Parallele Programmierung

Sommersemester 2012

Ameisensimulation von Christian Peter (0peter@informatik.uni-hamburg.de) und Dominik Rupp (0rupp@informatik.uni-hamburg.de)

Inhalt

EINLEITUNG

Aufgabe im Praktikum „Parallele Programmierung“

Für das Praktikum „Parallele Programmierung“ der Universität Hamburg soll, ein Programmierprojekt realisiert werden, welche die Vorteile von Multicore Systemen nutzt.

Als Programmiersprache soll dazu C oder alternativ FORTRAN verwendet werden. Zur Parallelsierung des Programms sollen die Bibliotheken OpenMP oder bevorzugt MPI eingesetzt werden. MPI (Message Passing Interface) ist eine Programmierschnittstelle, zum Austausch von Nachrichten in verteilten System bzw. zum Austausch von Nachrichten zwischen Prozessen.

Es soll vorerst ein serielles Programm geschrieben werden, welches eine komplexe Aufgabe löst. Dies können z.B. Lösungsalgorithmen für Spiele sein (z.B. bester Zug beim Schach) oder Simulationen aus der Natur (z.B. eine Windsimulation).

Erste Überlegungen zur Ameisensimulation

Die Idee zur Ameisensimulation bauen teilweise auf dem AntMe![1] Projekt auf, welches ein spielendes Erlernen der Programmiersprache C# erreichen will. Bei AntMe! erkunden Ameisen, von einem Bau aus die Umgebung, welche abzubauende Ressourcen bereit stellt. Dem Spieler (Programmierer) soll es gelingen, möglichst effektive Bewegungsmuster zu entwickeln, um die Ressourcen aufzufinden und andere Ameisen über sogenannte Markierungen, ebenfalls den Weg zu den Ressourcen zu weisen.

Unsere Ameisensimulation mit dem Namen AntSim, soll ebenfalls Ameisen erzeugen, die auf einer Karte nach Ressourcen suchen und diese dann abbauen. Bei entsprechender Wahl der Umgebungsvariablen (wie z.B. Anzahl der Ameisen, Anzahl der Simulationsschritte), müsste die serielle Ausführung der Simulation, mehrere Sekunden bis zu einigen Minuten in Anspruch nehmen. Eine geeignete Wahl der Umgebungsvariablen, wäre eine Kartengröße von ca. 1.000 mal 1.000 Feldern, so kann man jeden Simulationsschritt als Bild exportieren. Dies hätte zur Folge, dass es 1.000.000 betretbare Felder gibt, sodass man z.B. mit 200.000 Ameisen arbeiten könnte, wenn man bis zu 20% der Karte pro Schritt, nutzen möchte. Geht man davon aus, die Berechnung von 1.000 Ameisen mindestens eine Millisekunde dauert, würde bei einer Simulationslänge von 1.000 Schritten, eine Laufzeit von 200 Sekunden, bzw. 3 Minuten und 20 Sekunden folgen.

[1] http://www.antme.net

Dies wäre eine Größenordnung, bei der sich eine Parallelisierung lohnen könnte, auch weil man die Laufzeit, anhand der Simulationsschritte beliebig erhöhen kann.

Weitere mögliche Features

Die oben beschriebene Funktionalität, stellt nur eine Mindestanforderung für unser Programm dar und lässt sich noch durch viele weitere Ideen erweitern. So war noch nicht von Anfang an klar, wie die Kommunikation unter den Ameisen stattfinden soll. Die einfachste Möglichkeit ist, dass alle Ameisen ein gemeinsames „Gehirn“ besitzen und Informationen, wo sich z.B. eine Ressource befindet „per Gedankenübertragung“ austauschen können. Weiter wäre es möglich, dass der Ameisenbau diese Informationen abspeichern und an in der nähe befindliche Ameisen, weitergeben kann. Orientiert man sich an der Natur, so stellt man fest, dass die Ameisen Informationen über Duftstoffe (sog. Pheromone) austauschen können. In AntMe! wären das die Markierungen, die auf der Karte abgelegt werden können und nach kurzer Zeit wieder verschwinden. Eine ähnliche Funktionsweise, wäre für die Simulation wünschenswert, da dies eine modellhafte Abbildung der Natur darstellen würde.

Weiterhin haben wir Überlegungen zur Sichtweite der Ameisen getätigt. Denkbar ist es, dass die Ameise nur weiß, was sich auf dem eigenen Feld befindet, oder dass sie die angrenzenden Felder „sehen“ kann. Zudem könnte man sich vorstellen, dass es Ameisen gibt, die z.B. zwei Felder weit sehen können oder andere Ameisen die auch über eine größere Distanz die Umgebung wahrnehmen können.

Diese Unterschiedlichen Typen von Ameisen nennt man in der Natur Kasten. Eine Ameise mit einer hohen Sichtweite könnte z.B. der Kaste Späher angehören, wohingegen die anderen Ameisen Arbeiter sind. Ein solches Verhalten ließe sich leicht implementieren, wenn man einen bestimmten Prozentanteil der Ameisen mit einer bestimmten Kaste erzeugt.

```
20      int percent = rand() % 100;
21      if(percent < 80)
22          newAnt.Caste = 0;
23      else
24          newAnt.Caste = 1;
25      if(newAnt.Caste == 0)       // Arbeiter
26          newAnt.ViewDist = 2;
27      else if(newAnt.Caste == 1) // Späher
28          newAnt.ViewDist = 4;
29
30      appendAnt(list, newAnt);
```

Alle Ameisen sollen am Bau starten und von dort aus ein bestimmtes Bewegungsmuster verfolgen. Findet eine Ameise eine Ressource, so soll sie diese zurück zum Bau bringen. Orientiert man sich auch hier an der Natur so kann man davon ausgehen, dass die Ameise den Weg zum Bau kennt, da sie diesen über den Sonnenstand ermitteln kann.[2]

Denkbare Erweiterungen wären außerdem auf der Karte befindliche Feinde, wie es auch in AntMe! durch die Wanzen realisiert ist. Hier muss man auch dafür sorgen, dass sich mehrere Ameisen gemeinsam gegen eine Wanze stellen, um diese besiegen zu können. Zudem könnte man sich vorstellen, dass es mehrere (befreundete) Ameisenbaus gibt oder auch verschiedene Völker die eventuell auch mit Gewalt, eigene Interessen verfolgen.

Realisierung

Der serielle Code

Der noch nicht parallelisierte Code, setzt sich aus vielen Funktionen, globalen Variablen, Strukturen und Konstanten zusammen. Die wichtigsten, sollen im Folgenden beschrieben werden.

Wichtige Konstanten sind die Kartengröße, mit der die Anzahl der Zeilen bzw. Spalten der Karte festgelegt werden können, die Anzahl der zu erzeugenden Ameisen und die Simulationslänge.

Neben den Strukturen, welche lediglich verkettete Listen realisieren, gibt es eine Struktur „Ant“, die die wesentlichen Merkmale einer Ameise hält, die Struktur „Bau“ für den Ameisenbau und eine Struktur Vector2, welche einen zweidimensionalen Vektor bereitstellt, welcher Zeilen und Spalteninformationen speichert.

Die Karte, bestehend aus einem zweidimensionalen Array, eine Liste von Ameisen, ein Bau, sowie eine Zählvariable, welche angibt, wie viele Ameisen zum momentanen Zeitpunkt erzeugt wurden, werden global gehalten.

Die main Funktion dient als eine Art „Game Loop“ indem hier, nach einigen Initialisierungen, die Simulationsschritte iteriert werden. Innerhalb eines jeden Schritts wird für alle bereits erzeugten Ameisen, die Bewegung auf ein anderes Feld der Karte berechnet. Außerdem werden in der main Methode, neue Ameisen erzeugt, sodass sich nach der Hälfte der

[2] Aus http://de.wikipedia.org/wiki/Ameisen#Sinnesorgane:
Außerdem wurde die Fähigkeit zur Analyse linear polarisierten Lichts nachgewiesen, wodurch die Tiere auch bei teilbedecktem Himmel den Sonnenstand ermitteln können. Diese Fähigkeit dient vermutlich der Orientierung im Gelände (nachgewiesen bei der Wüstenameise *Cataglyphis*[13]).

Simulationsschritte, alle Ameisen auf der Karte befinden. Bei 1.000 Ameisen in 1.000 Runden, wären das also zwei Ameisen pro Runde bzw. je zwei Ameisen von Runde 1 bis 500.

Zu Beginn wird die Position des Baus auf die Mitte der Karte festgelegt. Die Anzahl der Ressourcen ist abhängig von der gewählten Kartengröße. Sie berechnet sich aus Anzahl der Felder geteilt durch 300. Bei einer Kartengröße von 25*12 mit 1200 Felder ergeben sich also vier Ressourcenfelder, die jeweils eine Größe von 2x3 auf der Karte bedecken.

Zu sehen in einer einfachen Visualisierung auf der Konsole:

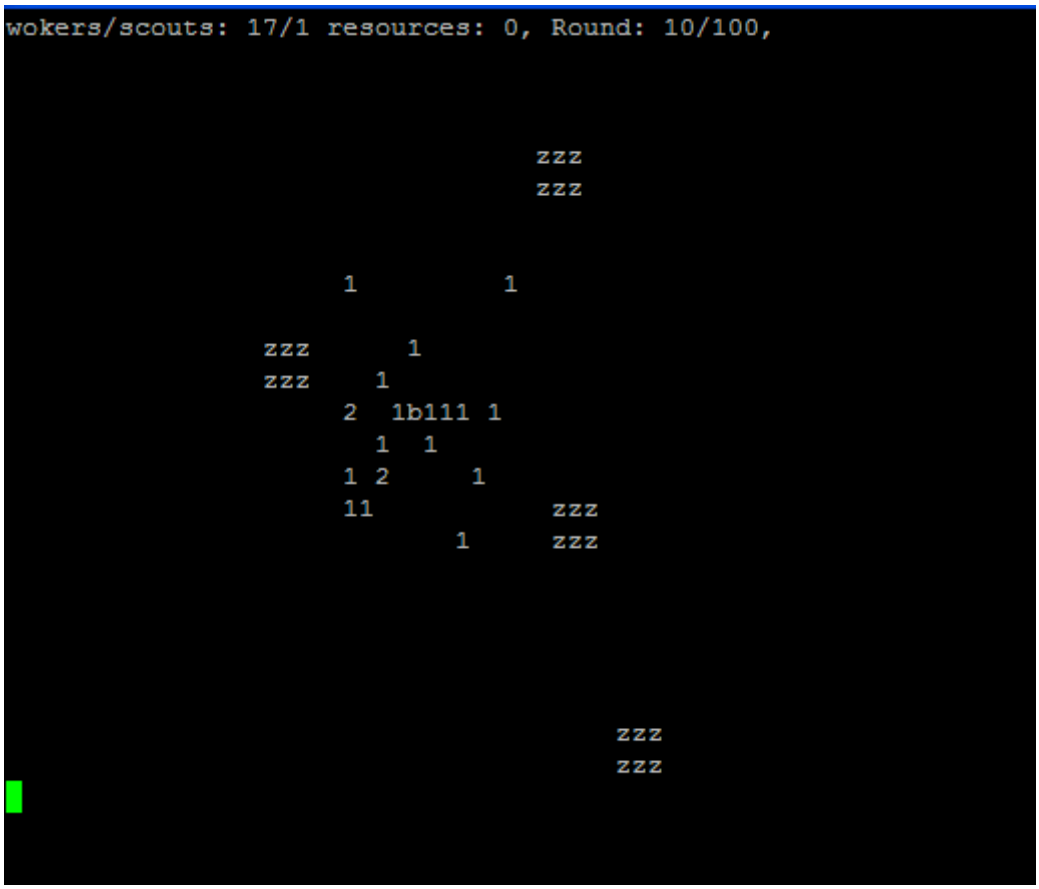


Abbildung 1 Ausgabe der AntSim auf der Konsole. Das kleine b beschreibt den Ameisenbau. Die Blöcke von kleinen z, sind die Ressourcen (gemeint ist z wie Zucker). Die Zahlen, besagen wie viele Ameisen sich an dieser Position befinden.

Hat eine Ameise keine Information (mehr) darüber wohin sie sich als nächstes Bewegung soll, wird in der Funktion moveRnd(..) zufällig ein neuer Vektor bestimmt der die Bewegung für die nächsten Runden festlegt. Dabei werden die Eigenschaften Row und Col der Struktur MovePos gesetzt. Wird zum Beispiel für Row = -3 und Col = 2 gesetzt, bedeutet das, dass die Ameise sich in den nächsten fünf Runden um drei Felder nach unten und zwei Felder nach rechts bewegt.

Dabei kann jede Ameise pro Runde nur einen Schritt weit gehen. Sind die fünf Runden verstrichen, wird von moveRnd(..) wieder ein neues Ziel berechnet.

Stößt die Ameise in der Zwischenzeit auf eine Ressource, so kann sie diese je nach Sichtweite über wenige Felder hinweg wahrnehmen, läuft darauf zu und baut diese ab. Es wird das Flag HasResource bei der Ameise gesetzt und der Weg zurück zum Bau berechnet, den die Ameise – wie wir annehmen – kennt. Auf dem Rückweg versprüht die Ameise die sogenannten Markierungen, die es den anderen Ameisen ermöglichen soll, ebenfalls zu dieser Ressource zu finden. Diese Wegweiser haben eine begrenzte Lebensdauer und halten z.B. Weglänge * Faktor lange, können aber immer wieder erneuert werden. Jede Runde wird die Restdauer aller Markierungen dekrementiert, bis sie bei Lebensdauer = 0 entfernt wird. Am Bau angekommen wird ein Counter inkrementiert, der die Anzahl der abgebauten Ressourcen zählt.

Andere Ameisen nehmen Markierungen in ihrer Reichweite nun wahr und bewegen sich anhand dieser zur Ressource. Es bildet sich eine Ameisenstraße und viele Arbeiter beteiligen sich nun beim Abbau der Ressourcen. Diese haben allerdings nur eine begrenzte Kapazität, sodass diese irgendwann erschöpft sind. Die Ameisen laufen weiterhin auf der Straße entlang, die Markierungen lösen sich nach und nach auf, da keine Ressourcen mehr gefunden werden und das alte Bewegungsmuster wird weitergeführt.

Die Ameisen, Ressourcen und Markierungen werden dabei in einfach verketteten Listen gehalten. Zusätzlich werden diese Informationen aber auch in der Karte gehalten. Muss z.B. geprüft werden ob eine Ameise eine Markierung sieht, ist es schneller und einfacher dies über die Karte zu ermitteln. Um das Problem zu umgehen, dass Informationen auf der Karte verloren gehen, da dort nur jeweils ein Integer pro Feld gespeichert wird, aber auch um z.B. einfach alle Markierungen dekrementieren zu können ist es sinnvoll alle Informationen als Liste zu halten.

Überlegungen zur Parallelisierung

Es stand fest, dass wir die Parallelisierung den Projektvorgaben entsprechend mit MPI realisieren. Die grundsätzlichste Frage stellt sich, ob man eher die Karte in kleine Abschnitte segmentieren sollte oder auf einer Liste von Ameisen arbeitet und diese an die einzelnen Threads verteilt.

Bei der Segmentierung der Karte könnte es vorteilhaft sein, dass eine sehr große Karte nicht in einem sehr großen Array im Speicher gehalten werden muss, sondern dass jeder Prozess nur seinen Ausschnitt im Speicher hält. Es stellt sich aber das Problem, wie man damit umgeht,

dass mehrere Prozesse möglicherweise versuchen in das gleiche Arrayfeld zu schreiben und damit Ameisen „verloren gehen“. Dies wollten wir lösen, indem wir zusätzlich immer die Liste von Ameisen im Speicher halten, wo auch die aktuelle Position gespeichert ist.

Zusätzlich erschien die Segmentierung anhand der Karte aber nicht so sinnvoll, da die Ameisen, sobald sich eine Ameisenstraße bildet nur von wenigen Threads bearbeitet werden können.

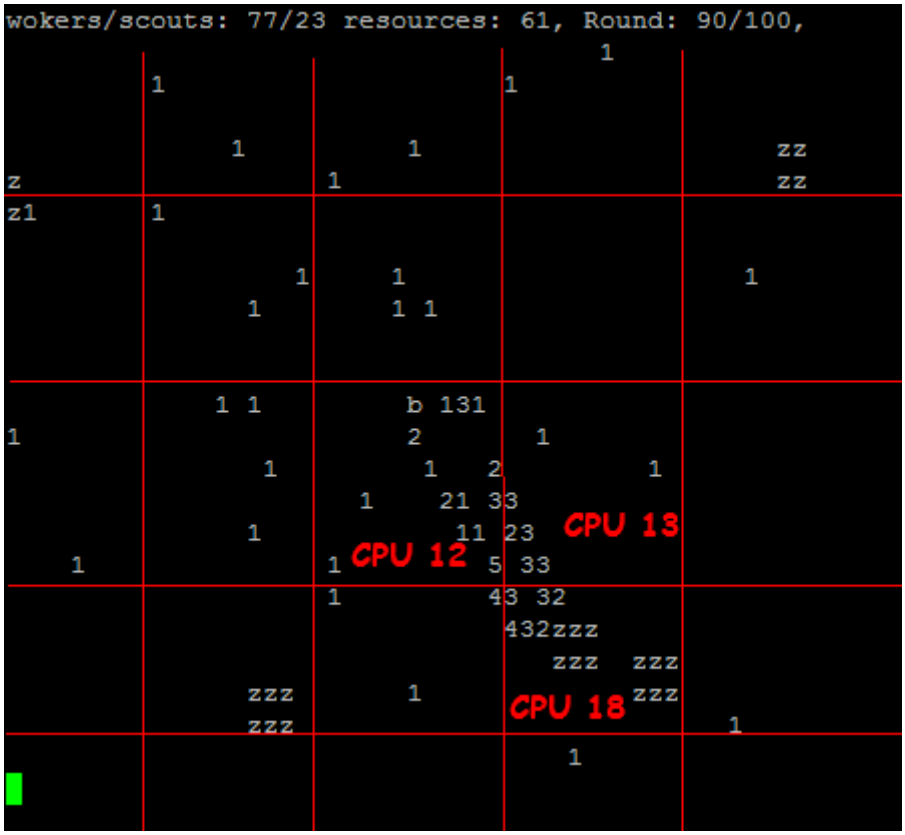


Abbildung 2 Beispielhafte Segmentierung der Karte mit 25 CPUs. Hauptsächlich CPU12, 13 und 18 wären aktiv. CPU 0 und 8 dagegen hätten gar keine Arbeit.

Stattdessen wäre bei einer Aufteilung anhand der Ameisenliste eine Gleichverteilung auf alle Threads möglich. Allerdings muss man die Informationen welcher Thread welche Ameise „kennt“ verwalten, was sich auch als nicht einfach herausstellen würde.

Erster Versuch der Parallelisierung

Nachdem die sequentielle Version nun lauffähig war, machten wir eine Kopie des Projekts und banden die MPI Header ein. Die erste parallele Version sollte mit nur zwei Threads arbeiten, wobei ein „master thread“ die Aufgaben verteilt und ein „worker thread“ in einer Schleife auf Aufgaben wartet und sich auf Befehl beendet. Globale Variablen wurden nun weitestgehend lokal gehalten, da jeder Thread seinen eigenen globalen Speicherbereich hat und diese dadurch entsprechend über MPI Nachrichten synchronisiert werden müssen.

Der Master verteilt pro Runde die Aufgaben Ameisen zu erzeugen und Ameisen zu bewegen. Der Slave führt diese Aufgaben entsprechend aus und fragt beim Master nach Informationen. Z.B. ob sich auf der aktuellen Position eine Ressource oder Markierung befindet oder in Sichtweite ist. Der Slave muss außerdem den Master benachrichtigen, wenn eine Resource abgebaut wurde oder Markierungen erzeugt wurden.

Vom Master werden also pro Runde mindestens so viele Nachrichten an den Slave geschickt wie sich Ameisen auf dem Feld befinden. Dazu kommen die Nachrichten die notwendig sind um in der ersten Hälfte der Simulation die Ameisen zu erzeugen.

Der Slave hat pro Ameise mindestens drei Anfragen, nämlich ob sich eine Resource auf dem aktuellen Feld befindet, ob eine Resource in Sichtweite ist und ob eine Markierung in Sichtweite ist. Außerdem muss nach Auswertung, dieser Informationen pro Ameise die neu ermittelte Position an den Server gesendet werden.

Bei 50.000 Ameisen, 12 Threads und 1000 Simulationsdurchgängen bereits 50.062.000 Kommunikationsoperationen

Als die Implementierung mit zwei Threads ordnungsgemäß funktionierte, war es zur Lösung mit beliebig vielen Threads kein großer Schritt mehr. Allerdings wird der Master jetzt von mehreren Threads angefragt, was die Skalierbarkeit der Anwendung deutlich beeinträchtigen müsste.

Nach den ersten Tests mussten wir feststellen, dass die parallele Ausführung um mehr als Faktor 1.000 langsamer war als die sequentielle Version. Es war schnell klar, dass zu viele MPI Send und Receive benötigt werden. Jeder Send und Receive verbraucht mehrere 10.000 CPU Zyklen, in welchen etliche einfache Operationen getätigt werden könnten. Der Profiler gprof bestätigte diese Vermutung. Es stellte sich heraus, dass ca. 97% der CPU-Zeit durch MPI Aufrufe verbraucht werden.

Zusätzlich skalierte die Anwendung so gut wie garnicht, sobald mehr als zwei Worker im Einsatz waren. Ein Test mit einer Kartengröße von 240x500, 500 Ameisen und einer Simulationlänge von 1.000 Runden zeigte auf, dass sich das Programm im Mittel 11 Sekunden bei einem Worker, 8 Sekunden bei zwei Workern brauchte. Eine weitere Erhöhung der Worker Threads brachte bereits keine Zeitersparnis mehr.

Es war klar, dass möglichst viele MPI Aufrufe eingespart werden mussten und die Worker selbstständiger Arbeiten mussten.

Die parallele Implementierung

Um die Schwächen der ersten Implementierung auszubessern mussten die Worker Threads mehr Informationen über den aktuellen Kartenstatus bekommen. Bevor jeder Worker beginnt Ameisenpostionen zu berechnen, bekommt er nun vom Master vor jedem Durchlauf die aktuellen Informationen über Ressourcen und Markierungen.

```
154  void sendResourcesToSlaves(struct VectorList *resourceList, int tasks)
155  {
156      int i;
157      for(i=2; i < tasks; i++)
158      {
159          struct VectorList *currentResource = resourceList;
160          while(currentResource->Next != NULL)
161          {
162              currentResource = currentResource->Next;
163              int resPos[2] = { currentResource->Value.Row, currentResource->Value.Col };
164              if(debugOutput && 0)
165                  printf("send resource at pos %d %d to task %d\n", resPos[0], resPos[1], i);
166              MPI_Send(resPos, 2, MPI_INT, i, MPI_ResourcePosition, MPI_COMM_WORLD);
167          }
168      }
169  }
```

Dies sollte schon einen erheblichen Performancegewinn darstellen, da die Slaves nun nicht mehr für jede Ameise beim Master anfragen müssen ob sich Resourcen oder Markierungen in Reichweite befinden.

Allerdings stellt sich nun die Herausforderung, dass ein Worker eine Resource abbauen könnte, die eigentlich schon erschöpft ist. Daher ist für den Abbau eine Resource immernoch eine Anfrage beim Master notwendig. Antwortet der Master, dass eine Resource nicht abgebaut werden kann, so entfernt der Slave die Resource auch von dessen interner Liste.

Eine zweite Nachricht ist nun noch notwendig um den Master zu benachrichtigen, dass eine Resource am Bau angekommen ist.

Am teuersten wirken sich nun die Markierungen aus, da nach dem Abbau einer Resource nun immer MPI Send Operationen notwendig sind um den Rückweg einer Ameise zu kennzeichnen.

Insgesamt konnten somit von Seiten des Workers schon extrem viele Nachrichten eingespart werden. Waren es vorher mindestens vier Send und drei Receive Operationen pro Ameise pro Runde, so wird nun im besten Fall nur eine Send Operation pro Ameise benötigt. Im seltenen Fall, dass eine Ameise, eine Resource abbaut, werden je eine Send und eine Receive Operation zusätzlich benötigt. Im Falle, dass eine Ameise auf dem Rückweg zum Bau ist um eine Resource abzugeben, wird, um die Markierung anzuzeigen, ein zusätzlicher Send benötigt.

Auch von Seiten des Masters war noch Verbesserungspotential vorhanden. Da pro Runde sowieso immer alle Ameisen bewegt werden sollen, war es unnötig den Befehl für jede Ameise einzeln zu senden. Stattdessen bekommt jeder Worker nun die Nachricht, jede Ameise für die er zuständig ist zu bewegen, was bereits eine massive Einsparung an MPI Operationen darstellt.

```
453
454         for(index = 2; index < ntasks; index++)
455         {
456             int id;
457             MPI_Send(&id, 1, MPI_INT, index, MPI_MoveAnts, MPI_COMM_WORLD);
458         }
459
```

Weitere Tests mit einer großen Menge an Ameisen ergaben, dass die Worker immernoch zu viele Sends benötigen um die Positionen der Ameisen an den Master weiterzugeben. Statt bisher für jede Ameise einzeln die Informationen zu übermitteln werden diese daher nun in einem Array übermittelt, sodass pro Send bis zu 500 Ameisenpositionen an den Master gesendet werden.

```
710         int antInformation[MultiOperation];
711
712         struct AntList *currentAnt = antList;
713         while(currentAnt->Next != NULL)
714         {
715             currentAnt = currentAnt->Next;
716             if(debugOutput)
717                 printf("Slave_MoveAnt id:%d\n", currentAnt->Value.AntId);
718             calculateAntMovement(&(currentAnt->Value), resourceList, flavorList);
719             antInformation[curCounter] = currentAnt->Value.AtPos.Row;
720             antInformation[curCounter+1] = currentAnt->Value.AtPos.Col;
721
722             curCounter += 2;
723
724             if(curCounter == MultiOperation || currentAnt->Next == NULL)
725             {
726                 if(debugOutput)
727                     printf("Slave_MoveAnt sending:%d of data\n", curCounter);
728                 MPI_Send(antInformation, curCounter, MPI_INT, 0, MPI_MoveAnts, MPI_COMM_WORLD);
729                 curCounter = 0;
730             }
731         }
```

Nun war die Performance zwar immernoch langsamer als die der sequentiellen Version, doch skalierte die Awendung mit 12 Threads und war nur noch ca. Faktor 3-4 langsamer. Wobei hier gesagt werden muss, dass unsere parallele Version um Funktionen erweitert wurde, die die sequentielle Version nicht besaß.

Weitere Optimierungen der parallelen Version

Das Ziel die Hauptrechenzeit auf die für die Simulation notwendigen Operationen zu beschränken war nun erreicht. Die MPI Funktionen nehmen nur noch 2-3% der Rechenzeit ein.

Der Profiler liefert noch einmal überraschende Ergebnisse.

%	cumulative	self		
time	seconds	seconds	calls	name
60.31	9.13	9.13	483124784	getCoordinate
38.81	11.76	2.63	124583	antSeesRessource
32.86	13.74	1.98	124583	antOnResource

Die Funktion getCoordiniate wird ca. eine halbe Milliarden mal aufgerufen und verbraucht 60% der CPU Zeit. Da die Methode lediglich eine Struktur erstellt und wenige simple Berechnungen durchführt, kann man nur Zeit einsparen indem man die Methode weniger häufig aufruft, oder den Code an den verwendeten Stellen direkt einbindet um die Zeit für den Aufruf der Methode zu sparen.

Das inline Schlüsselwort, welches nicht von allen C Compilern unterstützt wird, brachte einen nur sehr kleinen Performancegewinn. Nach Betrachung der aufrufenden Stellen wurde klar, dass jeder Aufruf seine Berechtigung hatte und ein Weglassen ein Fehlverhalten der Anwendung mit sich brächte. Hier wäre es denkbar noch mit schnelleren Listen zu arbeiten, um einen weiteren Performancegewinn zu erhalten. In Frage kämen hier Hashtables, die wir aber aufgrund fehlender Bibliotheken auf dem Cluster nicht realisieren konnten.

Am meisten Aufrufe werden durch die Funktion antSeesFlavor und antSeesResource verursacht. Es stellt sich heraus, dass der Vector den getCoordinate(..) zurück gibt, nicht einmal verwendet wird. Das Ersetzen des Aufrufs durch eine simple Längenbestimmung bringt einen weiteren deutlichen Geschwindigkeitsgewinn.

Gprof liefert für die Top 3 folgenden Output (Kartengröße weiterhin 240x500, allerdings 2.000 Ameisen, 4.000 Simulationsschritte und 12 Threads):

% time	Cumulative seconds	self seconds	calls	name
56.40	23.53	23.53	599600	antSeesFlavor
13.72	29.26	5.73	599600	antOnResource
13.04	34.70	5.44	599600	antSeesRessource

getCoordinate(..) konnte also komplett verdrängt werden. Verdächtig ist nun, dass alle drei Methoden gleich oft aufgerufen werden, dabei sollte doch z.B. gar keine Überprüfung auf antSeesFlavor stattfinden wenn antSeesResource bereits true liefert.

Tatsächlich ergibt sich hier nochmals ein Einsparpotential von einigen Prozent. Wie sich herausstellt, müssen alle Methoden nicht geprüft werden, wenn die Ameise bereits eine Resource trägt, was mittels eines einfachen Zugriff auf die Struktur Ant abrufbar ist. Die Funktion antSeesResource muss zudem auch nicht überprüft werden, wenn antOnResource schon true geliefert hat.

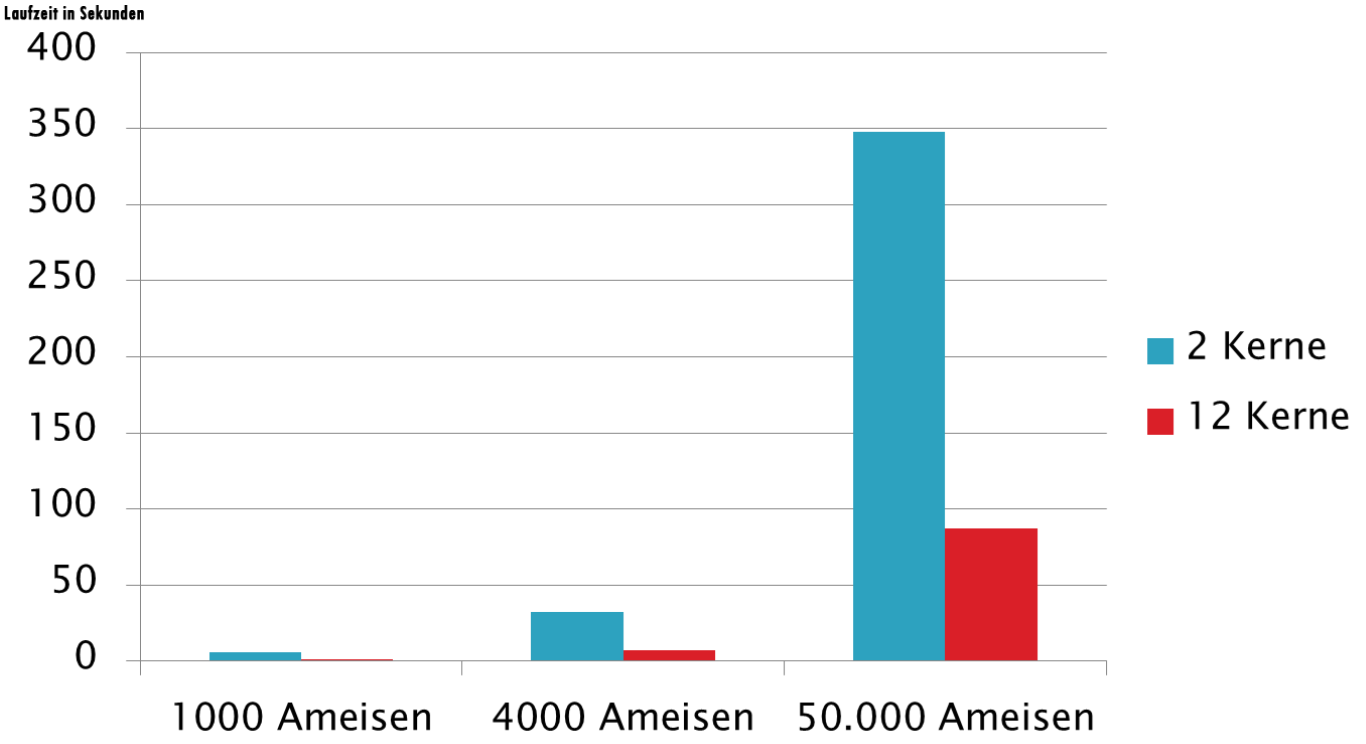
Mittels eines weiteren Flags IsFollowingFlavor für die Struktur Ant kann man ausserdem noch einige Aufrufe von antSeesFlavor einsparen. Eine erneute Auswertung von Gprof ergibt:

% time	cumulative seconds	self seconds	calls	name
48.94	14.28	14.28	433323	antSeesFlavor
17.75	19.46	5.18	538994	antSeesRessource
17.68	24.62	5.16	539728	antOnResource

Skalierung

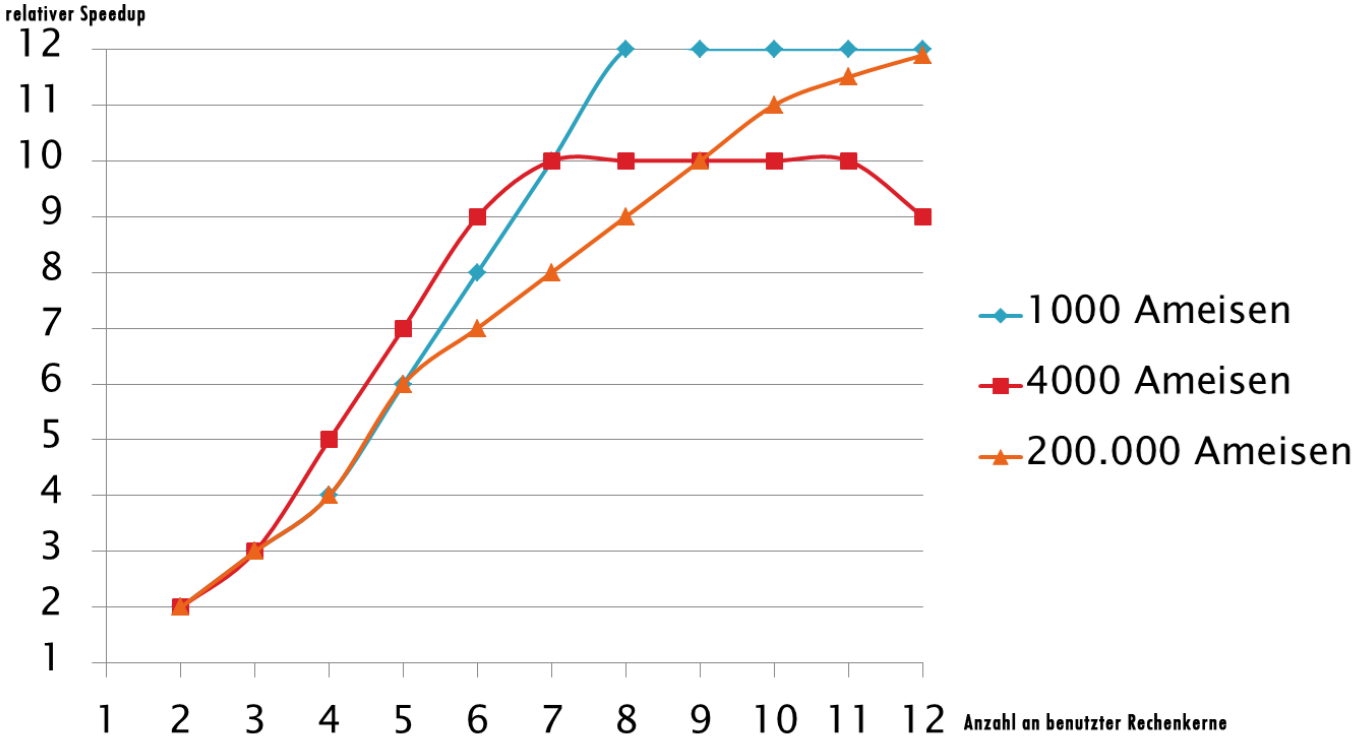
Programmlaufzeit

Als erstes testen wir die Laufzeiten unseres Programmes. Hierfür wählen wir verschiedene Anzahlen an Ameisen. Die Größe der Karte ist in unserer parallelen Version zu vernachlässigen, da nur der Masterthread die Karte in seinem Speicher hält und Auskunft darüber gibt, beeinflusst die Kartengröße maximal die sequentielle Performance des Masterthreads. Hier wäre es später noch möglich mehrere Threads zur Verwaltung der Karte zu implementieren, falls Kartengrößen erwünscht sind, die die Kapazität eines Kernes signifikant übersteigen. Dies ist jedoch in unseren theoretischen Ansätzen nicht notwendig.



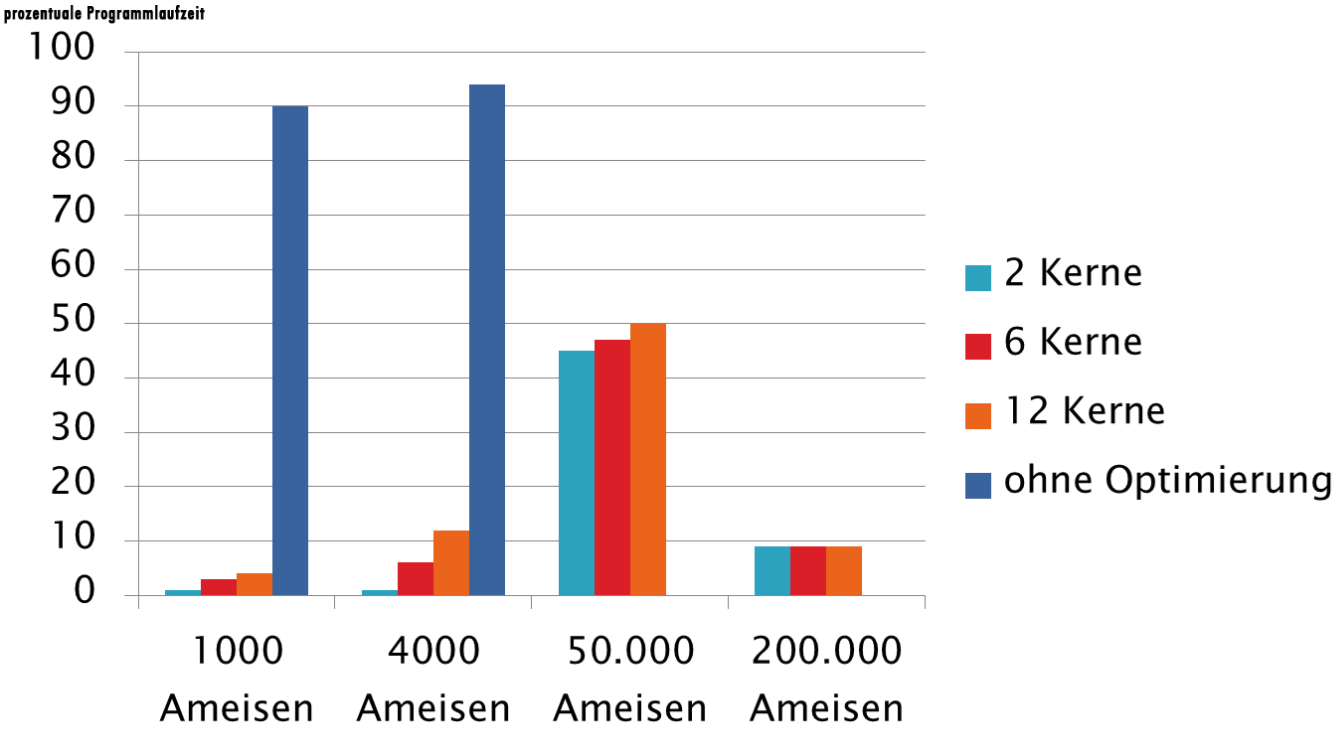
Speedup

Untersuchen wir vorerst die Skalierung unseres Programmes auf nur einem Knoten mit verschiedenen Parametern, so ergeben sich folgende Kurven für 1-12 benutzte Rechenkerne:



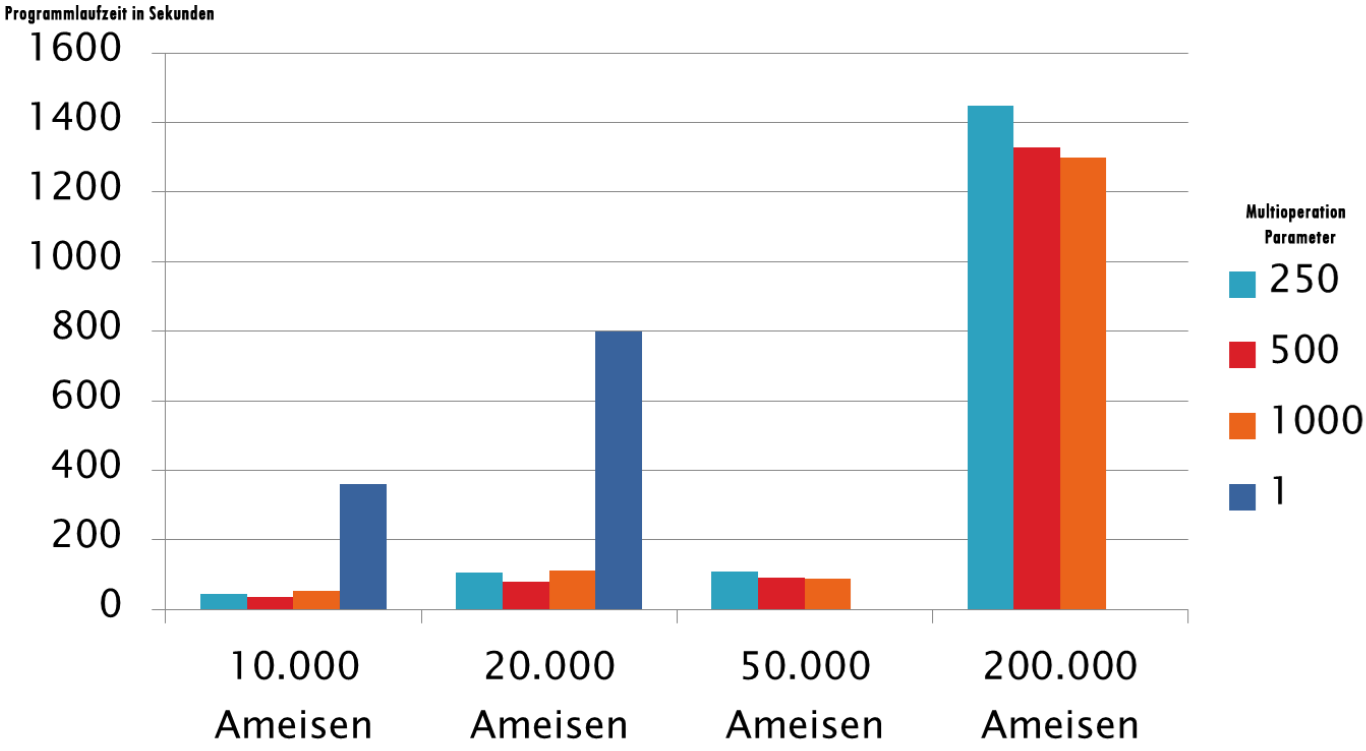
Hierbei fällt auf, dass bei Nutzung von 3-7 Kernen die Version mit 4000 Ameisen am besten Skaliert.Bei 7 Kernen erreicht die Simulation mit 4000 Ameisen aber einen maximalen relativen Speedup. Bei 8 Kernen erreicht dann auch die Version mit 1000 Ameisen einen maximalen relativen Speedup. Die Version mit 200.000 Ameisen Ameisen skaliert nahezu linear bis hin zu 12 benutzten Kernen. Hier vermuten wir, dass unsere Implementierung von [Standardmäßig 500 Operationen] mehreren MPI_Sends dafür ausschlaggebend ist. Daher ist es notwendig sich den prozentualen Anzahl der MPI Kommunikation an der Programmlaufzeit anzugucken.

Overhead der MPI Kommunikation



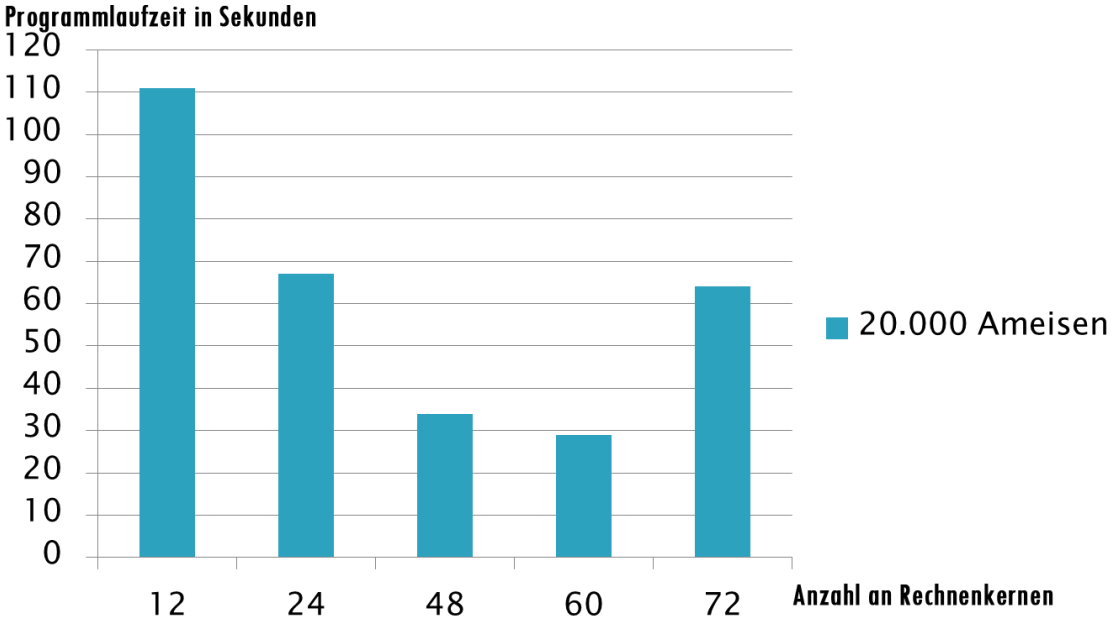
Hier sehen wir, dass bei 1000 Ameisen ohne Optimierung die MPI Kommunikation 90% der Arbeitszeit der Kerne eingenommen hat. Dies war auf die immense Anzahl an Kommunikationsvorgängen zurückzuführen. Nach unseren Optimierungen erhalten wir akzeptable Werte von 0-10% bei 4000 Ameisen. Bei 200.000 Ameisen sinkt der Overhead jedoch rapide. Dies ist darauf zurückzuführen, dass die sequenzielle Rechenzeit jedes Threads im Verhältnis zur Programmlaufzeit steigt. Dies wiederum lässt sich auf die verwendeten Listen zurückführen. Hashtables könnten hier eine Abhilfe bieten, um das Programm noch einmal zu beschleunigen. Wir konnten dies aufgrund fehlender Bibliotheken auf dem Cluster jedoch leider nicht realisieren.

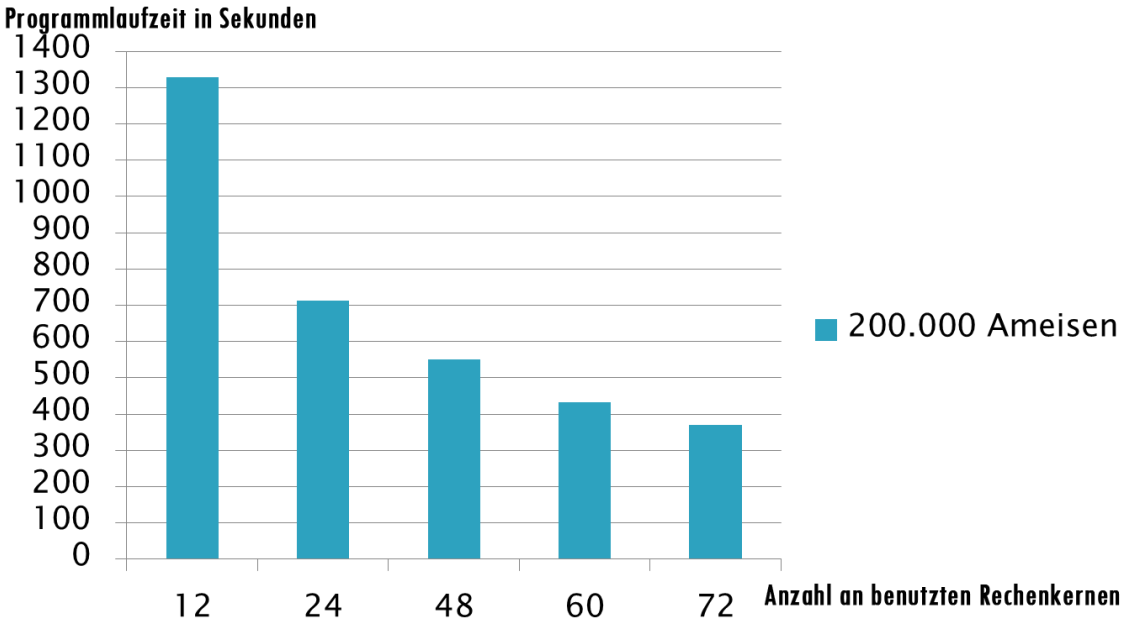
Zusammenfassung von MPI Operationen



Hier sehen wir, dass unser Multioperationsparameter zwar immense Verbesserungen im Vergleich zu einzelnen Operationen bietet, der Unterschied zwischen 250-1000 zusammengefasster Operationen jedoch marginal ist. Des Weiteren erkennen wir, dass es gerade bei kleineren Ameisenzahlen eher nachteilig ist größere Anzahlen an MPI Operationen zusammenzufassen, bei größeren Anzahlen steigt der Nutzen jedoch wieder. Wir führen dies darauf zurück, dass die einzelnen Kerne bei kleineren Anzahlen an Ameisen keinen großen Nutzen aus den zusammengefassten Operationen ziehen können, da hierdurch Wartezeiten entstehen. Bei großen Anzahlen an Ameisen reduziert ein hoher Grad an Zusammenfassung jedoch auch die Laufzeit. Jedoch lässt sich über 1000 zusammengefasster Operationen hinaus kein Nutzen mehr feststellen. Dies führen wir darauf zurück, dass die Kerne bei 2,7 GHZ Takt eine bestimmte Anzahl an Ameisen pro Zyklus verarbeiten und dieser Wert scheinbar zwischen 500 und 1000 Ameisen liegt, wodurch es nicht sinnvoll ist mehr als diese 1000 Ameisen pro Zyklus zusammenzufassen, da dadurch Wartezeiten entstehen.

Abschließend betrachten wir noch einmal die Skalierung unseres Programmes auf mehreren Knoten des Clusters. Wir erwarten hierbei einen nicht mehr linearen Speedup, da die Kommunikation zwischen den Knoten erheblich langsamer ist als innerhalb eines Knotens.





Wie man den Diagrammen entnehmen kann skaliert das Programm umso besser je mehr Ameisen für die Simulation erzeugt werden. Dies liegt daran, dass dadurch die relative Rechenlast pro Kern steigt und die MPI Kommunikation pro Sekunde pro Kern reduziert wird. Der Masterthread kann jedoch auch bei 72 Kernen und 200.000 Ameisen noch schnell genug Informationen bereitstellen, wodurch bei mehr als 200.000 Ameisen eventuell auch ein Speedup über 72 Kerne hinaus möglich sein könnte.