Parallelisierung des Mergesorts

Rituraj Singh

Schlüsselwörter : *Parallel-Mergesort , Sequential-Mergesort , Divide-and-Conquer , OpenMP, C/C++ , Speedup*

Abstrakt

In diesem Dokument haben wir versucht, die Leistung von **Parallel-Mergesort** und **Sequential-Mergesorts** mithilfe OpenMP und Visualstudio Schritt für Schritt zu analysieren.

Inhalt

Einleitung

Merge-Sort ist ein Sortieralgorithmus, der auf dem Divide-and-Conquer-Paradigma basiert.

Beim Divide-and-Conquer-Paradigma **lösen wir ein Problem rekursiv**, wie folgende -

- *Teilen(**Divide**)* - wir das Problem in eine Reihe von Teilproblemen ein, bei denen es sich um kleinere Instanzen der handelt gleiches Problem.
- *Erobern(**Conquer**)* - wir die Teilprobleme, indem Sie rekursiv lösen. Wenn die Teilproblemgrößen sind klein genug, lösen Sie jedoch einfach die Teilprobleme auf einfache Weise
- *Kombinieren(**Combine**)* - wir die Lösungen für die Teilprobleme zu der Lösung für das ursprüngliche Problem

---

Rekursives Mergesort

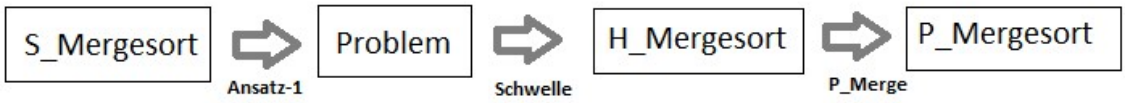
- ***rekursive Merge-Sort-Funktion*** - Diese unterteilt jedes Problem in zwei kleinere Teilprobleme, links und rechts. Diese beiden Teilprobleme sind völlig unabhängig voneinander
- ***Merge-Routine*** - die sortiert und kombiniert schließlich diese beiden Teilprobleme

---

Hinweise zum Code	
A	A ist das Array,an dem wir arbeiten
erstell_A	Wir haben eine Dienstprogrammfunktion (**erstell_A**) erstellen, die ein Array A der Größe A_size (vom Benutzer zur Laufzeit angegeben) mit Zufallszahlen generiert. Die Zahlen im Array A liegen im Bereich **von 0 bis 999**.
copy	Wir verwenden ein anderes Array **copy**, das dieselbe Größe wie A hat, um Zwischenergebnisse des Algorithmus zu speichern
druck_A	eine Dienstprogrammfunktion (**druck_A**) druckt Array A , wenn Größe A kleiner als 50 ist.
Präfixnotation	Entweder Mergesort-Funktion oder Merge-Routine hat **S oder P** als Präfix. **S – Sequentielle Version** **P – Parallele Version**

Notationen und Reihenfolge

Erläuterung		
Art des Mergesorts	***rekursive Merge-Sort-Funktion***	***Merge-Routine***
S_Mergesort	S_Mergesort	S_Merge
H_Mergesort	H_Mergesort	S_Merge
P_Mergesort	P_Mergesort	P_Merge



Implementierung des S_Mergesorts

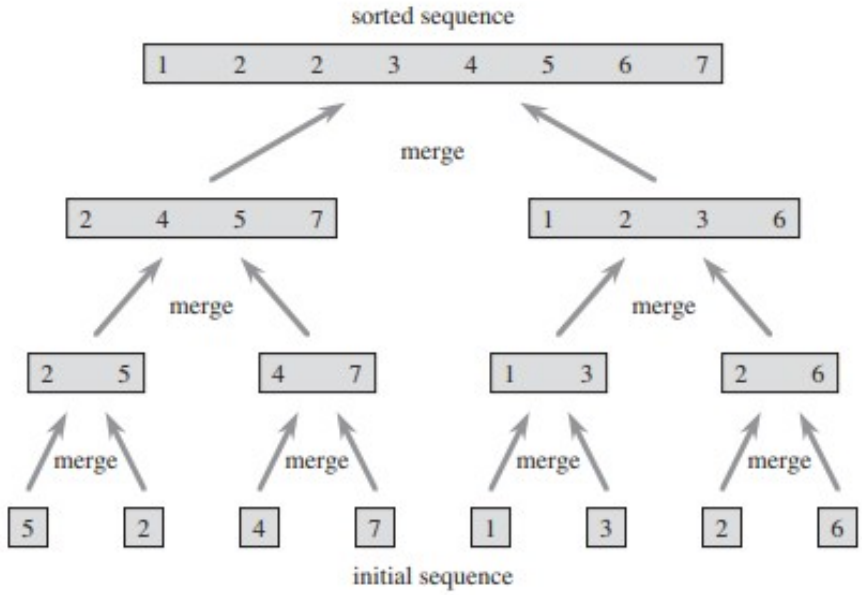
Wir wiederholen den linken und rechten Teil des Arrays so lange, bis Index **low < high**

```
void S_MergeSort(int* to, int* temp, int low, int high)
{
  if (low >= high)
    return;
  int mid = (low + high) / 2;
  S_MergeSort(temp, to, low, mid);
  S_MergeSort(temp, to, mid + 1, high);
  S_Merge(to, temp, low, mid, mid + 1, high, low);

}
```

S_MergeSort sieht so aus.

- **S_Merge** war eine einfache merge-routine, die eine ***lineare Zeit*** benötigt, um die beiden Teilprobleme zusammenzuführen.
- **S_merge** nimmt an, dass die Arrays, die zusammengeführt werden sollen, nebeneinander liegen.



S_Merge funktioniert wie in der obigen Abbildung beschrieben.

Laufzeitmessung von S_Mergesort

Die von **S_MergeSort** benötigte Zeit wurde notiert.Die folgenden Ergebnisse wurden gefunden –

Größe von A.	Zeit in Sekunden
10.000.000	11,34
20.000.000	24,62
30.000.000	40,04

(bei Ausführung auf einem Intel X64-Rechner mit 4 physischen Kernen)

Grund zur Parallelisierung des S_Mergesorts

Wenn wir auf die Methode des sequentielles **S_MergeSort** achten, gibt es links und rechts zwei Unterprobleme, diese beiden Teilprobleme ***völlig unabhängig voneinander sind***. Wir können beide Teilprobleme parallelisieren, damit sie parallel ausgeführt werden können.

Ansatz-1

Um zu parallelisiern verwenden Wir den Befehl **#pragma omp parallel sections** in **S_MergeSort** funcktion wie folgte, damit ***die zwei Abschnitte parallel ausfuhren*** konnen –

```
void S_MergeSort(int* to, int* temp, int low, int high)
{
   if (low >= high)
      return;
   int mid = (low + high) / 2;

#pragma omp parallel sections
   {
#pragma omp section
  S_MergeSort(temp, to, low, mid);
#pragma omp section
   S_MergeSort(temp, to, mid + 1, high);
   }
S_Merge(to, temp, low, mid, mid + 1, high, low);
}
```

S_MergeSort sieht so aus

Hinweis **: Merge-routine ist S_Merge**

Problem : Es wurde festgestellt, dass diese Version des paralleles Merge-sorts mehr Zeit benötigt als die sequentielle Version, anstatt die Laufzeit zu verkürzen.

Grund für dieses Problem

Es wird angenommen, dass zu viel Parallelität der Grund für dieses ineffiziente Verhalten war. Denn die Subprobleme, die noch sehr klein waren, waren auch ebenfalls parallelisiert, anstatt sie nacheinander auszuführen.

Implementierung des **H_MergeSorts**

***Lösung** des obigen Problems besteht darin**, nicht zu viel zu parallelisieren**.*

Wir parallelisieren alle Probleme, die über einer bestimmten Schwelle (**Sequentielle_Schwelle**) liegen, und führen die verbleibenden Probleme nacheinander aus.

```
void H_MergeSort(int* to, int* temp, int low, int high)
{
  if (high - low + 1 <= Sequentielle_Schwelle)
  {
    S_MergeSort(to, temp, low, high);
    return;
  }

  int mid = (low + high) / 2;

#pragma omp parallel sections
  {
#pragma omp section
    H_MergeSort(temp, to, low, mid);
    H_MergeSort(temp, to, mid + 1, high);
  }
  S_Merge(to, temp, low, mid, mid + 1, high, low);
}
```

H_MergeSort sieht so aus

Laufzeitmessung von H_Mergesort

Die folgenden Laufzeiten(in Sekunden) wurden bei den beiden Versionen festgestellt , wenn die **Sequentielle_Schwelle** auf 1024 festgelegt wurde

Größe von A.	H_MergeSort	S_MergeSort
10.000.000	7,35	11,34
20.000.000	15,46	24,62
30.000.000	23,40	40,04

Verbesserung : Im Durchschnitt zeigte **H_MergeSort** eine Verbesserung der Laufzeit auf den **Faktor/Speedup 1,5**

S_Merge als Bottleneck

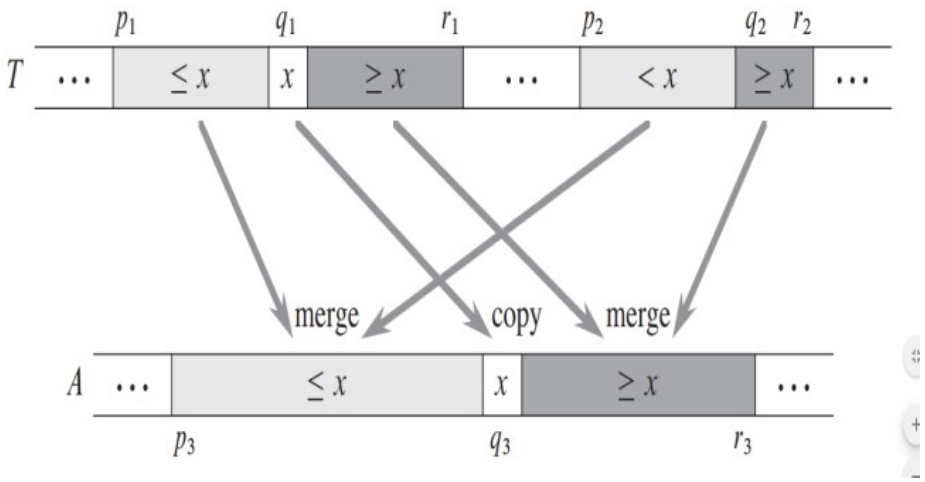
S_Merge, die sequentielle Version der Merge-routine, benötigt eine lineare Zeit, um abgeschlossen zu werden, daher war es wichtig Parallele merge-Routine **P_Merge** zu erstellen.

Implementierung der P_Merge-Routine

P_Merge geht davon aus, dass die beiden Arrays, an denen wir arbeiten, sortiert sind.

Wir führen die beiden sortierten Subarrays zusammen , um ein Pivot-Element, das der **Median Element** im linken Array ist.

- ***Die Zusammenfuhrung ist außerdem in zwei Abschnitten parallel ausgeführt.***
- ***Um Median-Element X zu finden, wird Binary-Search verwendet.***
- ***Binary-Search ist in Rechten Array gemacht nach der Mittel-Element des Linken Array***
- ***P_Merge zusammenführt die Elemente kleiner als X im linken Teil und größer als X im rechten Teil , wie unten gezeigt***



P_Merge funktioniert wie in der obigen Abbildung beschrieben

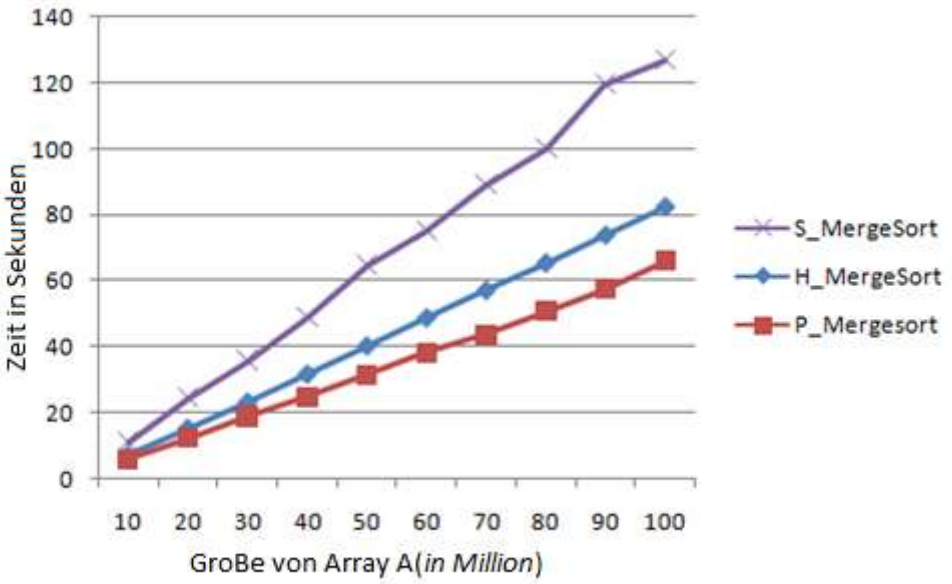
Im Gegensatz zu **S_Merge** geht **P_Merge** jedoch nicht davon aus, dass die beiden Subarrays, sind innerhalb des Arrays benachbart.

Implementierung des P_Mergesorts

Nun **P_MergeSort** hat **P_MergeSort als** Sort-funktion und neu implementierte parallele **P_Merge**-Routine

Laufzeitmessung von P_Mergesort

Die folgenden Laufzeiten(in Sekunden) wurden festgestellt in **x64-Umgebung**



Verbesserung : Im Durchschnitt zeigte **P_MergeSort** eine Verbesserung der Laufzeit auf den **Faktor/Speedup 2.0**

Ergebnisse auf *x64* *gegen* *x86* Laufzeitumgebungen

In Visual Studio haben wir die Möglichkeit, unseren Code auf verschiedenen Architekturen auszuführen.

als die x64 und x86 Umgebungen verglichen wurden. Die folgenden **Laufzeiten(in Sekunden)** wurden festgestellt.

Größe von A.	***S_MergeSort(x64)***	***P_MergeSort(x64)***	***S_MergeSort(x86)***	***P_MergeSort(x86)***
10000000	12.1284	6.66353	11.553	10.0256
20000000	25.1518	13.5204	24.4657	20.4802
30000000	39.3517	20.4603	37.3515	30.9179
40000000	53.8997	27.4094	50.7285	41.0215
50000000	67.7846	34.7764	63.5763	51.5804
60000000	81.5048	42.0005	76.7673	61.7064

- In der **x64-Umgebung** wurde eine Beschleunigung um ***den Faktor 2*** erreciht.
- In **x86-Umgebung** wurde eine auf ***1,2 verringerte Beschleunigung*** bemerkt.

Der ***Hauptgrund*** für dieses Verhalten könnte ein Unterschied in der Art und Weise sein, wie Zahlen in der x86- und x64-Architektur gespeichert werden ***(WORDSIZE)***.

Fazit

- Auf einem Computer mit 4 Kernen und **x64 Umgebung** war es festgestellt, dass **P_Mergesort** mehr als **zweimal** schneller als **S_Mergesort** ist.
- Im Falle einer **x86- Umgebung** , **P_Mergesort** mehr als **1.2** schneller als **S_Mergesort** ist.
- Die Laufzeit eines parallelen Algorithmus hängt nicht nur von der Art und Weise seiner ***Implementierung*** und aber auch von den ***zugrunde liegenden Architekturen*** ab.

Referenzen und Code

- https://gitlab.hochschule-stralsund.de/rituraj.singh/verteilte-programmierung
- **Introduction_to_Algorithms_Third_Edition_(2009)** T.H. Cormen , C.E. Leiserson
- http://people.cs.pitt.edu/~bmills/docs/teaching/cs1645/lecture_par_sort.pdf
- https://www.cs.uky.edu/~jzhang/CS621/chapter7.pdf