Rechnerstrukturen im SS 2009

Parallelismus und Parallele Programmierung

Oliver Mattes
mattes@ira.uka.de

Universität Karlsruhe (TH)
Forschungsuniversität • gegründet 1825

Institut für Technische Informatik
Lehrstuhl für Rechnerarchitektur

18. Juni 2009

Überblick

Parallelismus:

- Quantitative Maßzahlen
- Aufgabe 1.1
- Aufgabe 1.2
- Aufgabe 1.3

Parallele Programmierung:

- Parallelisierungsprozess
- Programmiermodelle
- Aufgabe 2

Sonstiges:

- Hinweise zur Klausur

Parallelismus: Quantitative Maßzahlen

Definitionen:

Definitionen:

- $P(1)$ Anzahl der Einheitsoperationen auf einem Einprozessorsystem
- $P(n)$ Anzahl der Einheitsoperationen auf einem Multiprozessorsystem mit n Prozessoren
- $T(1)$ Ausführungszeit auf einem Einprozessorsystem in Schritten
- $T(n)$ Ausführungszeit auf einem Multiprozessorsystem mit n Prozessoren in Schritten

Vereinfachende Voraussetzungen:

- $T(1) = P(1)$
- $T(n) \leq P(n)$

Parallelismus: Quantitative Maßzahlen

Beschleunigung (Speed-Up)

$$S(n) = \frac{T(1)}{T(n)}$$

üblicherweise gilt:

$$1 \leq S(n) \leq n$$

Effizienz

$$E(n) = \frac{S(n)}{n} = \frac{T(1)}{n * T(n)}$$

daraus folgt:

$$\frac{1}{n} \leq E(n) \leq 1$$

Parallelismus: Quantitative Maßzahlen

Mehraufwand für die Parallelisierung

$$R(n) = \frac{P(n)}{P(1)}$$

$$1 \leq R(n)$$

Mehraufwand für die Organisation, Synchronisation und Kommunikation der Prozessoren in Multiprozessorsystemen.

Parallelindex

$$I(n) = \frac{P(n)}{T(n)}$$

$$1 \leq S(n) \leq I(n) \leq n$$

Mittlerer Grad der Parallelität. Anzahl der parallelen Operationen pro Zeiteinheit.

Parallelismus: Quantitative Maßzahlen

Amdahls Gesetz (Amdahl's Law)

$$T(n) = \underbrace{\frac{1}{n} * T(1) * (1-a)}_{1} + \underbrace{T(1) * a}_{2}$$

- a mit $(0 \leq a \leq 1)$ ist der Anteil eines Programms, der nur sequentiell ausgeführt werden kann.

1. Ausführungszeit des parallel ausführbaren Programmteils $(1-a)$ dividiert durch die Anzahl n der parallel arbeitenden Prozessoren.
2. Ausführungszeit des nur sequentiell ausführbaren Programmteils a.

Parallelismus: Quantitative Maßzahlen

Amdahls Gesetz (Amdahl's Law)

$$T(n) = \frac{T(1)}{n} * (1 - a) \quad + \quad T(1) * a$$

Durch Einsetzen in die Formel für die Beschleunigung erhält man:

$$\begin{aligned} S(n) &= \frac{T(1)}{T(n)} = \frac{T(1)}{\frac{T(1)}{n} * (1 - a) + T(1) * a} \\ &= \frac{1}{(1 - a) * \frac{1}{n} + a} \\ S(n) &\leq \frac{1}{a} \end{aligned}$$

$\Rightarrow$ Die erreichbare Beschleunigung wird durch den sequentiellen Programmteil begrenzt.

Parallelismus: Quantitative Maßzahlen

Skalierbarkeit

- Von Skalierbarkeit spricht man, wenn das Hinzufügen von weiteren Verarbeitungselementen zu einer kürzeren Gesamtausführungszeit führt, ohne daß das Programm geändert werden muß.
- Wichtig für die Skalierbarkeit ist eine angemessene Problemgröße

Parallelismus: Quantitative Maßzahlen

Weiteres

- Superlinearer Speed-Up
- Auslastung U
- Algorithmenabhängigkeit $\Leftrightarrow$ algorithmenunabhängigkeit
- Kommunikationskosten
- Definition nach Flynn

Aufgabe 1.1 – Leistungsbewertung

Gegeben sei ein Multiprozessorsystem mit 16 Prozessoren. Die Leistungssteigerung gegenüber einem Einprozessorsystem sei $S(16) = 8$. Die Ausführungszeit auf dem Einprozessorsystem sei $T(1) = 80$ und die Anzahl der auszuführenden Einheitsoperationen auf dem Multiprozessorsystem sei $P(16) = 120$.

a) Berechnen Sie die Effizienz $E(16)$, die parallele Ausführungszeit $T(16)$ und den Parallelindex $I(16)$.

b) Interpretieren Sie den berechneten Parallelindex.

c) Ermitteln Sie anhand von Amdahls Gesetz den Bruchteil des Programms, der nur sequentiell ausführbar ist.

Aufgabe 1.1 – Leistungsbewertung

a) **Berechnen Sie die Effizienz $E(16)$, die parallele Ausführungszeit $T(16)$ und den Parallelindex $I(16)$.**

$$E(n) = \frac{S(n)}{n} \quad \Rightarrow \quad E(16) = \frac{S(16)}{16} = \frac{8}{16} = 0,5$$

$$S(n) = \frac{T(1)}{T(n)} \quad \Rightarrow \quad T(16) = \frac{T(1)}{S(16)} = \frac{80}{8} = 10$$

$$I(16) = \frac{P(n)}{T(n)} \quad \Rightarrow \quad I(16) = \frac{P(16)}{T(16)} = \frac{120}{10} = 12$$

Aufgabe 1.1 – Leistungsbewertung

b) **Interpretieren Sie den berechneten Parallelindex.**

- Der Parallelindex gibt den mittleren Grad der Parallelität an, d.h. die Anzahl der parallelen Operationen pro Zeiteinheit.
- Der berechnete Wert ist kleiner als die Anzahl der Prozessoren. Das bedeutet, dass zeitweise einige Prozessoren keinen Befehl ausführen.

Aufgabe 1.1 – Leistungsbewertung

b) **Interpretieren Sie den berechneten Parallelindex.**

- Der Parallelindex gibt den mittleren Grad der Parallelität an, d.h. die Anzahl der parallelen Operationen pro Zeiteinheit.
- Der berechnete Wert ist kleiner als die Anzahl der Prozessoren. Das bedeutet, dass zeitweise einige Prozessoren keinen Befehl ausführen.
- Besser verdeutlicht wird dies durch Berechnung der Auslastung U, die angibt wieviel Operationen jeder Prozessor im Durchschnitt pro Zeiteinheit ausführt.

$$U(n) = \frac{I(n)}{n} \quad \Rightarrow \quad U(16) = \frac{12}{16} = \frac{3}{4} = 75\,\%$$

Aufgabe 1.1 – Leistungsbewertung

c) **Ermitteln Sie anhand von Amdahls Gesetz den Bruchteil des Programms, der nur sequentiell ausführbar ist.**

Ahmdahls Gesetz:

$$T(n) = T(1) * \left(\frac{(1-a)}{n} + a \right) \; = \; T(1) * \frac{1 - a \; + \; na}{n}$$

Aufgabe 1.1 – Leistungsbewertung

c) **Ermitteln Sie anhand von Amdahls Gesetz den Bruchteil des Programms, der nur sequentiell ausführbar ist.**

Ahmdahls Gesetz:

$$T(n) = T(1) * \left(\frac{(1-a)}{n} + a \right) \; = \; T(1) * \frac{1 - a \; + \; na}{n}$$

$$\Rightarrow \qquad 10 = 80 * \frac{1 - a \; + \; 16a}{16} = 80 * \frac{1 \; + \; 15a}{16} = 5 * (1 + 15a)$$

Aufgabe 1.1 – Leistungsbewertung

c) **Ermitteln Sie anhand von Amdahls Gesetz den Bruchteil des Programms, der nur sequentiell ausführbar ist.**

Ahmdahls Gesetz:

$$T(n) = T(1) * \left(\frac{(1-a)}{n} + a \right) \; = \; T(1) * \frac{1 - a \; + \; na}{n}$$

$$\Rightarrow \quad 10 = 80 * \frac{1 - a \; + \; 16a}{16} = 80 * \frac{1 \; + \; 15a}{16} = 5 * (1 + 15a)$$

$$\Rightarrow \quad 15a = 1 \;\; \Rightarrow \;\; a = \frac{1}{15} \approx 6,7\,\%$$

$\approx 6,7\,\%$ des Programmcodes sind nur sequentiell ausführbar.

Aufgabe 1.2 – Leistungsbewertung

Die Ausführungszeit einer sequentiellen Anwendung betrage T_{seq}. Von dieser Anwendung lassen sich 20 % nicht parallelisieren. Die verbleibenden 80 % werden zwischen den Prozessoren gleichmäßig verteilt. Das bedeutet, dass jeder Prozessor ungefähr den gleichen Anteil der zu parallelisierenden Aufgabe bearbeitet und jeder gleichviel Zeit benötigt.

Beispielsweise betrage die Ausführungszeit der parallelen Anteile der Anwendung 20 % der sequentiellen Ausführungszeit, wenn vier Prozessoren sie ausführen.

a) Setzen Sie voraus, dass die Parallelisierung keinen Aufwand verursacht. Berechnen Sie die Beschleunigung und die Effizienz bei einer unterschiedlichen Anzahl von Prozessoren. Fügen Sie die Ergebnisse in die vorgegebenen Tabelle ein.

Aufgabe 1.2 – Leistungsbewertung

a) **Setzen Sie voraus, dass die Parallelisierung keinen Aufwand verursacht. Berechnen Sie die Beschleunigung und die Effizienz bei einer unterschiedlichen Anzahl von Prozessoren. Fügen Sie die Ergebnisse in die vorgegebenen Tabelle ein.**

Par. Ausführungszeit: $T(n) = \frac{80\,\%}{n} * T_{seq} \; + \; 20\,\% * T_{seq}$

Beschleunigung: $S(n) = \frac{T_{seq}}{T(n)}$

Effizienz: $E(n) = \frac{S(n)}{n}$

Aufgabe 1.2 – Leistungsbewertung

a) **Setzen Sie voraus, dass die Parallelisierung keinen Aufwand verursacht. Berechnen Sie die Beschleunigung und die Effizienz bei einer unterschiedlichen Anzahl von Prozessoren. Fügen Sie die Ergebnisse in die vorgegebenen Tabelle ein.**

Par. Ausführungszeit: $T(n) = \frac{80\,\%}{n} * T_{seq} \; + \; 20\,\% * T_{seq}$

Beschleunigung: $S(n) = \frac{T_{seq}}{T(n)}$

Effizienz: $E(n) = \frac{S(n)}{n}$

Prozessoren	n=2	n=4	n=8	n=16	n=32
Beschleunigung	5/3=1,67	5/2=2,50	10/3=3,33	4	40/9=4,44
Effizienz	5/6=0,83	5/8=0,625	10/24=0,42	1/4=0,25	5/36=0,14

Achtung: In der Klausur bei Berechnungen wenn möglich Brüche statt gerundete Zahlen als Ergebnisse angeben!

Aufgabe 1.2 – Leistungsbewertung

Prozessoren	n=2	n=4	n=8	n=16	n=32
Beschleunigung	5/3=1,67	5/2=2,50	10/3=3,33	4	40/9=4,44
Effizienz	5/6=0,83	5/8=0,625	10/24=0,42	1/4=0,25	5/36=0,14

b) **Evaluieren Sie zusätzlich die Skalierbarkeit der einzelnen Lösungen**

Aufgabe 1.2 – Leistungsbewertung

Prozessoren	n=2	n=4	n=8	n=16	n=32
Beschleunigung	5/3=1,67	5/2=2,50	10/3=3,33	4	40/9=4,44
Effizienz	5/6=0,83	5/8=0,625	10/24=0,42	1/4=0,25	5/36=0,14

b) **Evaluieren Sie zusätzlich die Skalierbarkeit der einzelnen Lösungen**

Die Skalierbarkeit ist sehr schlecht. Grund dafür ist, dass ein großer Anteil des Programms nicht parallelisierbar ist. Die Ausführungszeit dieses Anteils bestimmt mit steigender Anzahl von Prozessoren einen zunehmenden Anteil der gesamten Ausführungszeit.

Beschleunigung: $S(n) = \frac{T_{seq}}{\left(20\,\% + \frac{80\,\%}{n}\right) * T_{seq}} \xrightarrow{n \text{ sehr groß}} \frac{1}{20\,\%} = 5$

Die Effizienz strebt damit gegen 0.

Aufgabe 1.2 – Leistungsbewertung

c) **Zur Steigerung der Genauigkeit der Berechnung nehmen Sie nun an, dass durch jeden verwendeten Prozessor eine zusätzliche Berarbeitungszeit von 1 % der sequentiellen Ausführungszeit benötigt wird. Berechnen Sie Beschleunigung und Effizienz für 64 Prozessoren.**

Aufgabe 1.2 – Leistungsbewertung

c) **Zur Steigerung der Genauigkeit der Berechnung nehmen Sie nun an, dass durch jeden verwendeten Prozessor eine zusätzliche Berarbeitungszeit von 1 % der sequentiellen Ausführungszeit benötigt wird. Berechnen Sie Beschleunigung und Effizienz für 64 Prozessoren.**

Beschleunigung:

$$S(n) = \frac{T_{seq}}{\left(20\,\% + \frac{80\,\%}{n} + 1\,\% * n\right) * T_{seq}}$$

$$\Rightarrow \quad S(64) \approx \quad 1,17$$

Effizienz:

$$E(64) = \frac{S(64)}{64} = \frac{1,17}{64} = 0,018$$

Aufgabe 1.3 – Leistungsbewertung

Ein Einprozessorsystem soll erweitert werden. Dabei existieren folgende, in der Anschaffung gleich teure Alternativen:

- Ausbau zu einem 2-fach SMP-System
  - Installation eines zweiten identischen Hauptprozessors
  - Zugriff auf einen gemeinsamen Hauptspeicher
  - Synchronisationsoverhead: Ausführung des Programms wird um 2 % der unparallelisierten Ausführungszeit erhöht.
- Einsetzen eines mathematischen Koprozessors
  - 10x schnellere Ausführung der Gleitkommaarithmetik
  - Keine parallele Verarbeitung, d.h. der gleichzeitige Einsatz von Haupt- und Koprozessor, möglich.

Das zu bearbeitende Problem ist zu 74 % parallelisierbar. Der Anteil der Gleitkommaarithmetik am Gesamtprogramm beträgt 40 %. Bestimmen Sie, welche der beiden Möglichkeiten unter dem Gesichtspunkt der Ausführungszeit zu bevorzugen ist.

Aufgabe 1.3 – Leistungsbewertung

Die Ausführungszeiten lassen sich in beiden Fällen wiederum mit Hilfe von Ahmdahls Gesetz ausrechnen und damit auch die erreichte Beschleunigung vergleichen. T_{seq} ist hierbei die Zeit ohne Parallelisierung oder Koprozessorunterstützung.

Aufgabe 1.3 – Leistungsbewertung

Die Ausführungszeiten lassen sich in beiden Fällen wiederum mit Hilfe von Ahmdahls Gesetz ausrechnen und damit auch die erreichte Beschleunigung vergleichen. T_{seq} ist hierbei die Zeit ohne Parallelisierung oder Koprozessorunterstützung.

- SMP-System:

$$T_{SMP} = 74\,\% * \frac{1}{2} * T_{seq} + 26\,\% * T_{seq} + 2\,\% * T_{seq} = 65\,\% * T_{seq}$$

$$S_{SMP} = \frac{T_{seq}}{T_{SMP}} = \frac{1}{65\,\%} \approx 1,5385$$

Aufgabe 1.3 – Leistungsbewertung

Die Ausführungszeiten lassen sich in beiden Fällen wiederum mit Hilfe von Ahmdahls Gesetz ausrechnen und damit auch die erreichte Beschleunigung vergleichen. T_{seq} ist hierbei die Zeit ohne Parallelisierung oder Koprozessorunterstützung.

- SMP-System:

$$T_{SMP} = 74\,\% * \frac{1}{2} * T_{seq} + 26\,\% * T_{seq} + 2\,\% * T_{seq} = 65\,\% * T_{seq}$$

$$S_{SMP} = \frac{T_{seq}}{T_{SMP}} = \frac{1}{65\,\%} \approx 1,5385$$

- System mit Koprozessor:

$$T_{CP} = 40\,\% * \frac{1}{10} * T_{seq} + 60\,\% * T_{seq} = 64\,\% * T_{seq}$$

$$S_{CP} = \frac{T_{seq}}{T_{CP}} = \frac{1}{64\,\%} \approx 1,5625$$

Aufgabe 1.3 – Leistungsbewertung

- SMP-System:

$$S_{SMP} \approx 1,5385$$

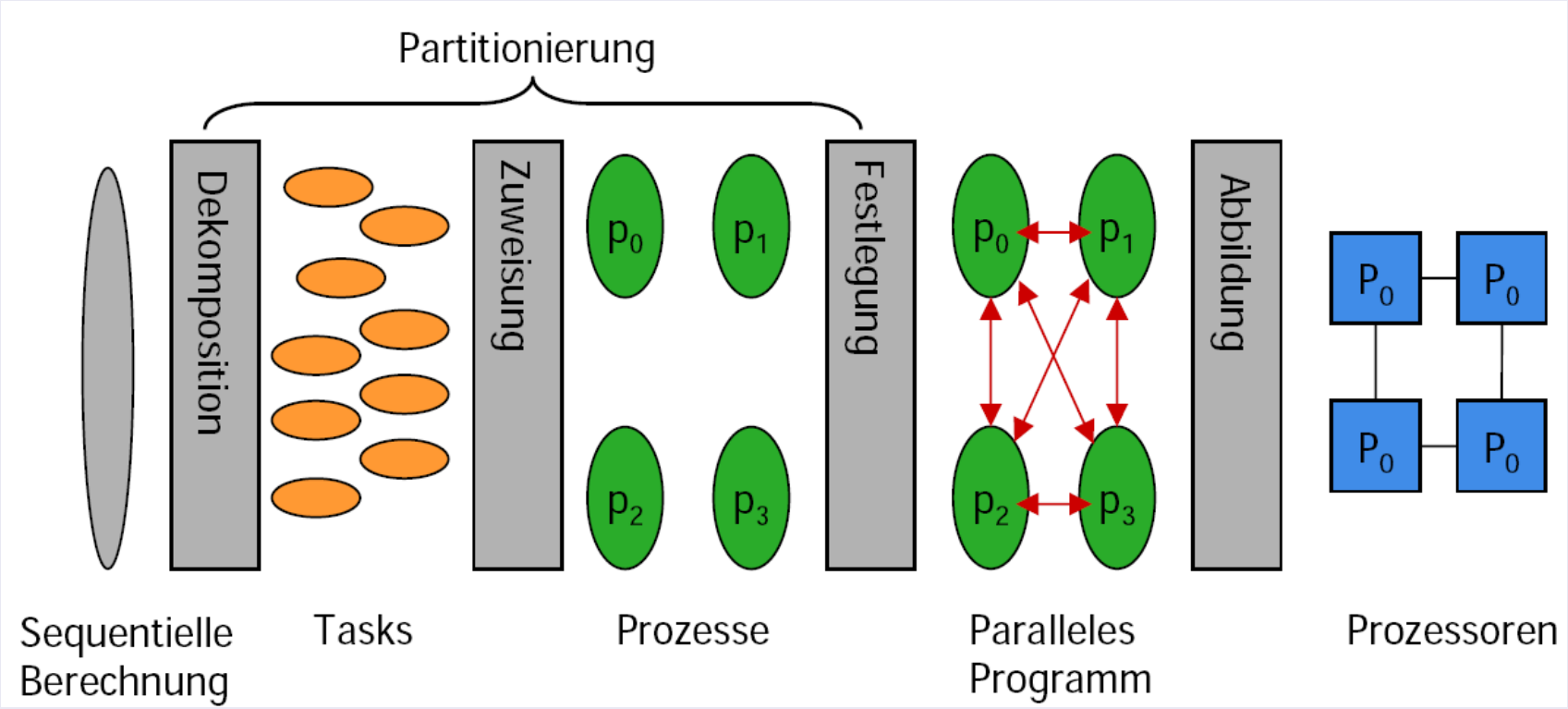
- System mit Koprozessor:

$$S_{CP} \approx 1,5625$$

$\Rightarrow$ Die Koprozessorlösung ist für diese Anwendung deshalb dem SMP-System vorzuziehen.

$\Rightarrow$ Ohne Beachtung der benötigten Synchronisationszeit würde die Berechnung ein fehlerhaftes Ergebnis liefern.

Parallelisierungsprozess



Parallelisierungsprozess

Ausgangspunkt ist ein sequentielles Programm

1. **Dekomposition** (oder Aufteilung) der Berechnung in Tasks

Parallelisierungsprozess

Ausgangspunkt ist ein sequentielles Programm

1. **Dekomposition** (oder Aufteilung) der Berechnung in Tasks
2. **Zuweisung** der Tasks zu Prozessen

Parallelisierungsprozess

Ausgangspunkt ist ein sequentielles Programm

1. **Dekomposition** (oder Aufteilung) der Berechnung in Tasks
2. **Zuweisung** der Tasks zu Prozessen
3. **Festlegung** des notewendigen Datenzugriffs, der Kommunikation und der Synchronisation zwischen den Prozessen

Parallelisierungsprozess

Ausgangspunkt ist ein sequentielles Programm

1. **Dekomposition** (oder Aufteilung) der Berechnung in Tasks
2. **Zuweisung** der Tasks zu Prozessen
3. **Festlegung** des notewendigen Datenzugriffs, der Kommunikation und der Synchronisation zwischen den Prozessen
4. **Abbildung** der Prozesse an die Prozessoren

- Dekomposition und Zuweisung werden zusammen auch als **Partitionierung** bezeichnet

Parallelisierungsprozess

Ausgangspunkt ist ein sequentielles Programm

1. **Dekomposition** (oder Aufteilung) der Berechnung in Tasks
2. **Zuweisung** der Tasks zu Prozessen
3. **Festlegung** des notewendigen Datenzugriffs, der Kommunikation und der Synchronisation zwischen den Prozessen
4. **Abbildung** der Prozesse an die Prozessoren

- Dekomposition und Zuweisung werden zusammen auch als **Partitionierung** bezeichnet

$\Rightarrow$ Task / Thread $\leftrightarrow$ Prozess $\leftrightarrow$ Programm $\leftrightarrow$ Prozessor

$\Rightarrow$ Beispiel für Parallelisierungsprozess an Hand der OCEAN-Simulation in den Vorlesungsfolien

Parallele Programmierung – Einführung

Programmiermodelle

- Shared-Memory-Programmiermodell
  - Threadbibliotheken (Win32 API, POSIX Threads)
  - Compilerdirektiven (OpenMP)
  - $\Rightarrow$ Multiprozessor mit gemeinsamem Speicher
- Nachrichten-orientiertes Programmiermodell
  - MPI
  - $\Rightarrow$ Multiprozessor mit verteiltem Speicher
- Datenparalleles Programmiermodell
  - High Performance Fortran
  - $\Rightarrow$ Vektorprozessoren,...

Parallele Programmierung - Beispiel π-Berechnung

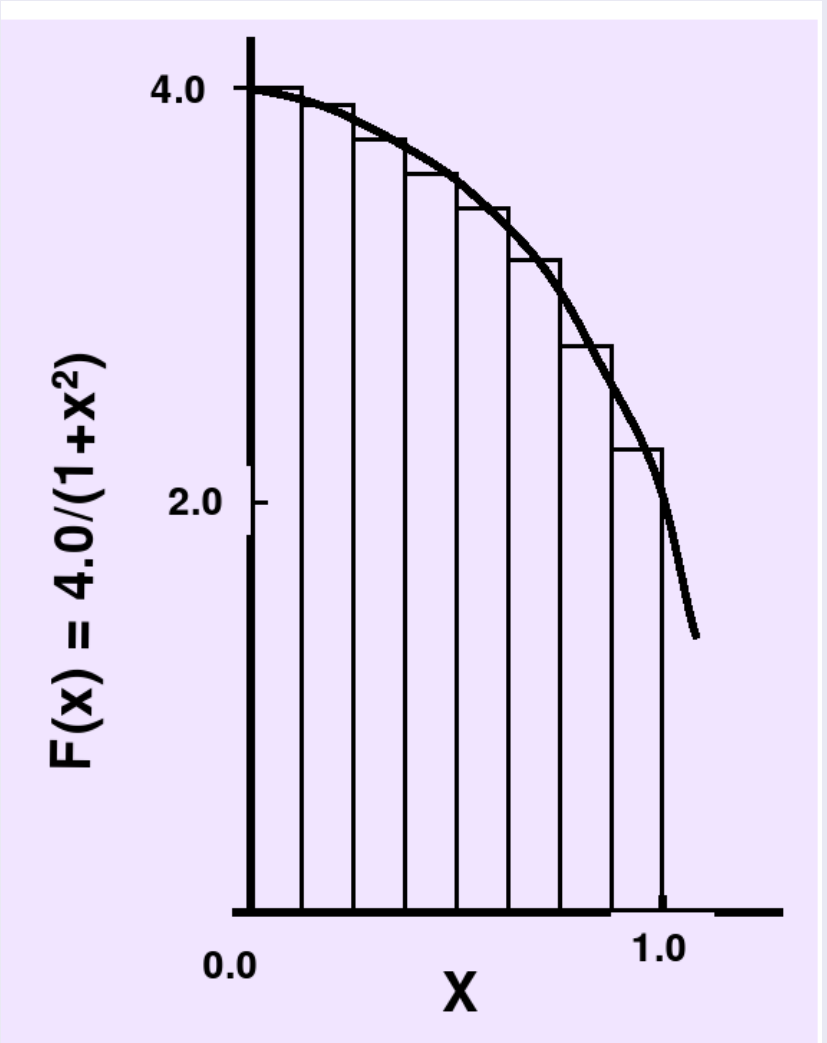
Numerische Integration

- Wir wissen:

$$\int_0^1 \frac{4.0}{(1+x^2)} \mathrm{d}x = \pi$$

- Näherung des Integrals durch

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$



Parallele Programmierung - Beispiel π-Berechnung

Sequentielles Programm

```
static long num_steps = 100000;
double step;
void main ()
{       int i;   double x, pi, sum = 0.0;

        step = 1.0/(double) num_steps;

        for (i=1;i<= num_steps; i++){
                x = (i-0.5)*step;
                sum = sum + 4.0/(1.0+x*x);
        }
        pi = step * sum;
}
```

Parallele Programmierung - Einführung

Thread-Programmierung

- Alle Threads eines Prozesses teilen sich Adressraum, Daten, Filehandler,. . .
- Parallele Programme für Shared-Memory-Systeme bestehen aus mehreren Threads
- Threads werden meist vom Betriebsystem verwaltet
- Unterstützung vom Betriebsystem notwendig, z.B. für die Erstellung,. . .
- Vorteil: durch die Thread-Bibliothek erhält man eine detailierte Kontrolle über die Threads
- Nachteil: die Thread-Bibliothek erzwingt, dass man die Kontrolle über die Threads übernimmt

Parallele Programmierung - Beispiel π-Berechnung

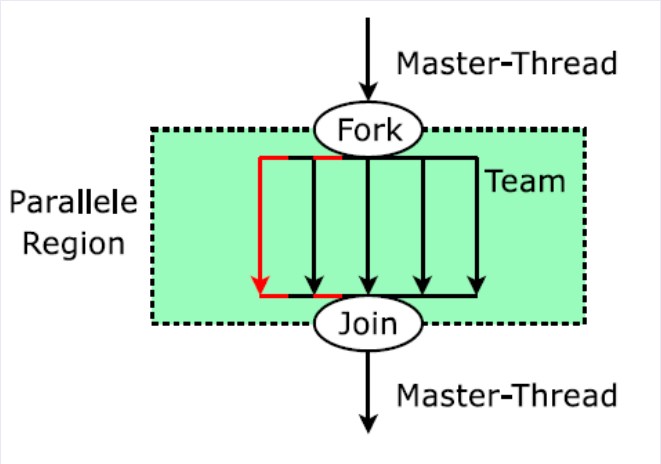
Win32 API

```
#include <windows.h>
#define NUM_THREADS 2
HANDLE thread_handles[NUM_THREADS];
CRITICAL_SECTION hUpdateMutex;
static long num_steps = 100000;
double step;
double global_sum = 0.0;

void Pi (void *arg)
{
   int i, start;
  double x, sum = 0.0;


  start = *(int *) arg;
  step = 1.0/(double) num_steps;

  for (i=start;i<= num_steps;
i=i+NUM_THREADS){
       x = (i-0.5)*step;
      sum = sum + 4.0/(1.0+x*x);
  }
  EnterCriticalSection(&hUpdateMutex);
  global_sum += sum;
  LeaveCriticalSection(&hUpdateMutex);
}
```

```
void main ()
{
  double pi; int i;
  DWORD threadID;
  int threadArg[NUM_THREADS];

  for(i=0; i<NUM_THREADS; i++)   threadArg[i] = i+1;

  InitializeCriticalSection(&hUpdateMutex);

  for (i=0; i<NUM_THREADS; i++){
          thread_handles[i] = CreateThread(0, 0,
            (LPTHREAD_START_ROUTINE) Pi,
                    &threadArg[i], 0, &threadID);
}

  WaitForMultipleObjects(NUM_THREADS,
                    thread_handles, TRUE,INFINITE);

  pi = global_sum * step;

  printf(" pi is %f \n",pi);
}
```

OpenMP

- OpenMP ist eine offene Spezifikation von Übersetzerdirektiven, Bibliotheken und Umgebungsvariablen, spezifiziert für die Parallelisierung von Programmen auf gemeinsamen Speicher
- Verwendet das Join-Fork-Modell



- Mehrere (auch verschaltelte) parallele Regionen pro Programm sind zugelassen

Parallele Programmierung - Beispiel π-Berechnung

OpenMP

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{          int i;    double x, pi, sum = 0.0;
           step = 1.0/(double) num_steps;
           omp_set_num_threads(NUM_THREADS);
#pragma omp parallel for reduction(+:sum) private(x)
           for (i=1;i<= num_steps; i++){
                      x = (i-0.5)*step;
                      sum = sum + 4.0/(1.0+x*x);
           }
           pi = step * sum;
}
```

Parallele Programmierung - Einführung

Message Passing Interface (MPI)

- MPI ist ein Standard für die nachrichtenbasierte Kommunikation in einem Multiprozessorsystem
- Nachrichtenbasierter Ansatz gewährleistet eine gute Skalierbarkeit
- Bibliotheksfunktionen koordinieren die Ausführung von mehreren Prozessen, sowie Verteilung von Daten, per Default keine gemeinsamen Daten
- Single Programm Multiple Data (SPMD) Ansatz

Parallele Programmierung - Beispiel π-Berechnung

MPI

```
#include <mpi.h>
void main (int argc, char *argv[])
{
        int i, my_id, numprocs;  double x, pi, step, sum = 0.0 ;
        step = 1.0/(double) num_steps ;
        MPI_Init(&argc, &argv) ;
        MPI_Comm_Rank(MPI_COMM_WORLD, &my_id) ;
        MPI_Comm_Size(MPI_COMM_WORLD, &numprocs) ;
        my_steps = num_steps/numprocs ;
        for (i=my_id*my_steps; i<(my_id+1)*my_steps ; i++)
        {
                x = (i+0.5)*step;
                sum += 4.0/(1.0+x*x);
        }
        sum *= step ;
        MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                        MPI_COMM_WORLD) ;
}
```

Aufgabe 2 – Parallele Programmierung

Eine Methode aus der Numerischen Mathematik arbeitet auf einem 2D-Torus mit $n * n$ Knoten.

In jeder Iteration werden zwei Schritte durchgeführt:

1. Im ersten Schritt werden die Zustände der Knoten, basierend auf ihrem aktuellen Zustand und den Zuständen ihrer Nachbarknoten, aktualisiert. Dazu müssen zuerst diese Zustände aus dem Speicher gelesen werden.
2. Im zweiten Schritt werden die neuen Zustände zurück in den Speicher geschrieben.

Aufgabe 2 – Parallele Programmierung

Eine Methode aus der Numerischen Mathematik arbeitet auf einem 2D-Torus mit $n * n$ Knoten.

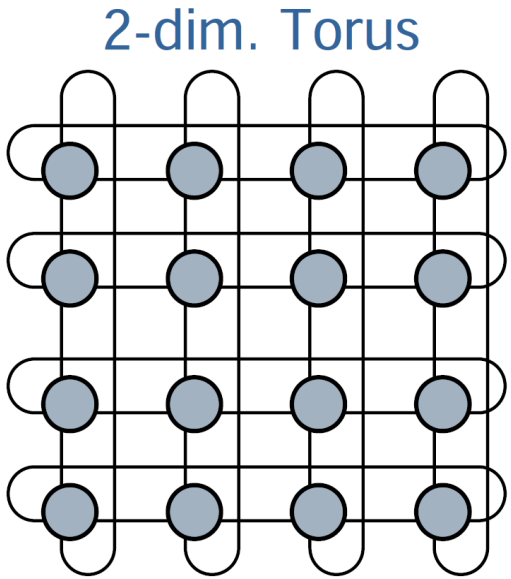
In jeder Iteration werden zwei Schritte durchgeführt:

1. Im ersten Schritt werden die Zustände der Knoten, basierend auf ihrem aktuellen Zustand und den Zuständen ihrer Nachbarknoten, aktualisiert. Dazu müssen zuerst diese Zustände aus dem Speicher gelesen werden.
2. Im zweiten Schritt werden die neuen Zustände zurück in den Speicher geschrieben.

Gegeben sei ein SMP-Knoten mit P Prozessoren.

a) **Berechnen Sie die Beschleunigung der Anwendung bei der Ausführung auf dem SMP-Knoten. Welches Problem tritt hierbei auf?**

Aufgabe 2 – Parallele Programmierung



SMP gemeinsamer Adreßraum, globaler Speicher

$\Rightarrow$ meist **UMA** (Uniform Memory Access), d.h. gleiche Zugriffszeit von allen Knoten

DSM gemeinsamer Adreßraum, physikalisch verteilter Speicher

$\Rightarrow$ meist **NUMA** (Non-Uniform Memory Access), d.h. unterschiedliche Zugriffszeiten

Aufgabe 2 – Parallele Programmierung

a) **Berechnen Sie die Beschleunigung der Anwendung bei der Ausführung auf dem SMP-Knoten. Welches Problem tritt hierbei auf?**

- Sequentielle Zeit $T(1)$:

$$\underbrace{5n^2}_{\text{Daten aus Speicher holen}} + \underbrace{n^2}_{\text{Berechnung}} + \underbrace{n^2}_{\text{Zurückschreiben}} = 7n^2$$

- Parallele Zeit $T(P)$:

$$\underbrace{5n^2}_{\text{Daten aus Speicher holen}} + \underbrace{\frac{n^2}{P}}_{\text{par. Berechnung}} + \underbrace{n^2}_{\text{Zurückschreiben}} = 6n^2 + \frac{n^2}{P}$$

- Beschleunigung $S(P)$:

$$S(P) = \frac{T(1)}{T(P)} = \frac{7n^2}{6n^2 + \frac{n^2}{P}} = \frac{7}{6 + \frac{1}{P}} < \frac{7}{6} \approx 1,17$$

Das Problem ist hierbei, daß der Speicher als limitierender Faktor wirkt und damit die Beschleunigung stark beschränkt ist.

Aufgabe 2 – Parallele Programmierung

Um eine Leistungssteigerung zu erhalten, wird der SMP-Knoten durch eine NUMA-Architektur mit P Prozessoren und P Speichern ersetzt.

Beachten Sie dabei folgende vereinfachende Annahme:

- Erfolgt ein Speicherzugriff auf einen entfernten Speicher, so benötigt ein Speichergriff zwei Zeiteinheiten.

b) **Welche Beschleunigung läßt sich erzielen, wenn die Zustände der Nachbarknoten immer aus entfernten Speichern abgerufen werden müssen?**

Aufgabe 2 – Parallele Programmierung

b) **Welche Beschleunigung läßt sich erzielen, wenn die Zustände der Nachbarknoten immer aus entfernten Speichern abgerufen werden müssen?**

- Sequentielle Zeit $T(1)$:

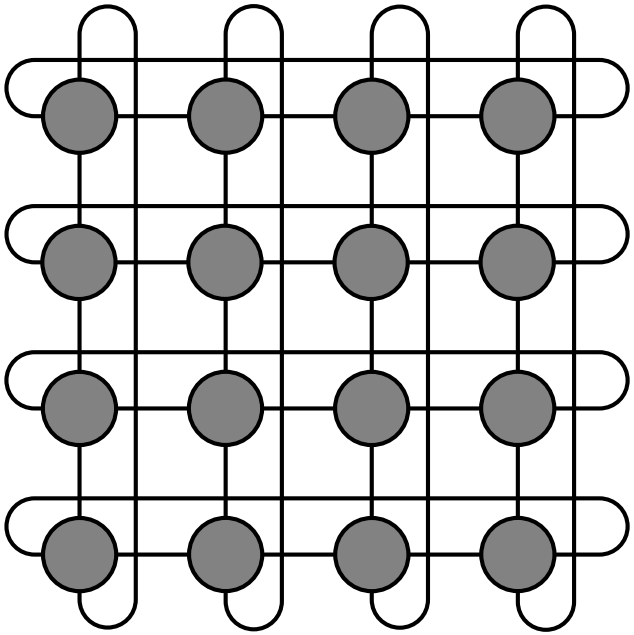
$$\underbrace{5n^2}_{\text{Daten aus Speicher holen}} + \underbrace{n^2}_{\text{Berechnung}} + \underbrace{n^2}_{\text{Zurückschreiben}} = 7n^2$$

- Parallele Zeit $T(P)$:

$$\underbrace{\underbrace{\frac{4*2*n^2}{P}}_{\substack{\text{4 Werte der Nachbarknoten aus entferntem Speicher}}} + \underbrace{\frac{n^2}{P}}_{\substack{\text{eigener Wert im lokalen Speicher}}}}_{\text{Daten aus Speicher holen}} + \underbrace{\frac{n^2}{P}}_{\substack{\text{parallele Berechnung}}} + \underbrace{\frac{n^2}{P}}_{\substack{\text{Zurückschreiben in lokalen Speicher}}} = \frac{11n^2}{P}$$

Aufgabe 2 – Parallele Programmierung

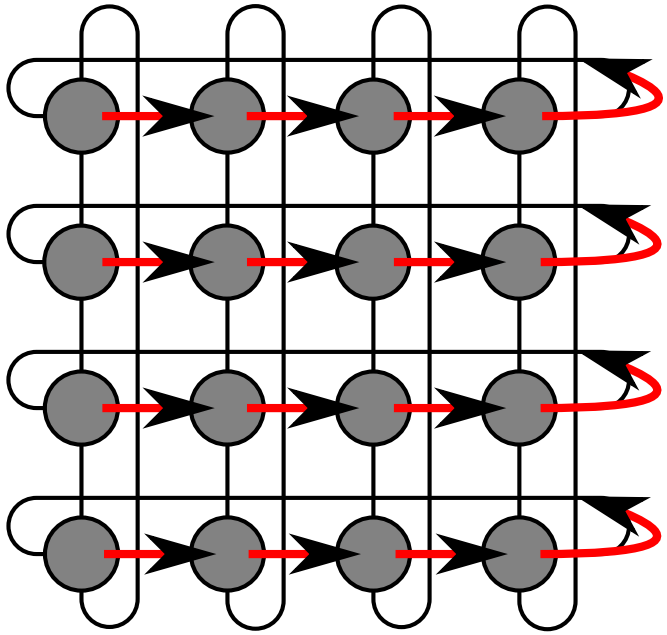
b) **Welche Beschleunigung läßt sich erzielen, wenn die Zustände der Nachbarknoten immer aus entfernten Speichern abgerufen werden müssen?**



- Parallele Zeit $T(P)$ für $n^2 = P$:

Aufgabe 2 – Parallele Programmierung

b) **Welche Beschleunigung läßt sich erzielen, wenn die Zustände der Nachbarknoten immer aus entfernten Speichern abgerufen werden müssen?**

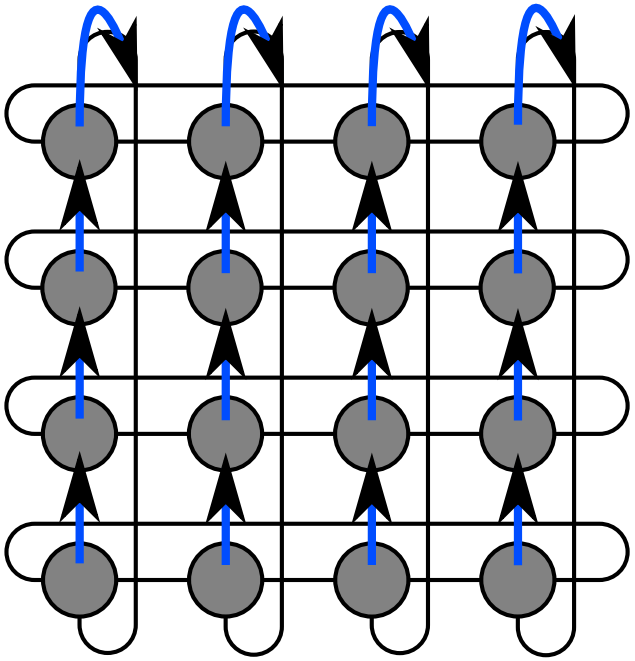


- Parallele Zeit $T(P)$ für $n^2 = P$:

$$2$$

Aufgabe 2 – Parallele Programmierung

b) **Welche Beschleunigung läßt sich erzielen, wenn die Zustände der Nachbarknoten immer aus entfernten Speichern abgerufen werden müssen?**

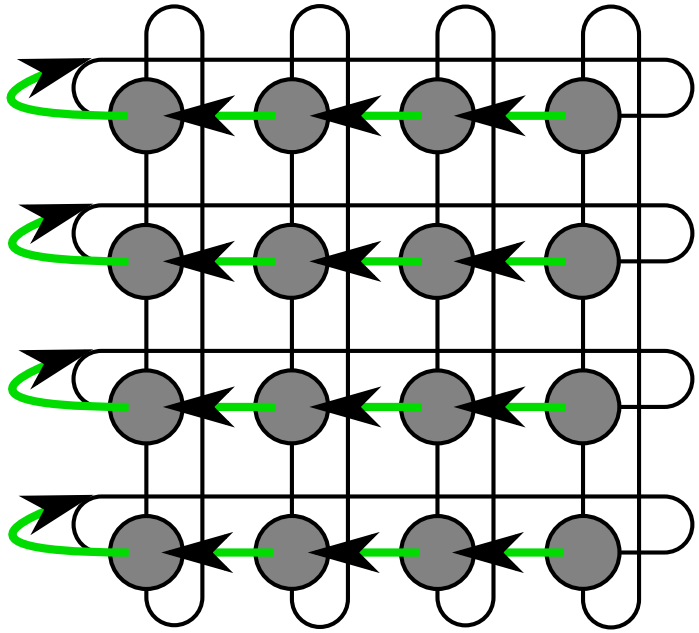


- Parallele Zeit $T(P)$ für $n^2 = P$:

$$2 + 2$$

Aufgabe 2 – Parallele Programmierung

b) **Welche Beschleunigung läßt sich erzielen, wenn die Zustände der Nachbarknoten immer aus entfernten Speichern abgerufen werden müssen?**

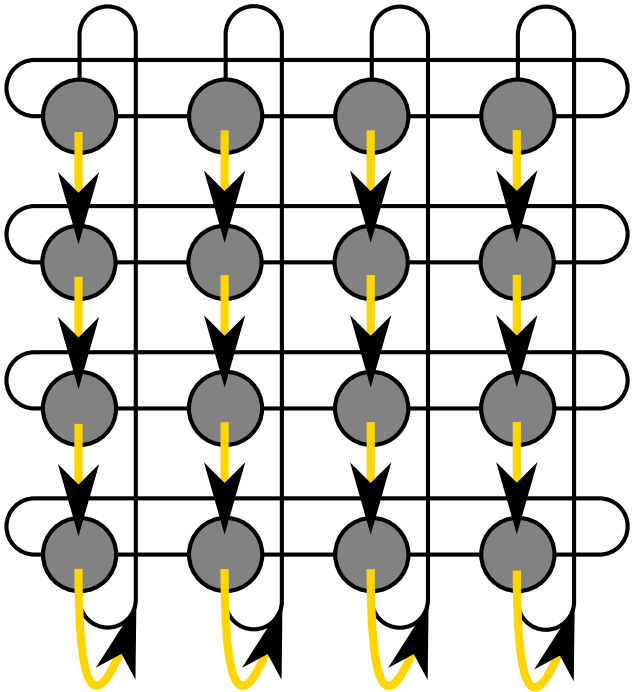


- Parallele Zeit $T(P)$ für $n^2 = P$:

$$2 + 2 + 2$$

Aufgabe 2 – Parallele Programmierung

b) **Welche Beschleunigung läßt sich erzielen, wenn die Zustände der Nachbarknoten immer aus entfernten Speichern abgerufen werden müssen?**

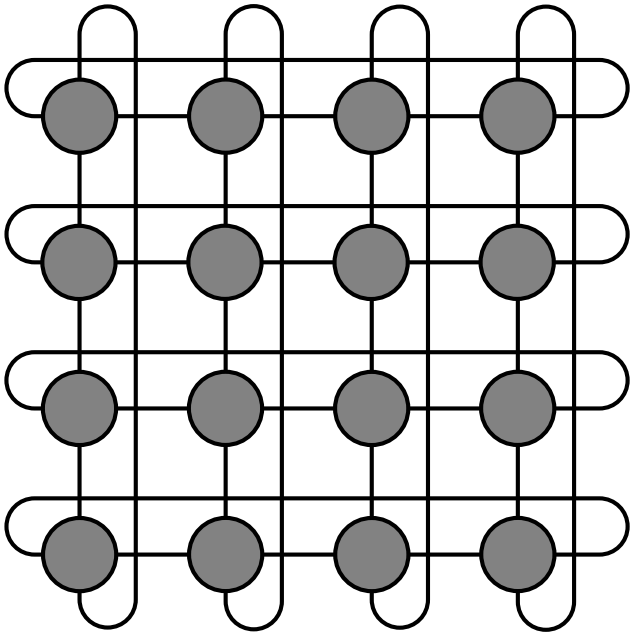


- Parallele Zeit $T(P)$ für $n^2 = P$:

$$2 + 2 + 2 + 2$$

Aufgabe 2 – Parallele Programmierung

b) **Welche Beschleunigung läßt sich erzielen, wenn die Zustände der Nachbarknoten immer aus entfernten Speichern abgerufen werden müssen?**

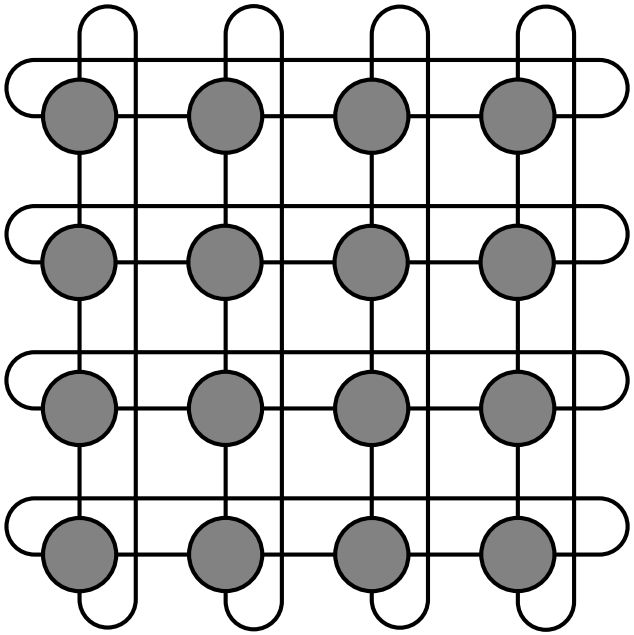


- Parallele Zeit $T(P)$ für $n^2 = P$:

$$2 + 2 + 2 + 2 = 4 * 2$$

Aufgabe 2 – Parallele Programmierung

b) **Welche Beschleunigung läßt sich erzielen, wenn die Zustände der Nachbarknoten immer aus entfernten Speichern abgerufen werden müssen?**

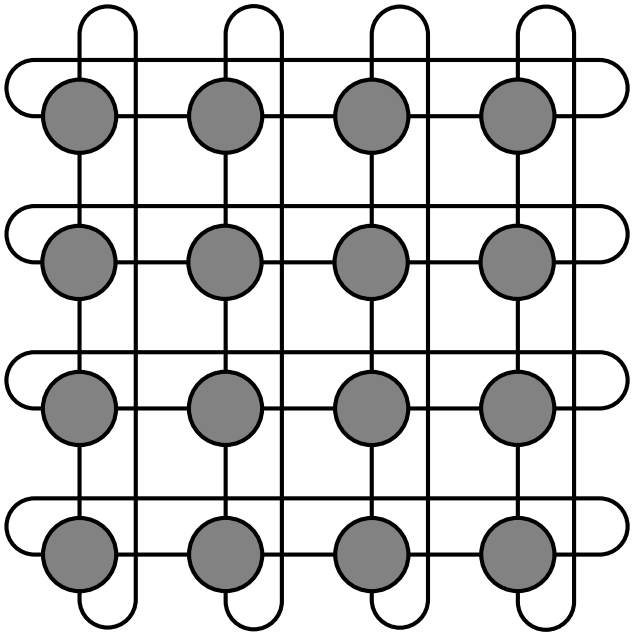


- Parallele Zeit $T(P)$ für $n^2 = P$:

$$T(P) = 4 * 2 + 1 + 1 + 1 = 11$$

Aufgabe 2 – Parallele Programmierung

b) **Welche Beschleunigung läßt sich erzielen, wenn die Zustände der Nachbarknoten immer aus entfernten Speichern abgerufen werden müssen?**

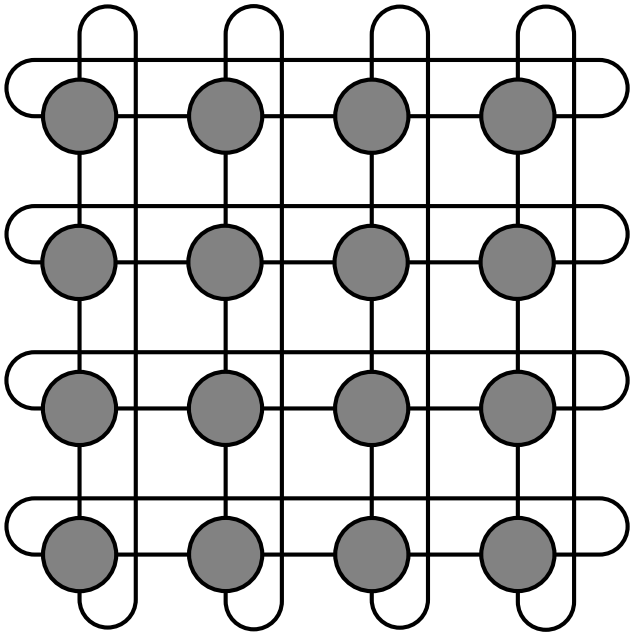


- Parallele Zeit $T(P)$ für $n^2 \neq P$, $n^2 \gg P$:

$$T(P) = (4 * 2 + 1 + 1 + 1) * \frac{n^2}{P}$$

Aufgabe 2 – Parallele Programmierung

b) **Welche Beschleunigung läßt sich erzielen, wenn die Zustände der Nachbarknoten immer aus entfernten Speichern abgerufen werden müssen?**



- Parallele Zeit $T(P)$ für $n^2 \neq P$, $n^2 \gg P$:

$$T(P) = \frac{4 * 2 * n^2}{P} + \frac{n^2}{P} + \frac{n^2}{P} + \frac{n^2}{P} = \frac{11n^2}{P}$$

Aufgabe 2 – Parallele Programmierung

b) **Welche Beschleunigung läßt sich erzielen, wenn die Zustände der Nachbarknoten immer aus entfernten Speichern abgerufen werden müssen?**

- Sequentielle Zeit $T(1)$:

$$T(1) = 5n^2 + n^2 + n^2 = 7n^2$$

- Parallele Zeit $T(P)$:

$$T(P) = \frac{4 * 2 * n^2}{P} + \frac{n^2}{P} + \frac{n^2}{P} + \frac{n^2}{P} = \frac{11n^2}{P}$$

- Beschleunigung $S(P)$:

$$S(P) = \frac{T(1)}{T(P)} = \frac{7n^2}{\frac{11n^2}{P}} = \frac{7}{11}P$$

$$\text{für } P = 2: \quad S(2) = \frac{14}{11} \approx 1,27$$

$$\text{für } P = 3: \quad S(3) = \frac{21}{11} \approx 1,90$$

$\Rightarrow$ Die Beschleunigung skaliert mit $\frac{7}{11}$ der Problemgröße (linear).

Klausur Rechnerstrukturen

Anmeldung zur Klausur am 11.08.2009

- Die Anmeldung zur Klausur ist vom **29.06. bis 02.08.** möglich ist!

$\Rightarrow$ Online über das Studierendenportal

$\Rightarrow$ Anmeldescheine in den orangen Kasten beim schwarzen Brett des Lehrstuhls für Rechnerarchitektur im **1. Stock in Gebäude 20.20**

(. . . wirklich im 1. Stock und nicht beim BIT8000. . .)

Spätere Anmeldungen sind nicht möglich!

- Rücktritt ist bis zum 09.08. elektronisch möglich

Fragen?

Ausblick

Nächste Übung:

- 25. Juni 2009
- Thema: Verbindungsstrukturen