„Ants“

Paralleles Programmieren
SS 2012

Von Dominik Rupp (0rupp@informatik.uni-hamburg.de)
Und Christian Peter (0peter@informatik.uni-hamburg.de)

Idee des Programmes

- Realitätsnahe Simulation der Nahrungssuche eines Ameisenvolkes
- Ameisenbewegung ist zufällig während sie nach Nahrung suchen
  - Wenn sie Zucker finden, gehen sie zum Bau zurück
  - Auf dem Weg werden Duftstoffe verstreut
  - Andere in der nähe befindliche Ameisen können diese Duftstoffe riechen und folgen ihnen
    - => Führt zu Schwarmintelligenz

Parallelisierung

- Zwei Möglichkeiten:
  - Aufteilen der Karte in Segmente und Verteilung auf verschiedene Prozessorkerne
    - Vorteil: Kommunikation nur am Rand der Kartenabschnitte notwendig
    - Nachteil: Last durch stark benutzte Hotspots auf wenige Kerne verteilt
    - Dies erfordert einen schwer umsetzbaren Lastausgleich durch dynamisches anpassen der Segmente
  - Verteilung der Ameisen auf verschiedene Prozessorkerne
    - Da Ameisen gewöhnlich nicht kollidieren und nicht direkt kommunizieren, ist dies für uns die sinnvollste Möglichkeit zur Parallelisierung
    - Lastausgleich wesentlich effektiver und leichter umsetzbar

Unser Parallelisierungsansatz

- 1 Masterthread
  - hält die Daten über die Rohstoffe/Duftstoffe auf der Karte
  - Synchronisiert die anderen Threads
  - Verteilt die entstehenden Ameisen an die Threads
  - Kennt Anzahl an Ameisen
  - Kann Ameisen eindeutig Threads zuordnen über Modulo Funktion

- X workingthreads
  - führen die Berechnungen auf den Ameisen durch
  - Jeder Thread hat seinen eigenen Datensatz an Ameisen
    - Daten von Ameisen auf Workerthreads weisen keine Überschneidungen auf

Parallelisierungsansatz

- Workerthreads
  - halten in ihrem eigenen Speicher die Daten über ihre Ameisen
    - Bekommt Anweisung vom Masterthread eine Ameise zu erstellen (pro Ameise eine einmalige Kommunikation)
    - Muss die Daten der Ameise danach nicht mehr verschicken
  - Halten in ihrem Speicher eine temporäre Kopie der Rohstoffvorkommen
    - Wird in jedem Simulationsdurchgang mit Masterthread synchronisiert (eine Send/Receive Operation pro Durchgang)
  - Stellen Anfrage, ob Ressource noch da ist und abgebaut werden kann (pro Durchgang maximal pro Ameise eine Send/Receive Operation)
  - Berechnet die Bewegung der Ameise
    - Abhängig davon ob Rohstofffeld/Duftstoff in „Sichtweite“ sehr rechenintensiv

Parallelisierungsansatz

- Masterthread
  - Hat Kenntnis über Anzahl an Ameisen und Zugehörigkeit zu den Threads
  - Hält Rohstoffvorkommen und Duftstoffe im Speicher
    - hat Methoden um Duftstoffe und Rohstoffvorkommen mit den Workerthreads zu kommunizieren
      - Workerthreads stellen Anfrage, ob die Ameise die Ressource abbauen kann (pro Ameise pro Durchgang ein Recv + Send)
      - Workerthreads übergeben neue Duftstoffe und fragen diese ab (pro Simulationsdurchgang pro Thread nur ein Recv + Send)

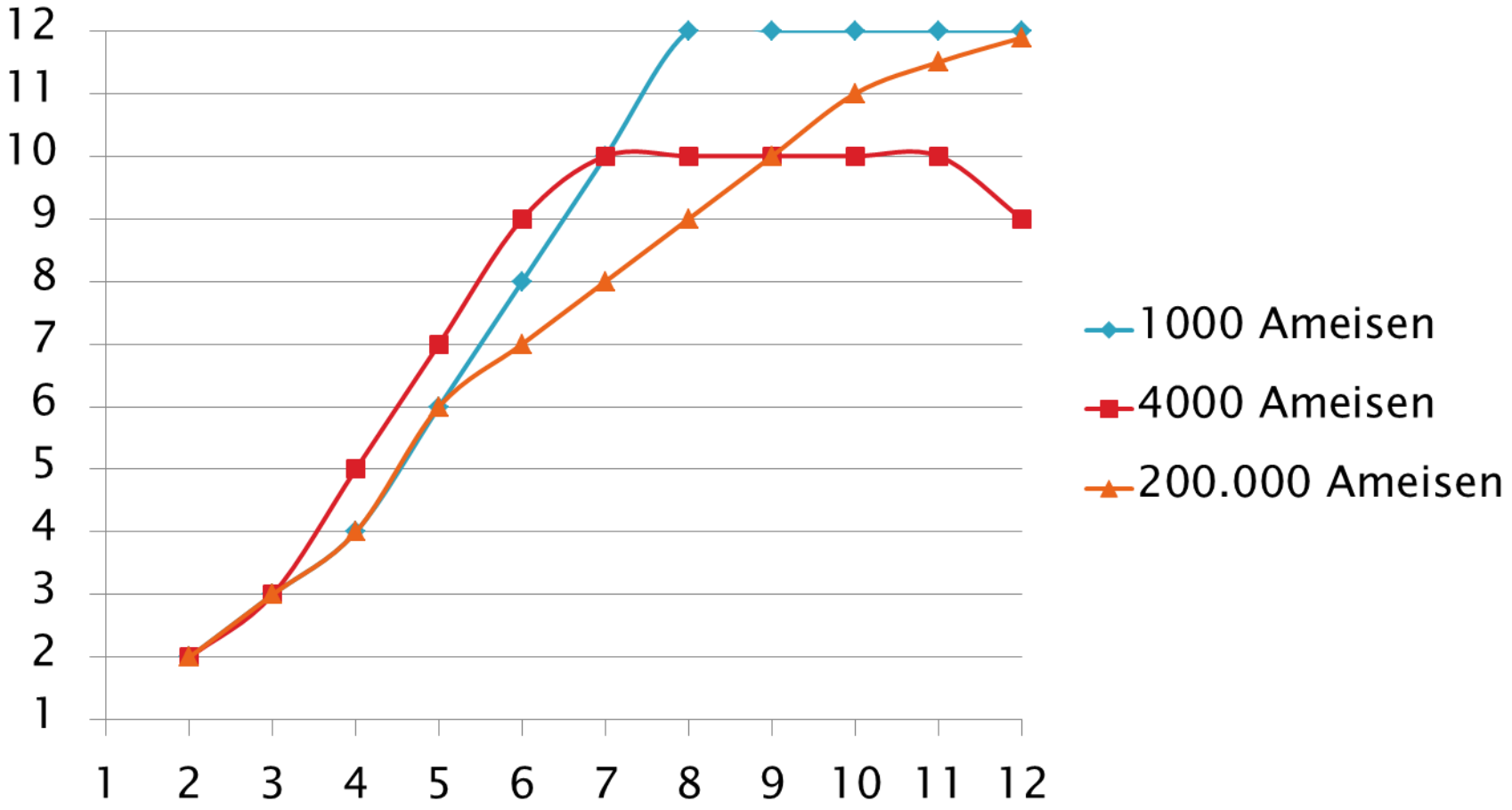
Probleme beim Parallelisieren

- Anfangs zu viel Kommunikationsoverhead
  - 90% des Aufwandes der Threads war durch MPI bedingt
  - Kommunikationsaufwand = Ameisen + (Simulationsdurchgänge*Threads) + (Simulationsdurchgänge * Ameisen)
  - Bei 50.000 Ameisen, 12 Threads und 1000 Simulationsdurchgängen bereits 50.062.000 Kommunikationsoperationen
- Asynchronität: Rohstoffe abgebaut die nicht mehr existierten, gesetzte Duftstoffe wurden ignoriert
- Ein Kern war bereits im nächsten Simulationsschritt, obwohl andere noch nicht so weit waren

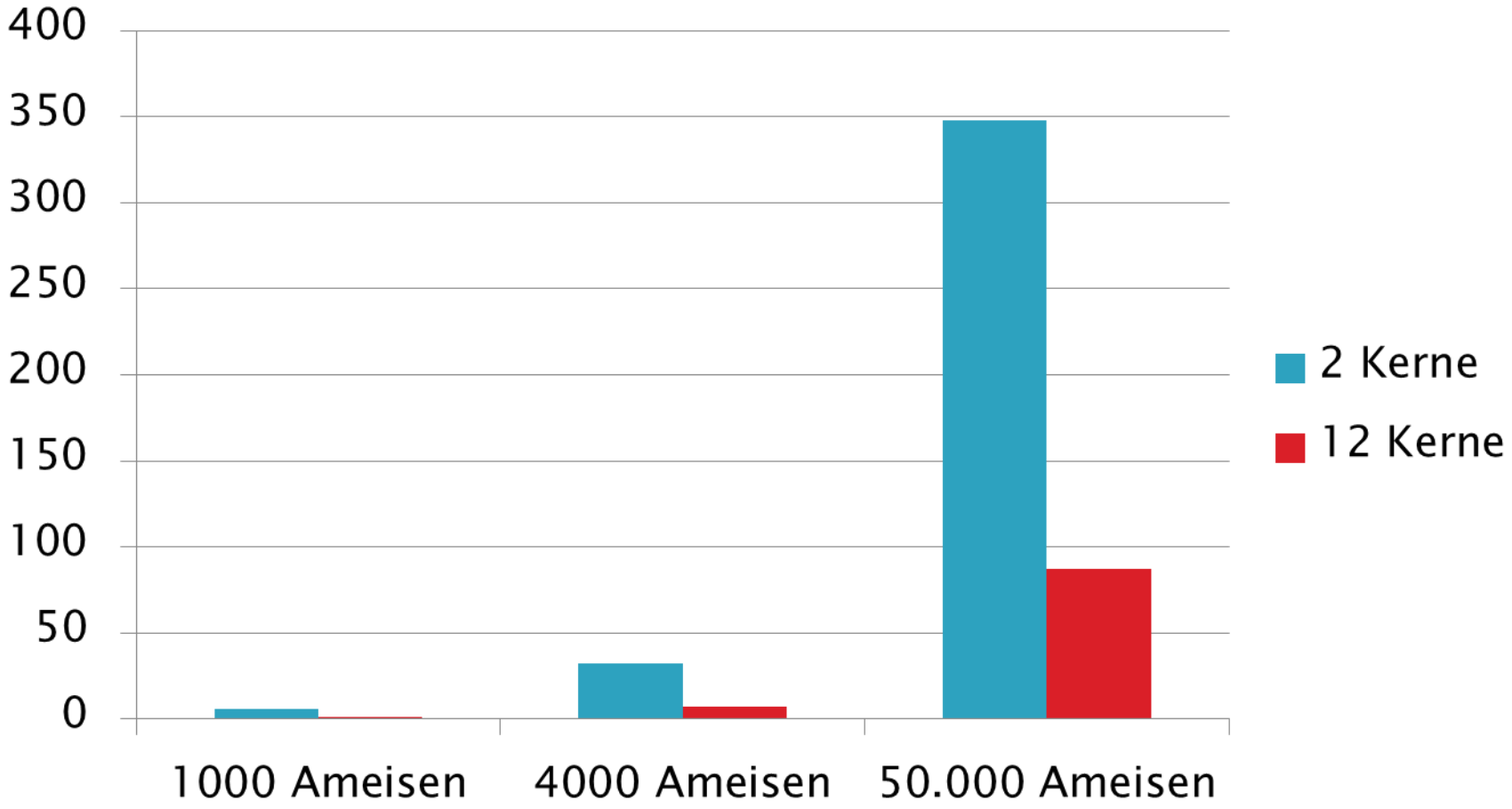
Lösungsversuche

- Kommunikation zwischen Threads wurde in Blöcke zu 250-1000 Kommunikationen zusammengefasst
  - => erheblicher Performancegewinn von bis zu 1000%
- Asynchronität: Einführung des Masterthreads
  - Anfänglich versucht alles ohne Masterthread zu lösen
    - Dezentrale Verwaltung der Umweltdaten
  - Der Programmiertechnische Aufwand schien uns zu groß für einen maximalen Speedup von 1.083 bei Nutzung von 12 Kernen.

Erreichter Speedup



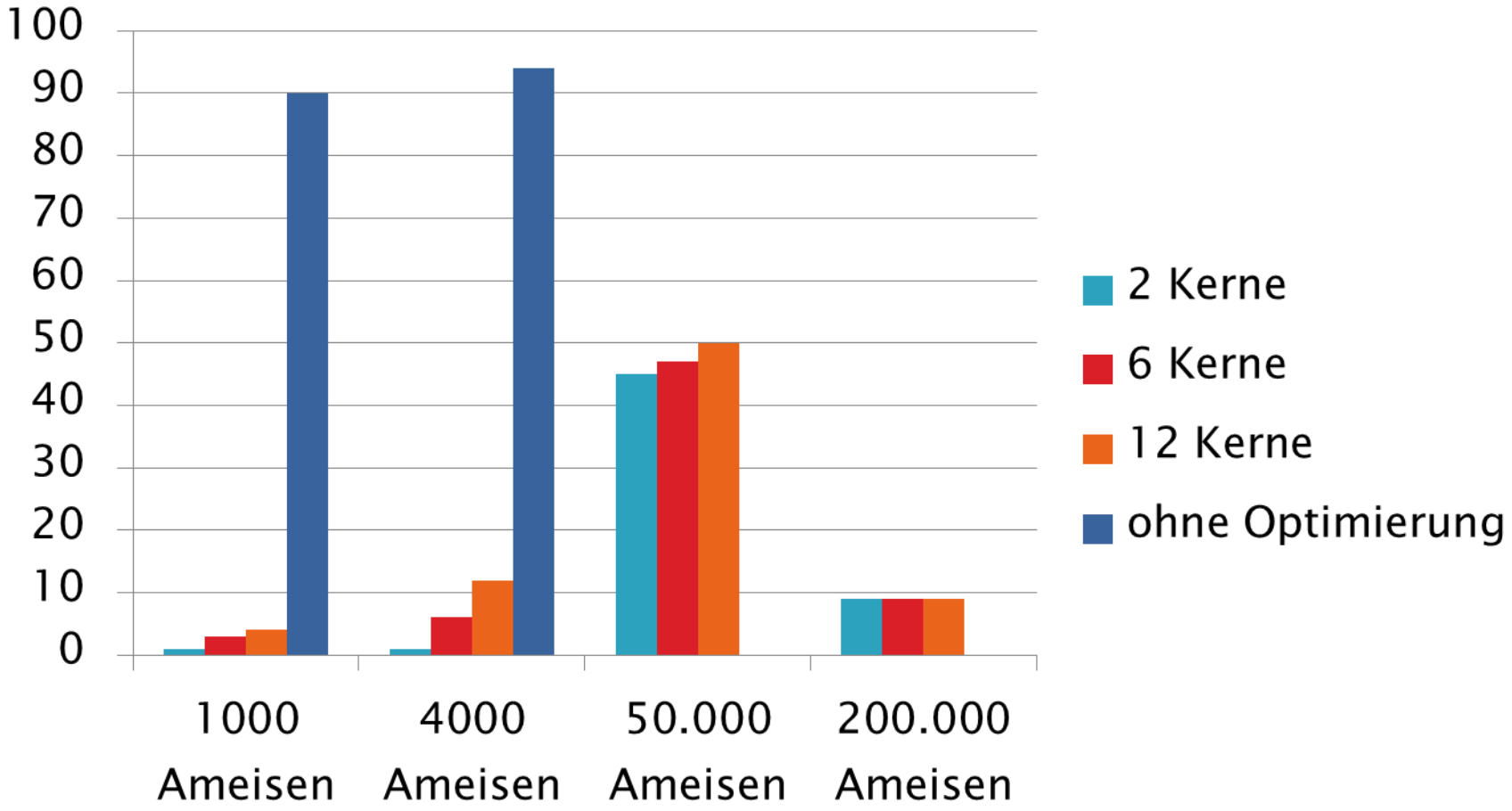
Programmlaufzeiten



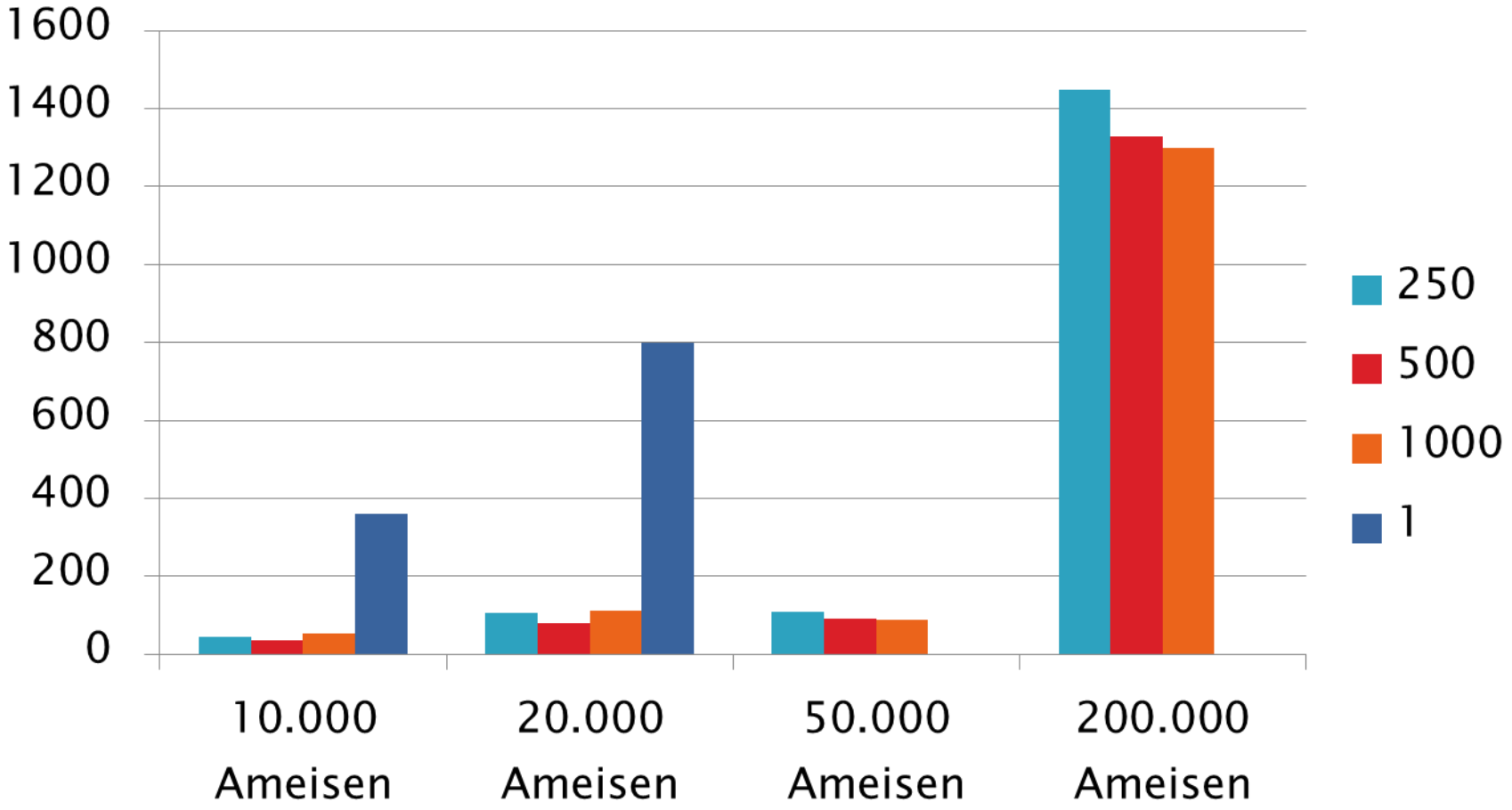
Optimierungsmaßnahmen

- Nutzung von Inline Funktionen wegen extrem hoher Anzahl an Callbacks
  - Vorgabe an Compiler Programmcode möglichst statt Referenz, die Funktion direkt einzufügen
- Zusammenfassung mehrerer Kommunikationsoperationen
  - 500-1000 als optimale Anzahl auf dem Cluster für unser Programm
    - Abhängig von jeweiliger Taktrate und Busgeschwindigkeit
    - Von Programm zu Programm sehr unterschiedlich

MPI Overhead in % (gemessen mit gprof)



Multioperation MPI



Multioperation MPI

- Zusammenfassen von Kommunikationsoperationen effektiv
  - Optimale Anzahl muss gefunden werden.
  - Mehr nicht immer besser
  - Ab 1000 geht es in unserem Fall wieder bergab
  - 500 in den meisten Konfigurationen optimal

Weitergehende Optimierung

- Da das Programm aktuell nur einen Masterthread nutzt, kann es nicht unendlich skalieren.
  - Mehrere Masterthreads, die jeweils Segmente der Karten verwalten wären denkbar, da diese durch einfache Funktion ihrem Segment zugeordnet werden können
  - Keine Kommunikation zwischen Masterthreads außer dem Synchronisationssignal notwendig

Weitergehende Optimierung

- 9% MPI Overhead bei 200.000 Ameisen vermutlich auf langsame Listen zurückführbar
  - Hash tables könnten eventuell helfen
  - Beeinträchtigt jedoch nicht den Speedup, da jeder Kern nur auf seinen Daten rechnet und die nur Ergebnisse ausgetauscht werden
  - Zeigt darüber hinaus, dass Speedup noch weit über 12 Kerne hinaus möglich ist