Karlsruher Institut für Technologie

Mapping von Prozessen auf Prozessoren

Programmierprojekt

Holger Klein
Jasmin Hoffmann

Institut für Theoretische Informatik
Forschungsgruppe Paralleles Rechnen
Fakultät für Informatik
Karlsruher Institut für Technologie

Betreuender Professor:	Prof. Dr. H. Meyerhenke
Betreuender Mitarbeiter:	Dr. Ing. R. Glantz

Bearbeitungszeit: 27. Oktober 2016 – 30. Januar 2017

Inhaltsverzeichnis

1. Einleitung

In jedem Rechennetz mit mehreren Prozessoren stellt sich die Frage nach der Verteilung der Aufgaben auf diese. Dabei sind manche Prozessoren "weiter" von einander weg, das heißt es benötigt mehr Zeit um Daten zwischen ihnen auszutauschen. Meist besteht ein Programm aus mehreren Prozessen, die für ihre Berechnung Informationen untereinander austauschen müssen. Je nach Prozesspaar ist dieses Kommunikationsaufkommen unterschiedlich groß. Somit liegt es nahe Prozesspaare mit hohem Kommunikationsaufkommen Prozessorpaaren mit geringem Weg zueinander zuzuweisen. Gerade in großen Prozessornetzwerken mit voller Auslastung ist dies ein nicht-triviales Optimierungsproblem.

Diese Arbeit befasst sich mit der Implementierung einer möglichen Lösung dieses Problems. Dabei wird eine Heuristik verwandt, die immer das Prozesspaar mit dem höchsten Kommunikationsaufkommen auf das Prozessorpaar mit den geringsten Kommunikationskosten abbildet. Daraufhin werden alle bereits abgebildeten Paare zu einem Superknoten zusammengefasst und in diesem neuen vereinfachten Graph werden daraufhin wieder die zwei Paare mit maximalem Aufkommen und minimalen Kosten gesucht, bis der Graph nur noch aus zwei Knoten besteht.

1.1 Verwendete Sprachen und Tools

Unser Code ist in C++ verfasst und via Cython für Python zur Verfügung gestellt. Unsere Implementierung ist in das Netzwerkanalyse-Toolkit NetworKit eingebettet und bedient sich dort vieler Klassen und Methoden. Unsere Qualitäts- und Laufzeittests wurden mit Google Test erstellt.

1.2 Arbeitsverteilung

Holger Klein hat in dieser Dokumentation das Kapitel Implementierung geschrieben. Von ihm stammen die zwei Graph-Klassen `ProcessGraphQuickConcat` und `ProcessorGraph QuickConcat`. Er hat ebenfalls das ipython-Notebook verfasst. Jasmin Hoffmann hat die übrigen Kapitel in dieser Ausarbeitung verfasst und die Klassen `MapProcessesToProcessors` und `SortedVectorWrapper` geschrieben. Die Tests haben wir gemeinsam durchgeführt.

1.3 Gliederung der Arbeit

Im Hauptteil dieser Arbeit wird zunächst die Problemstellung näher erörtert und die bereits erwähnte Heuristik detailliert geschildert, die das Optimierungsproblem löst. Daraufhin stellen wir unsere Herangehensweise an die Implementierung vor, ihren groben Aufbau und werden sie daraufhin Stück für Stück betrachten. Zuletzt analysieren wir die Performanz unserer Umsetzung anhand von Qualitäts- und Laufzeittests und vergleichen die Ergebnisse mit anderen Implementierungen.

2. Mapping von Prozessen auf Prozessoren

2.1 Problemstellung

Sei

$G_c = (V_c, E_c, \omega_c)$	ein Kommunikationsgraph,
V_c	die Menge der Prozesse,
E_c	die Menge der Kanten, die ein Kommunikationsaufkommen zwischen zwei Prozessen darstellen und
$\omega_c : E_c \mapsto \mathbb{R}^+$	eine Funktion, welche diesem Kommunikationsaufkommen jeweils einen Wert zuweist.

Analog sei

$G_p = (V_p, E_p, \omega_p)$	ein vollständiger Graph, der den Parallelrechner darstellt,
V_p	die Menge der Prozessoren,
E_p	die Menge der Kanten, die durch V_p bereits definiert ist, da der Graph vollständig ist und
$\omega_p : E_p \mapsto \mathbb{R}^+$	eine Funktion, welche die Kommunikationskosten bzw. -zeit zwischen zwei Prozessoren angibt.

Für die Problemstellung wird angenommen, dass $|V_c| = |V_p|$. Gesucht wird eine Bijektion $\pi : V_c \mapsto V_p$ dieser zwei Graphen, welche folgende maximale Dilatation minimiert:

$$\max_{\{u,v\} \in E_c} \omega_c(\{u, v\}) \omega_p(\{\Pi(u), \Pi(v)\})$$

2.2 Heuristik

Die folgende Heuristik soll zur Lösung des beschriebenen Problems implementiert werden:

1. Suche eine Kante $e_c^1 = \{v_c^1, w_c^1\}$ in G_c mit maximalem $\omega_c(\cdot)$-Wert und eine Kante $e_p^1 = \{v_p^1, w_p^1\}$ in G_p mit minimalem $\omega_p(\cdot)$-Wert. Setze $\Pi(v_c^1) := v_p^1$ und $\Pi(w_c^1) := w_p^1$.

2. Iteriere die folgenden Schritte bis G_c und G_p jeweils nur noch zwei Knoten haben. Beginne mit $i = 1$.

   (a) Kontrahiere e_c^i und e_p^i. Es entstehen zwei neue Knoten, v_c^{i+1} bzw. v_p^{i+1}. Wenn parallele Kanten entstehen, fasse Mehrfachkanten zu Einfachkanten zusammen. Der ω_c-Wert der einfachen Kante ist die Summe der ω_c-Werte der zusammengefassten Kanten.

   (b) Finde die Kante $e_c^{i+1} = \{v_c^{i+1}, v_c^*\}$, $e_c^{i+1} \in E_c$, mit maximalem ω_c-Wert.

   (c) Sei M_c die Menge aller Knoten von V_c, die bisher in v_c^{i+1} hineinkontrahiert worden sind. Finde eine Kante $e_p^{i+1} = \{v_p^{i+1}, v_p^*\}$, $e_p^{i+1} \in E_p$, sodass

   $$\sum_{v_c \in M_c} \omega_c(\{v_c, v_c^*\})\omega_p(\{\Pi(v_c), v_p^*\})$$

   minimal ist. Setze $\Pi(v_c^*) := v_p^*$.

2.3 Implementierung

Es ist möglich, die beschriebene Heuristik mit der von `Networkit` bereitgestellten Graphen-Datenstruktur zu implementieren. Da diese jedoch eine Allzweckstruktur mit viel Funktionalität ist, liegt es nahe, dass spezialisierte Datenstrukturen die benötigten Operationen schneller ausführen können. Wir entschieden uns daher dafür zwei spezielle Graphen-Datenstrukturen zu implementieren. Diese haben sehr eingeschränkte Funktionalität und sind daher sehr effizient.

Verwendete Klassen

- `MapProcessesToProcessors`: Dies ist die Klasse welche von `Algorithm` erbt und die Ausführung des Algorithmus initiiert und kontrolliert.
- `ProcessGraphQuickConcat` : Diese Klasse speichert während des Algorithmus den Zustand des Prozessgraphen. Sie ermöglicht eine sehr schnelle Konkatenation der momentan schwersten Kante.
- `ProcessorGraphQuickConcat`: Diese Klasse speichert während des Algorithmus den Zustand des Prozessorgraphen. Sie ermöglicht ein schnelles errechnen der Summe in Schritt c) in Schritt 2 der Heuristik.
- `SortedVectorWrapper`: Dies ist eine Hilfsklasse welche einen `vector` kapselt und diesen stets sortiert hält. Sie ermöglicht das Auffinden von Elementen in $\mathcal{O}(logn)$ (wobei n die Größe des `vector`s ist) durch binäre Suche.
- `TupleWithFirstComparator` Dies ist eine sehr kleine Helfer-Klasse. Sie kapselt ein Tupel und ermöglicht das Vergleichen dieses Tupels mit einer Variablen vom Typ des ersten Elements. Dies erleichtert das Verwenden von Standard-Algorithmen wie binärer Suche.

Wir stellen in unserer Klasse `MapProcessesToProcessors` drei öffentliche Methoden zur Verfügung:

- `run`: Hiermit kann analog zur Klasse `Algorithm`, von der wir erben, unsere Heuristik auf die zwei privaten Graphen processG und processorG angewandt werden.
- `printMapping`: Dies gibt das gesamte Mapping auf `cout` aus.
- `getMappedNode(node n)`: Gibt aus, auf welchen Knoten n gemappt wurde.

2.3.1 Die Graphen-Datenstrukturen

Wie bereits erwähnt, haben wir für beide Graphen jeweils spezialisierte Datenstrukturen implementiert, anstatt die von `Networkit` bereitgestellte zu verwenden. Dies hat folgende Gründe:

- Da wir die überreichten Graphen nicht verändern wollten, mussten wir ihre Information so oder so kopieren. Das Erstellen unserer Graphen ist daher vernachlässigbarer Overhead.
- Es ist für alle Knoten nötig, ihre ausgehenden Kantengewichte zweimal zu speichern: Einmal vor und nach dem Konkatenieren. Dies ist mit der `Graph`-Klasse nicht ohne weiteres möglich.
- Das Konkatenieren zweier Knoten lässt sich mit der `Graph`-Klasse nicht elegant implementieren. Unsere Klassen hingegen implementieren dies als eine Vektoraddition. Die ursprünglichen Kantengewichte des konkatenierten Knotens werden dabei automatisch gespeichert.

Die Konstruktoren beider Klassen sind dabei ähnlich. Beide erwarten eine konstante Referenz auf einen `Graph`en. Beide Klassen verfügen über eine Variable vom Typ `SortedVectorWrapper<node>`, in welcher alle Knoten des übergebenen Graphen gespeichert werden. Sortiert werden sie dabei nach dem Wert des `uint64_t`, welcher einen Knoten repräsentiert. Später wird zur Identifizierung der Knoten hauptsächlich der Index in diesem Vektor verwendet. Danach speichern beide die Kantengewichte in einer Matrix. Da der Prozessorgraph komplett ist, ist dies sehr unkompliziert. Hier werden die Kantengewichte in einem `std::vector<std::vector<edgeweight>>` hinterlegt. Der Prozessgraph hingegen speichert die Kantengewichte in einem `std::vector<SortedVectorWrapper<IndexAndWeight>>` namens `indexEdges`. `IndexAndWeight` ist ein `typedef` vom Typ `TupleWithFirstComparator<size_t, edgeweight>`. Es habe zum Beispiel Knoten u eine Kante mit Gewicht 4 zu Knoten v. Angenommen, u hat Index 5 und v Index 10. Dann hätte das fünfte Element von `indexEdges` den Eintrag `TupleWithFirstComparator<10, 10>`. Im Konstruktor beider Klassen wird auch gespeichert, welche Kante die schwerste beziehungsweise leichteste ist. Einer der Knoten dieser Kante wird in Zukunft als Konkatenationsknoten dienen. In diesen Knoten werden alle weiteren Knoten hineinkonkateniert. Im Prozessgraphen gibt es für die Kanten dieses Knotens einen eigenen `vector` namens `concatNodeWeightVector`.

2.3.1.1 Konkatenation

In jeder Iteration des Algorithmus wird die schwerste Kante des Prozessgraphen konkateniert. Daher ist es sehr wichtig, dass diese Aktion effizient implementiert ist. Durch unsere interne Speicherung der Kantengewichte als Matrix lässt sich dies erreichen. Die eigentliche

Konkatenation ist lediglich eine Vektoraddition. Der erste Eintrag jedes `IndexAndWeight` ist dabei der Index in `concatNodeWeightVector`, auf welchen der entsprechende zweite Eintrag aufaddiert werden muss. Welche Knoten bereits konkateniert wurden, wird in einem `vector<bool>` gespeichert. Während der Konkatenation wird auch die neue schwerste Kante gefunden und gespeichert. All dies ist in der Methode `concatNodeWithHeaviestNeighbour IndexVersion` implementiert. In jeder Iteration des Algorithmus benötigt man die Menge aller Kanten, welche von dem nun schwersten Nachbarn des `concatNodeWeightVector` zu allen bereits konkatenierten Knoten ausgeht. Diese wird nach jeder Konkatenation in `buildConcatedWeightsIndexVersion` gebildet. Was die Veränderung der Kantengewichte betrifft, hat die Konkatenation im Prozessorgraphen auf den Algorithmus keinen Einfluss. Daher wird hier lediglich der entsprechende Eintrag in `concated` auf `true` gesetzt.

2.3.1.2 Minimieren der Summe in Schritt 2c

Das Auffinden des Knotens, welcher

$$\sum_{v_c \in M_c} \omega_c(\{v_c, v_c^*\})\omega_p(\{\Pi(v_c), v_p^*\})$$

minimiert, geschieht in der Methode `getNodeWhichMinimizesSum` in dem Prozessorgraphen. Diese erwartet als Eingabe die in `buildConcatedWeightsIndexVersion` zusammengestellten Kantengewichte sowie die Knoten, auf welche diese konkatenierten Knoten abgebildet werden. Diese findet zunächst für alle Knoten $\Pi(v_c)$ deren Index. Dann sucht sie unter allen noch nicht konkatenierten Knoten denjenigen, welcher die Summe minimiert und gibt diesen aus.

2.3.2 Ablauf des Algorithmus

Der Algorithmus startet, sobald in der initiierten `MappingProcessesToProcessors`-Klasse die `run`-Methode aufgerufen wird. Diese bildet zunächst die schwerste Kante des Prozessgraphen auf die leichteste Kante des Prozessorgraphen ab. Danach werden diese Kanten konkateniert.

Danach wird die while-Schleife des Algorithmus durchlaufen, bis der Prozessgraph nur noch zwei Knoten enthält. In der while-Schleife passiert jedes Mal folgendes:

- Erfrage den momentan schwersten Nachbarn des Konkatenationsgraphen vom Prozessgraphen
- Bekomme die Liste aller Kantengewichte, welche dieser mit allen bereits konkatenierten Knoten hat
- Finde alle Knoten, auf welche die bereits konkatenierten Knoten abgebildet wurden
- Übergebe diese beiden Daten an `getNodeWhichMinimizesSum`
- Bilde den momentan schwersten Nachbarn auf das Ergebnis von `getNodeWhich MinimizesSum` ab

Sobald pro Graph nur noch zwei Knoten übrig sind, werden diese aufeinander abgebildet.

2.4 Qualitätstests

Die Korrektheit unserer Implementierung haben wir an verschiedenen Graphen mit $3-5$ Knoten geprüft.

Bei der Qualitätsprüfung haben wir für den Prozessorgraph einen zweidimensionalen Mesh und Torus mit jeweils 1024 Knoten verwandt. Die Prozessgraphen wurden analog zu unserer Vergleichsquelle [GMN14] gewählt: Die acht größten Graphen aus Walshaws Graphpartitionierungsarchiv [SWC04] sowie 13 soziale Netzwerke aus [SSS15]. Unsere Tests wurden auf einem 64-bit-System mit Intel®Core™ i5-2520M Vierkernprozessor bei 2.50GHz und 8GB DDR3 RAM durchgeführt. Die Vergleichswerte stammen von einer Workstation mit zwei Intel®Core™ i7-2600K Vierkernprozessoren bei 3.40GHz.

Gemessen wurde die zur Berechnung gebrauchte Zeit und die maximale Dilatation (siehe 2.1). Die Dilatation wurde wie in den Vergleichswerten in Verhältnis zu einem 1-zu-1-Mapping gesetzt, bei dem Knoten i im Prozessgraph direkt auf Knoten mit der Nummer i im Prozessorgraph abgebildet wird, so ergibt sich D. Über den mehrfachen Messungen der Zeit wurde das arithmetische Mittel t gebildet. Über den Ergebnissen aller Testgraphen wurden jeweils die geometrischen Mittel t_{gm} und D_{gm} berechnet. Dabei ist MapPtoP unser Algorithmus, jene darüber stammen aus der Vergleichsquelle. Im unteren Teil der Tabelle finden sich die Ergebnisse unseres Algorithmus auf den einzelnen Graphen.

Abgesehen von RCM braucht unsere Implementierung deutlich länger als die anderen, allerdings ist hier zu beachten, dass unser Testsystem nicht so leistungsstark war wie das der Vergleichswerte. Hinsichtlich der maximalen Dilatation erhalten wir sehr gute Werte. Da wir die maximale Dilatation minimieren möchten, ist ein kleinerer Wert hier besser. Bei Walshaws Graphen liegt die Heuristik im besseren Mittelfeld, bei den sozialen Netzwerken ist sie sogar die beste.

Anschließend haben wir in unserer Heuristik im letzten Schritt in

$$\sum_{v_c \in M_c} \omega_c(\{v_c, v_c^*\})\omega_p(\{\Pi(v_c), v_p^*\})$$

die Summe durch *max* ersetzt und die Auswirkungen beobachtet. Der Unterschied ist unter dt und dD ablesbar, dabei bedeutet ein negativer Wert, dass *max* schneller war oder eine niedrigere maximale Dilatation hatte. In etwa der Hälfte der Fälle hat sich die Dilatation verbessert, besonders beim Torus. In fast allen Fällen hat sich die Laufzeit verbessert.

	Grid				Torus			
Algorithmus	t_{gm}	D_{gm}			t_{gm}	D_{gm}		
Random	0.110	2.169			0.111	1.473		
RCM	0.239	1.433			0.249	1.512		
DRB	221.6	0.690			217.0	0.864		
GreedyAll	18.53	1.565			18.07	1.715		
GreedyMin	3.867	1.320			4.068	1.216		
GreedyAllC	18.05	0.615			18.08	0.684		
GreedyMinC	16.71	0.837			17.68	0.815		
MapPtoP	111.7	0.859			180.4	0.785		
Graph	t	D	dt	dD	t	D	dt	dD
144	615.6	1.449	-529.7	-0.585	620.4	1.029	-540.9	0.024
598a	78.20	1.325	-24.69	0.498	80.5	0.730	-20.28	0.030
auto	80.52	0.625	-22.75	-0.104	261.0	1.038	-200.0	-0.350
fe_ocean	98.19	0.738	-12.02	0.085	114.7	0.978	-27.10	-0.410
fe_rotor	67.73	0.611	-22.80	0.826	71.41	0.658	-18.25	-0.065
fe_tooth	96.05	1.242	-4.927	-0.376	109.7	0.768	-37.92	-0.364
m14b	93.65	0.535	-31.29	0.587	713.3	0.617	-635.1	-0.143
wave	104.6	0.822	-23.65	-0.303	134.2	0.608	-55.86	-0.075

Tabelle 2.1 Prozessgraphen aus Walshaws Graphpartitionierungsarchiv

	Grid				Torus			
Algorithmus	t_{gm}	D_{gm}			t_{gm}	D_{gm}		
Random	0.112	0.812			0.110	0.644		
RCM	0.504	0.871			0.513	0.762		
DRB	445.1	0.636			442.1	0.764		
GreedyAll	75.82	0.875			75.93	0.973		
GreedyMin	4.141	0.849			4.457	0.958		
GreedyAllC	99.59	0.631			99.57	0.755		
GreedyMinC	88.87	0.668			89.39	0.760		
MapPtoP	296.0	0.486			270.3	0.458		
Graph	t	D	dt	dD	t	D	dt	dD
PGPgiantcompo	75.54	0.465	-12.45	0	70.38	0.601	-15.08	0
email-EuAll	128.4	0.365	-16.17	-0.043	126.1	0.294	-24.81	-0.056
soc-Slashdot0902	517.3	0.552	27.88	-0.007	447.4	0.371	56.36	0
loc-brightkite_edges	769.3	0.503	-535.5	0	276.6	0.838	-79.69	0
coAuthorsCiteseer	196.4	0.311	-35.54	-0.035	229.8	0.437	-59.33	-0.107
wiki-Talk	557.0	0.658	-114.5	0.036	502.3	0.827	-51.05	0
citationCiteseer	262.7	0.441	-3.675	-0.150	294.8	0.287	-57.14	-0.073
coAuthorsDBLP	366.0	0.240	-62.30	0.107	372.5	0.253	-110.5	-0.112
web-Google	176.3	0.395	-48.06	0.043	186.7	0.371	-64.25	0
as-skitter	368.3	0.540	-44.32	0.107	297.3	0.350	28.72	-0.016
as-22july06	734.6	0.775	-624.1	0.025	710.3	0.676	-615.9	-0.074
p2p-Gnutella04	172.4	0.644	-46.65	0.511	176.0	0.583	-22.55	-0.058
loc-gowalla	400.6	0.775	-14.25	-0.118	426.0	0.533	-91.70	-0.086

Tabelle 2.2 soziale Netzwerke als Prozessgraph

3. Zusammenfassung

Es hat sich gezeigt, dass trotz des sehr umfangreichen Methoden-Angebots der NetworKit-Graph-Klasse sich in unserem Fall eine weitestgehende Neuimplementierung von gleich zwei Graph-Klassen angeboten hat, da wir viele Summen benötigt haben, die in jedem Schleifendurchlauf über den Kantengewichten lediglich ergänzt und nicht neu gebildet wurden. Ebenfalls war bei einem vollständigen Graphen der Zugriff über eine Matrix naheliegend und hat den Zugriff vereinfacht und beschleunigt. So haben wir einiges an überflüssigem Speicher- und Laufzeitverbrauch gespart.

Bei der Qualitätsanalyse hat sich gezeigt, dass unsere Implementierung auf unserem Testsystem deutlich langsamer war als die Vergleichsimplementierungen, allerdings ist dieses Ergebnis verfälscht durch die unterschiedliche Leistungsstärke der Testsysteme. Die Qualität des Mappings war hingegen eines der Besseren und teilweise sogar das Beste unter den Algorithmen.

Literaturverzeichnis

[GMN14] Glantz, R., H. Meyerhenke und A. Noe: *Algorithms for Mapping Parallel Processes onto Grid and Torus Architectures.* CoRR, abs/1411.0921, 2014.

[SSS15] Safro, Ilya, Peter Sanders und Christian Schulz: *Advanced Coarsening Schemes for Graph Partitioning.* J. Exp. Algorithmics, 19:2.2:1–2.2:24, Januar 2015.

[SWC04] Soper, A.J., C. Walshaw und M. Cross: *A Combined Evolutionary Search and Multilevel Optimisation Approach to Graph-Partitioning.* Journal of Global Optimization, 29(2):225–241, 2004.