktuellen Zykel vorgeschrieben werden. Die Prozessoren erhalten ankommende Nachrichten alle G Zeiteinheiten, bis der dafür reservierte Speicher verbraucht ist, und führen lokale Berechnungen zwischen aufeinaderfolgenden Empfängen aus. In der zweiten Periode werden alle Nachrichten, die in der ersten Periode generiert wurden zu ihren vorbestimmten Zielen losgeschickt.
Weil das LogP Programm ausgeführt wird, ohne dass die Prozessoren warten müssen, ist sicher, dass in L/2 aufeinanderfolgenden Schritten nicht mehr als L/(2G) Nachrichten von einem Prozessor generiert werden und kein Prozessor das Ziel für mehr als L/(2G) solcher Nachrichten sein kann. Dies impliziert, dass in der ersten Periode des Simulationszyklus jeder BSP Prozessor höchstens L/(2G) ankommende Nachrichten zu lesen hat, und dass die zweite Periode das Routing einer h-Relation umfasst, wobei $h$ <= L/(2G). Deshalb ist die gesamte Laufzeit eines Zykels höchstens L/2 + $g$*(L/(2G)) + $\ell$. Angenommen, dass ein Zykel mit einem Segment der LogP Berechnung der Dauer L/2 korrespondiert (L/2 = 1), dann ergibt sich:

Slowdown für die Simulation von LogP mit BSP ist O(1+g/G + $\ell$/L).

Für $\ell$ = θ(L) und $g$ = θ(G), wird der Slowdown konstant.

### 7.2 Simulation von BSP auf LogP

Wir betrachten nun das umgekehrte Problem und simulieren ein BSP Programm auf einer LogP Maschine. Jeder LogP Prozessor simuliert die lokalen Berechnungen und Nachrichtenübermittlungen seines entsprechenden BSP Prozessors. Dies gestaltet sich schwieriger, weil eine LogP Maschine die Anforderungen eines BSP Programms nicht erfüllen kann. Am Ende des BSP Superschritts muss eine Synchronisation durchgeführt werden, für die die LogP Maschine keinen Mechanismus bietet und es müssen beliebige h-Relationen übermittelt werden, die die Kapazitätsbedingungen von LogP wahrscheinlich nicht erfüllen. Hierfür müssen Algorithmen gefunden werden, die diese beiden Probleme lösen. Diese werden im Folgenden vorgestellt.

#### 7.2.1 Synchronisation

Zur Simulation der Synchronisation wird Combine and Broadcast benutzt und das oben bereits vorgestellte Prefix. Für einen gegebenen assoziativen Operator op und p Input Variablen $x_0, x_1,...,x_{p-1}$ anfangs über verschiedene Prozessoren verteilt berechnet Combine and Broadcast op $(x_0, x_1,...,x_{p-1})$ und verteilt es an alle Prozessoren und Prefix schickt dem i-ten Prozessor op $(x_0, x_1,...,x_i)$ für $0 <= i <= p-1$.
Nach dem Abschluss der eigenen Aktivität im simulierten Superschritt trägt ein Prozessor den Wert 1 als Input in die Combine and Broadcast Berechnung mit einer Und - Verknüpfung ein. Der Superschritt ist abgeschlossen, wenn Combine and Broadcast eine 1 an alle Prozessoren schickt.
Ein einfacher Algorithmus für Combine and Broadcast kann mit einer Baumstruktur realisiert werden. Die p Prozessoren sind die Knoten eines Baums von Grad $\max\{2,\lceil L/(2G)\rceil\}$. Dadurch wird sichergestellt, dass zu einem Knoten nie mehr als $\lceil L/(2G)\rceil$ Nachrichten gesendet werden und somit die Kapazitätsbedingungen erfüllt werden. Die Blätter beginnen mit dem Senden ihrer booleschen Werte und die inneren Knoten warten auf die Werte ihrer Kinder, bevor sie sie mit ihrem eigenen kombinieren und weitersenden. Die Wurzel berechnet den finalen Wert und startet eine absteigende Broadcast Phase. Wenn alle Prozessoren gleichzeitig starten ist der Algorithmus zur Zeit

$$T_{CB} <= \frac{2*L*\log p}{\log(2+\lceil L/(2G)\rceil)} = 2*L*\log_{2+\lceil L/(2G)\rceil}(p)$$

mit seiner Berechnung fertig. Wenn die Prozessoren zu verschiedenen Zeiten den Algorithmus aufrufen, ist $T_{CB}$ die Zeit um Combine and Broadcast abzuschließen, gemessen ab der Eintrittszeit des letzten Prozessors. Dieser Algorithmus ist bis auf den Faktor 2 optimal, weil jeder Algorithmus für Combine and Broadcast mindestens $L*\log_{2+\lceil L/(2G)\rceil}(p)$ Zeit brauchen würde.
Die BSP Barrieren Synchronisation setzt das Ende des aktuellen Supersteps fest, indem sie überprüft, ob alle h-Relationen ihr Ziel erreicht haben. Die Barriere kann implementiert werden, indem der Routing Algorithmus mit der Ausführung von Combine and Broadcast überlappt, welches die gesendeten und empfangenen Nachrichten zählt und beide Werte miteinander vergleicht und terminiert, wenn sie übereinstimmen.
Daraus ergibt sich, dass eine Synchronisation für den BSP Superschritt auf LogP in der Zeit

$$T_{synch} = O(2*L*\log_{2+\lceil L/(2G)\rceil}(p))$$

durchgeführt werden kann.

### 7.2.2 Deterministisches Routen von h-Relationen

Das größte Hindernis am Routen beliebiger h-Relationen sind die LogP Netzwerk Kapazitätsbedingungen. Für $h <= \lceil L/(2G) \rceil$ kann eine h-Relation geroutet werden in G*h* + L<=3L/2, indem alle Prozessoren alle G Schritte ihre Nachrichten senden. Für größeres *h* würden die Kapazitätsbedingungen verletzt. Deshalb wird ein Mechanismus benötigt, der die h-Relationen in Subrelationen vom Grad $<= \lceil L/(2G) \rceil$ zerlegt.
Nach Halls Theorem kann jede h-Relation zerlegt werden in disjunkte 1-Relationen und deshalb vorab in optimal Gh+L Zeit in LogP übertragen werden. Diese Techniken sind tatsächlich nützlich, wenn die h-Relation zur Compile Zeit bestimmt werden kann.
Im Allgemeinen jedoch ist die h-Relation erst zur Laufzeit bekannt und die geforderte Zerlegung kann erst dann durchgeführt werden. Eine Standard Zerlegung basiert auf Sortieren. Bei einer h-Relation sei r (bzw. s) die maximale Anzahl von einem Prozessor gesendeter Nachrichten (bzw. die empfangen werden sollen), also *h* = max{r,s}. Der Parameter r kann von den LogP Prozessor mit Hilfe von Combine and Broadcast innerhalb von $r + T_{CB}$ berechnet werden. Das Protokoll für das Routen von *h*-Relationen besteht aus zwei Phasen:

1. Sortieren: Jeder Prozessor kreiert Dummy Nachrichten mit dem Ziel p, damit die Anzahl der Nachrichten, die von jedem Prozessor gesendet werden sollen, für alle Prozessoren r ist. Dann werden die p*r Nachrichten sortiert nach ansteigendem Ziel, so dass am Ende jeder Prozessor aufeinanderfolgende Nachrichten aus der sortierten Sequenz hat.
2. Nachrichtenauslieferung: Die Dummy Nachrichten werden wieder entfernt und die original Nachrichten werden nach einem geeigneten Plan zu ihren Zielen gesendet.

Diese zwei Phasen werden jetzt im Detail beschrieben.

### 7.2.3 Sortieren

Nachdem die Dummy Nachrichten erstellt wurden, haben wir insgesamt N = p*r Nachrichten, r pro Prozessor, die sortiert werden müssen. Wir betrachten Sortierschemen, die in zyklisch arbeiten, wobei jeder Zykel aus lokalen Berechnungen gefolgt von einer Datenumverteilung besteht, organisiert als eine vorberechnete Sequenz von 1-Relationen. Die lokale Berechnung beinhaltet das sequentielle Sortieren von r Paketen mit Schlüsseln [0...p-1]. So ein Sortieren kann von einem Prozessor in der Zeit

$$T_1(r) = r \min (\log r, \log p / \log r)$$

durchgeführt werden, indem eine optimale allgemeine Sortierstrategie, wie z.B. Mergesort, mit einem Radixsort Algorithmus kombiniert wird. Beachte, dass wenn $r = p^{\varepsilon}$, für eine positive Konstante ε, lokales Sortieren O(r) dauert.
Im Folgenden wird ein Sortierschema basierend auf dem AKS Netzwerk beschrieben. Das AKS Netzwerk kann als gerichteter Graph mit p Knoten, verbunden durch K = O (log p) Kantenmengen, wobei mit jedem Set ein Matching zwischen p Knoten realisiert wird, betrachtet werden. Für p = 2 entspricht der Sortieralgorithmus Odd-Even-Sort. Der Sortieralgorithmus läuft in K Schritten. Im i-ten Schritt tauschen die Endpunkte jeder Kante des i-ten Matchings ihre Nachrichten aus und bestimmen das Minimum oder das Maximum je nach ihrer Position unter Berücksichtigung der Richtung ihrer Kante. Wenn jeder Prozessor r>1 Nachrichten hat wird zuerst ein lokales Sortieren innerhalb eines jeden Prozessors durchgeführt und dann fährt der Algorithmus fort wie zuvor, indem er jede Vergleich und

Austausch Operation durch eine Misch und Aufteil Operation auf sortierten Sequenzen von r Nachrichten ersetzt.
Der obige Algorithmus kann einfach in LogP implementiert werden, weil die in jedem Schritt geforderte Nachrichtenübertragung bereits vorher bekannt sind und in eine Sequenz von r 1-Relationen zerlegt werden kann, die in der Zeit Gr+L übertragen werden können. Auf LogP ist die Laufzeit des Algorithmus

$$T_{AKS}(L,G,p,r) = O((Gr+L)*\log p)$$

Weil die Kosten des einleitenden Sortierschritts und des lokalen Mischens immer von den Kosten der Nachrichtenübermittlung dominiert werden.

### 7.2.4 Nachrichtenübertragung

Nach der Sortierphase sind die Nachrichten, die denselben Zielprozessor haben, benachbart in der sortierten Sequenz und deshalb formen sie eine Teilfolge, die Prozessoren zugewiesen werden, mit aufeinanderfolgenden Indexnummern. Durch die Benutzung einer Abwandlung der Prefix Primitive, führen wir zuerst ein *Segment Prefix* durch um Nachrichten, die für den selben Prozessor bestimmt sind, aufeinanderfolgenden Ränge zu zuweisen. Dann partitioniert jeder Prozessor seine Nachrichten in zwei Mengen. Die erste Menge enthält alle Nachrichten, die zu Teilfolgen gehören, die innerhalb der Sequenz des Prozessors beginnen und der Prozessor speichert die Nachrichten in der Teilfolge mit Rang 1. Die zweite Menge enthält die übrigen Nachrichten. Beachte, dass in jedem Prozessor die zweite Menge nur Nachrichten für ein Ziel enthält.

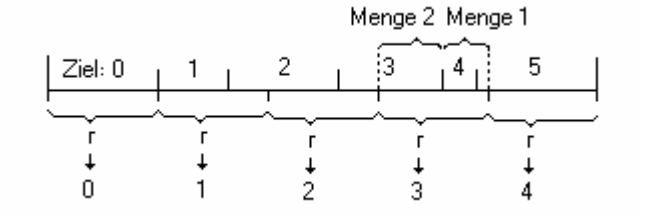


Bild 10: Die Aufteilung der zu sendenden Nachrichten für p = 4 und r > s. Oben steht das jeweilige Ziel der Nachrichten und unten steht der Prozessor, dem die r Nachrichten zugewiesen sind. Bei Prozessor 3 ist exemplarisch die Aufteilung in Menge 1 und Menge 2 dargestellt.

Das Übertragen der Nachrichten wird in zwei Phasen erledigt. In der ersten Phase sendet jeder Prozessor Nachrichten, die zu der ersten Menge gehören, in beliebiger Reihenfolge, eine Nachricht alle G Schritte. Die Kapazitätsbedingungen werden nie verletzt, weil nicht zwei Prozessoren Nachrichten für dasselbe Ziel senden. In der zweiten Phase senden die Prozessoren Nachrichten, die zu der zweien Menge gehören, gemäß ihrem Rang. Weil jeder Prozessor am Anfang r Nachrichten bereithält und es höchstens s Nachrichten mit demselben Ziel gibt, ist die totale Laufzeit des Nachrichtenauslieferns:

$$T_{Übertragung} = O(r + \log (2\lceil L/(2G)\rceil^{\underline{L \log}}) + G(r+s) + L)$$

Zusammenfassend ist die Gesamtzeit für die Simulation eines BSP Superschritts auf LogP mit maximal *w* lokalen Operationen

$$T = O(w + T_{synch} + T_{Übertragung} + T_{AKS})$$

$$= w + (Gh + L)\, S(L,G,p,h) \qquad \text{mit}$$

$$S(L,G,p,h) = (Gh + L)\ \log\left(2\lceil L/(2G)\rceil\right) + \log p \qquad \text{(L log p)}$$

$\Rightarrow$ $S(L,G,p,h) = O(\log p)$
$\Rightarrow$ $T = O(w + (Gh + L) \log p)$

Für G,L = O(p) und großes *h* ist S(L,G,p,*h*) = O(1). Der Slowdown beim Simulieren des BSP Modells auf LogP ist also höchstens logarithmisch.

### 7.3 Die Unterstützung von BSP und LogP Abstraktionen auf Punkt zu Punkt Netzwerken

Die vorangegangenen Abschnitte zeigen, dass die Simulation von BSP auf LogP erheblich komplizierter ist als andersherum. Weiterhin weist das letztere für G = $g$ und L = $\ell$ einen kleineren Slowdown auf als das erstere. Diese quantitativen Ergebnisse verstärken den Eindruck, dass BSP eine stärkere Abstraktion als LogP zur Verfügung stellt, wenn man von vergleichbarer Bandweite und Latenzzeit ausgeht.
Daher ist es vorstellbar, dass die BSP Implementierung höhere Werte für Bandweite und Latenzzeit hat, wenn man beide Modelle nacheinander auf der gleichen Hardware implementiert. Tatsächlich ist das wichtigste bei der Implementierung der BSP oder der LogP Abstraktion der Routing Algorithmus für h-Relationen. Er muss beliebige Werte für h für die BSP Implementierung unterstützen und nur $\lceil L/(2G)\rceil$ - Relationen für LogP. Es muss überprüft werden, ob die Einschränkung auf Relationen mit kleinerem Grad tatsächlich zu schnelleren Routing Algorithmen führt, das heißt zu kleineren Werten von G und L für LogP verglichen mit den entsprechenden BSP Parametern.
Seien $g^*$ und $\ell^*$ bzw. $L^*$ und $G^*$ die kleinsten Werte der BSP bzw. LogP Parameter, die bei einer Implementierung auf einer gegebenen Maschine möglich sind. Wir sind an der Beziehung zwischen ($g^*$, $\ell^*$) und ($L^*$, $G^*$) interessiert. Unsere Simulationsergebnisse aus dem vorherigen Abschnitt ergeben folgende Schranken:

$$G^* = O(g^*) \text{ und } L^* = O(\ell^*)$$
$$g^* = O(G^*\, S(L^*, G^*, p, h) = O(G^* \log p) \text{ und}$$
$$\ell^* = O(L^*\, S(L^*, G^*, p, h)) = O(L^* \log p)$$

Zusammenfassend: $\Omega(1) <= g^* / G^*, \ell^* / L^* <= O(\log p)$

Jedoch könnten kleinere Parameter Verhältnisse auftreten, wenn die beste direkte Implementierung der zwei Modelle für die gegebene Maschine in Betracht gezogen wird.
Als nächstes beschäftigen wir uns mit Maschinen die sehr genau durch passende Punkt zu Punkt Netzwerke auf Prozessoren mit lokalem Speicher modelliert werden können. Für viele bekannte Verbindungen sind Algorithmen bekannt, die h-Relationen für beliebiges *h* in optimaler Zeit übertragen $\theta(\gamma(p)h + \delta(p))$, wobei $\delta(p)$ für den Netzwerk Durchmesser steht und $\gamma(p) = O(p/b(p))$, wobei b(p) die Bisektionsbandweite ist. Diese Algorithmen bieten eine optimale Wahl für beide LogP und BSP Implementierungen. Tabelle 1 zeigt die Bedeutung für $\gamma(p)$ und $\delta(p)$ für eine Reihe von solchen Netzwerken.
Wir betrachten nun die Voraussetzungen für die LogP Parameter für eines dieser Netzwerke. Die Definition des Modells impliziert, dass jede $\lceil L/(2G)\rceil$ - Relationen in der Zeit L übertragen sein muss. Also muss $\lceil L/(2G)\rceil\, \gamma(p) + \delta(p) <= L$ und daraus folgt $G = \Omega(\gamma(p))$ und $L = \Omega(\delta(p))$.

| Topologie | $\gamma(p)$ | $\delta(p)$ |
|---|---|---|
| d-dim Array d = O(1) | $p^{1/d}$ | $p^{1/d}$ |
| Hypercube (multi-port) | 1 | log p |
| Hypercube (single-port) | log p | log p |
| Butterfly, Shuffle-Exchange | log p | log p |
| Pruned Butterfly Mesh-of-Trees | $\sqrt{p}$ | log p |

Tabelle 11: Bandweite und Latenzzeit Parameter für bekannte Topologien

Für die BSP Parameter wäre es möglich $g = \gamma(p)$ und $\ell = \delta(p)$ zu wählen, wobei die Wahl von $\ell$ daher kommt, dass in allen Topologien Barrierensynchronisation in einer Zeit durchgeführt werden kann, die proportional zu dem Durchmesser ist. Deshalb ist der LogP Parameter für solche Maschinen beschränkt und ist so hoch wie der entsprechende BSP Parameter. Basierend auf den vorangegangenen Überlegungen kommen wir zu folgendem Schluss:
Für die meisten Netzwerk Verbindungen in der Literatur ist $G^* = \theta(g^*)$ und $L^* = \theta(\ell^*)$

**8.Schlussbemerkung**

Wir haben das BSP Modell definiert und begründet, dass es ein vielversprechender Kandidat dafür ist, eine Brücke zu bilden zwischen Software und Hardware im Bereich der parallelen Programmierung. Dabei weist es einen sehr hohen Abstraktionsgrad auf, indem ein Synchronisationsmechanismus vorausgesetzt wird und es sich auf wenige Parameter beschränkt, die für das Design und die Analyse der Algorithmen entscheidend sind, was man auch an der Kürze der Beispiele sieht. Der Nachteil davon ist, dass ein Programm unter Umständen eine längere Laufzeit hat, weil die in einem Superschritt empfangenen Daten erst im nächsten Superschritt verwendet werden können. Weil der Latenzzeit Parameter ursprünglich durch den Bandweiten Parameter abgeschätzt wurde, entstand auch eine gewisse Ungenauigkeit in der Laufzeitvorhersage. Jedoch scheint das Modell dahingehend weiterentwickelt worden zu sein, dass es diesen Wert nun explizit berücksichtigt.
Ebenso wurde das LogP Modell vorgestellt und seine Qualifikation für die Rolle als überbrückendes Modell dargelegt. Das LogP Modell besitzt mehr Parameter und kommt zu komplexeren Algorithmen Entwürfen und Analysen. Diese Komplexität findet sich auch in den Algorithmen aus den Beispielen wieder. LogP wurde von dem Standpunkt aus entwickelt, dass die Vorgängermodelle die Leistungscharakteristiken der Maschinen nicht genau genug wieder geben und unvollständig sind. Eine weiterer Aspekt bei LogP sind die Kapazitätsbedingungen. Sie sollen das Warten der Prozessoren verhindern. Ein Nachteil davon ist allerdings, dass ein LogP Programm dadurch auf einem anderen Computer mit anderen Laufzeit und Bandbreiten Parametern unter Umständen warten muss. Dieses Phänomen verschlechtert die Portabilität von LogP Programmen.
Die Gegenüberstellung beider Modelle zeigt aber, dass sich die beiden Modelle leicht angleichen und simulieren lassen, was darauf hinweist, dass sie nicht sehr verschieden sind. Allerdings ist der Vergleich beider Modelle durchaus noch erweiterbar und die Ergebnisse

nicht endgültig. Die Überkreuzsimulation zeigt die Ähnlichkeit beider Modelle. Aber die Benutzung desselben Routing Algorithmus für die Realisierung der BSP und LogP Abstraktionen ist fraglich. Interessant wäre eine Analyse erst dann, wenn dieser Algorithmus die Kapazitätsbedingungen von LogP auch ausnützen würde. Und eine weitere Idee wäre bei der Analyse Computer zu benutzen, die die erforderliche Barrierensynchronisation für BSP bereitstellen oder die Kapazitätsbedingung von LogP ausnutzen. Mit diesen könnten unterschiedliche Umsetzungen für Algorithmen erstellt werden und dann die sich daraus ergebenden Laufzeiten und der Programmieraufwand beider Modelle verglichen werden.
Bisher jedenfalls ist das BSP Modell überzeugender und hat auch schon mehrere Umsetzungen gefunden, beispielsweise werden auf http://www.bsp-worldwide.org/ aktuelle Arbeiten mit BSP vorgestellt. Außerdem wurde von der Paderborner Universität eine BSP Library erstellt. Die PUB-Library ist eine C-Bibliothek zur Entwicklung paralleler Algorithmen nach dem BSP Modell.

## 9.Literaturverzeichnis

Valiant, L.G., 1990. A Bridging Model for Parallel Computation, Communications of the ACM, 33:8, August 1990, pages 103-111

Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K.E., Santos, E., Subramomian, R., and von Eicken, T. 1993. LogP: Towards a Realistic Model of Parallel Computation, Proceedings of the Fourth ACM SIGPLAN Symposium on Principles an Practice of Parallel Programming, May 1993, pages 1-12.

Gianfranco Bilardi, Kieran T Herley, Andrea Pietracaprina, Geppino Pucci, Paul Spirakis: BSP vs LogP, ACM SPAA 96, Padua Italy 1996.