VIII PARALELLES UND OBJEKTORIENTIERTES PROGRAMMIEREN

S. BARTELS, 24.7.2018

1. Parallelisierung

Eine Beschleunigung von Algorithmen kann in einigen Fällen durch den Einsatz mehrerer Prozessoren erzielt werden. Soll etwa das Skalarprodukt zweier Vektoren $x, y \in \mathbb{R}^n$ berechnet werden, das heißt die Summe der Produkte der Komponenten

$$x \cdot y = \sum_{j=1}^{n} x_j y_j,$$

so kann die Summe in Teilsummen zerlegt werden, das heißt beispielsweise

$$x \cdot y = \sum_{j=1}^{n_1} x_j y_j + \sum_{j=n_1+1}^{n_2} x_j y_j + \cdots + \sum_{j=n_{p-1}+1}^{n_p} x_j y_j.$$

Die Teilsummen können auf verschiedenen Prozessoren gleichzeitig berechnet und anschließend zusammengefügt werden. Ein solches Vorgehen bezeichnet man als *paralleles Rechnen*. Bei der praktischen Umsetzung müssen verschiedene Aspekte beachtet werden:

- Existiert ein gemeinsamer Speicher, auf den alle Prozessoren zugreifen können, oder müssen die jeweils benötigten Daten an die Prozessoren verschickt werden?
- Greifen die verschiedenen Prozesse auf gemeinsame Variablen zu und ändern diese womöglich, so muss sichergestellt werden, dass dies in geordneter Weise passiert und nicht zu falschen Resultaten führt.

Im obigen Beispiel können Schwierigkeiten vermieden werden, indem Hilfsvariablen eingeführt werden, in denen die Teilsummen abgespeichert werden. Moderne Mehrkernrechner arbeiten mit 4 bis 32 Prozessoren, die effizient auf einen gemeinsamen Speicher zugreifen können, was als *shared-memory*-Architektur bezeichnet wird. Die Anzahl der Teilaufgaben, die auch als Prozesse oder *threads* bezeichnet werden, sollte die Anzahl verfügbarer Prozessoren des verwendeten Computers nicht übersteigen. Die möglichen Schwierigkeiten bei der Parallelisierung von Algorithmen sind vergleichbar mit dem simultanen Zugriff mehrerer Personen auf ein gemeinsames Konto.

1.1. **Open MP.** Die Bibliothek *Open MP* erlaubt eine sehr einfache Parallelisierung von C++-Programmen auf Rechnern mit gemeinsamem Speicher. Dabei steht *MP* für *message passing*, was in diesem Kontext das Versenden und Empfangen der relevanten Informationen an und von den einzelnen Prozessen beschreibt. Zur Realisierung werden mittels sogenannter Compiler-Direktiven, die durch das Symbol # markiert werden, Teile des Programms automatisch verschiedenen Prozessoren zugewiesen. Besondere Sperrmechanismen verhindern unerwünschte Effekte, die durch unkontrolliertes Zugreifen auf gemeinsame Variablen entstehen könnten. Sogenannte Synchronisationsmechanismen fügen private Hilfsvariablen der verschiedenen Prozesse zusammen. Solche Variablen müssen jedoch gesondert definiert werden, da standardmäßig alle Variablen gemeinsame Variablen sind. Eine Übersicht der wichtigsten Befehle in Open MP findet sich in Tabelle 1.

`#include <omp.h>`	Einbinden der Open MP-Bibliothek
`g++ -fopenmp`	Kompilieren eines Programms
`find /usr -name "libgomp*"`	Prüfen der Verfügbarket (unter Unix)
`export OMP_NUM_THREADS=32`	Setzen der maximalen Anzahl von Teilprozessen (unter Unix)
`omp_get_max_threads()`	Anzahl verfügbarer Teilprozesse
`omp_set_num_threads(np)`	Anzahl verwendeter Prozesse
`omp_get_wtime()`	Messen der CPU-Zeit
`#pragma omp parallel for [reduction (op:var)]`	Compiler-Direktive zur Parallelisierung einer for-Schleife

TABELLE 1. Elementare Kommandos zum Arbeiten mit Open MP.

Beispiel 1.1. *Wir betrachten die Berechnung der Summe*

$$s = \sum_{j=1}^{n} j = \frac{1}{2} n(n+1).$$

Eine Realisierung in C++ unter Verwendung der Open MP-Bibliothek ist in Abbildung 1 gezeigt. Dabei erfolgt die Parallelisierung der for-Schleife zunächst mit der Direktive

```
#pragma omp parallel for
```

Dies führt jedoch im Fall mehrerer Prozesse, das heißt im Fall $n_p > 1$, durch unkoordinierte Schreib- und Lesezugriffe auf die gemeinsame Variable `sum` *auf fehlerhafte Ergebnisse. Durch die Verwendung der erweiterten Direktive*

```
#pragma omp parallel for reduction (+:sum)
```

werden in den Prozessen Hilfsvariablen angelegt und nach Abarbeitung der Teilaufgaben zusammengefügt.

```
1  // kompilieren: g++ omp_ex.cc -o omp_ex -fopenmp
2  #include <stdio.h>
3  #include <omp.h>
4
5  int main(){
6    int np = 8, sum = 0, n = 100;
7    omp_set_num_threads(np);
8  #pragma omp parallel for // reduction (+:sum)
9    for (int i=0; i<=n; i++){
10     sum = sum+i;
11   }
12   printf("sum = %d (correct result = %d)\n",sum,n*(n+1)/2);
13 }
```

ABBILDUNG 1. C++-Programm mit paralleler for-Schleife; wird der Zusatz `reduction (+:sum)` auskommentiert, so kommt es zu fehlerhaften Ergebnissen.

1.2. **Skalarprodukt und Vektoraddition.** Besonders effizient ist das Parallelisieren von for-Schleifen bei elementaren Operationen der linearen Algebra wie dem Berechnen eines Skalarprodukts oder einer Linearkombination großer Vektoren. Ein C++-Programm mit Unterroutinen zur Berechnung von

$$x \cdot y, \quad x := ax + by,$$

für Vektoren $x, y \in \mathbb{R}^n$, deren for-Schleifen mittels Befehlen der Open MP-Bibliothek parallelisiert sind, ist in Abbildung 2 gezeigt. Bei der komponentenweisen Berechnung der Linearkombination treten keine gemeinsamen Variablen auf, so dass keine besondere Vorsicht bei der Parallelisierung erforderlich ist. Im Hauptprogramm wird die Anzahl zu verwendender Prozesse über die Variable `np` definiert und ihre Variation erlabut es zu untersuchen, welche Beschleunigung die Verwendung mehrerer Prozesse ermöglicht. Dabei stellt man fest, dass (i) keine weitere Beschleunigung möglich ist, wenn die Rechner-bedingte maximale Anzahl von Prozessoren überschritten wird, und (ii) im Bereich der verfügbaren Prozessoren im Allgemeinen keine optimale Verbessung eintritt, das heißt eine Verdopplung der Prozesse nicht notwendigerweise zu einer Halbierung der Laufzeit führt. Letzteres ist durch sogenannte *Kommunikationskosten*, wie beispielsweise Wartezeiten beim Zugriff auf gemeinsame Variablen, verursacht. Falsche Resultate ergeben sich, wenn der Zusatz `reduction (+:val)` in der Unterroutine `scal_prod` weggelassen wird.

2. OBJEKTORIENTIERUNG

Die Methodik der Objektorientierung erweitert das Konzept von Variablenverbünden. Zusätzlich zur Zusammenfassung von Variablen unterschiedlicher Typen zu einer Variable können Funktionen, sogenannte *Methoden*,

```
// kompilieren: g++ omp_lin_alg.cc -o omp_lin_alg -fopenmp
#include <iostream>
#include <vector>
#include <omp.h>

typedef typename std::vector<double> doubleVec;

double scal_prod(doubleVec &x, doubleVec &y){
  double val = 0.;
  #pragma omp parallel for reduction (+:val)
  for (int j=0; j<(int)x.size(); j++)
    val += x[j]*y[j];
  return val;
}

void update_vec(doubleVec &x, doubleVec &y, double a, double b){
  #pragma omp parallel for
  for (int j=0; j<(int)x.size(); j++)
    x[j] = a*x[j]+b*y[j];
}

int main(){
  int np = 4;
  int N = 1e8;
  double t1, t2, val1, val2, q = .5;
  doubleVec u(N);
  omp_set_num_threads(np);
  u[0] = 1.;
  for (int j=1; j<N; j++)
    u[j] = q*u[j-1];
  t1 = omp_get_wtime();
  val1 = scal_prod(u,u);
  update_vec(u,u,1.,-1.);
  val2 = scal_prod(u,u);
  t2 = omp_get_wtime();
  std::cout << "Ergebnis = [" << val1 << "," << val2 << "] ";
  std::cout << "(korrekt: [4/3,0])\n";
  std::cout << "Anzahl Prozesse = " << np << ", ";
  std::cout << "benoetigte Zeit = " << t2-t1 << std::endl;
}
```

ABBILDUNG 2. Programm mit parallelen Unterroutinen zur Berechnung des Skalarprodukts und Linearkombinationen zweier Vektoren.

auf einem solchen Variablenverbund, der dann als *Klasse* oder *Objekttyp* bezeichnet wird, definiert werden. Eine Variable einer Klasse wird als *Objekt* bezeichnet. Die Motivation dieses Zugangs ist, dass damit die in der

Realität häufig verwendeten Klassifizerungen von Gegenständen oder Lebewesen sinnvoll abgebildet werden können. Beispielsweise gehören Objekte wie Äpfel, Bananen und Birnen zur Klasse Obst und lassen sich mit ähnlichen Kenndaten oder Attributen wie Farbe, Geschmack und Gewicht beschreiben. Ein wesentliches Ziel des objektorientierten Programmierens, das die klassischen Konzepte des Programmierens erweitert, ist die Wiederverwendbarkeit und bessere Strukturierung aufwändiger Programme. Klassen können genutzt werden, ohne deren konkrete Implementierung zu kennen. Die Ideen des objektorientierten Programmierens werden häufig mit den Begriffen *Abstraktion* und *Kapselung* verbunden.

2.1. **Objekte und Klassen.** Die Definition einer Klasse erfolgt mit dem Schlüsselwort `class`. Innerhalb der Klasse werden Variablen und Funktionen, die in diesem Kontext auch als Attribute und Methoden bezeichnet werden, definiert:

```
class KlassenName {
  typ1 var1;
  typ2 var2;
  ...
  val_typ1 methode1(arg_typ1 arg1, ...){
    ...
  }
  ...
}
```

Eine Variable beziehungsweise ein Objekt dieser Klasse wird dann wie eine gewöhnliche Variable deklariert:

```
KlassenName obj_name;
```

Der Zugriff auf die Attribute und Methoden des Objekts erfolgt wie bei Variablenverbünden mit dem Punktoperator:

```
obj_name.var1 = val1;
res = obj_name.methode1(arg1,...,argn);
```

Dabei ist jedoch zu beachten, dass der Zugriff auf die Attribute und Methoden im Allgemeinen nur innerhalb der Klassendefinition zulässig ist. Der Zugriff aus anderen Programmteilen ist möglich, wenn der Zugriffsmodifikator `public` verwendet wird. Die standardmäßig verwendete Zugriffsdeklaration `private` sollte zur Verbesserung der Übersichtlichkeit verwendet werden. Eine weitere Option ist `protected`, die bei der sogenannten Vererbung von Klassen relevant ist. Eine Übersicht der Zugriffsmodifikatoren findet sich in Tabelle 2, ihre Verwendung ist aus dem in Abbildung 3 gezeigten Beispielprogramm ersichtlich.

Beispiel 2.1. *Das in Abbildung 3 gezeigte Programm definiert eine Klasse BankKonto, mit Methoden* `eroeffnen`, `abfrage` *und* `buchung` *sowie Attributen* `kontonummer` *und* `kontostand`. *Auf die Attribute kann nur innerhalb der Klassendefinition zugegriffen werden. Der Wert der Variable* `kontostand`

`public`	Zugriff aus allen Programmteilen
`private`	Zugriff nur innerhalb der Klassendefinition
`protected`	Zugriff nur innerhalb der Klassendefinition und erbenden Klassen

TABELLE 2. Zugriffsmodifikatoren bei der Definition von Klassen.

```
1  // kompilieren: g++ klassen_bsp.cc -o klassen_bsp
2  #include <iostream>
3
4  class BankKonto{
5    public:
6      void eroeffnen(int num, double anf_betrag){
7        kontonummer = num;
8        kontostand = anf_betrag;
9      }
10     double abfrage(){
11       return kontostand;
12     }
13     void buchung(double betrag){
14       kontostand += betrag;
15     }
16   private:
17     int kontonummer;
18     double kontostand;
19 };
20
21 int main(){
22   BankKonto konto_albert;
23   konto_albert.eroeffnen(123456,72.15);
24   konto_albert.buchung(-7.10);
25   std::cout << konto_albert.abfrage() << std::endl;
26   // nicht zulaessig ist Zugriff auf Variable kontostand:
27   // std::cout << konto_albert.kontostand << std::endl;
28 }
```

ABBILDUNG 3. Definition einer Klasse mit von außen erreichbaren Methoden aber nicht direkt von außen erreichbaren Variablen.

kann jedoch über die von außen verfügbare Methode `abfrage` *bestimmt werden.*

Die Möglichkeit der Unterscheidung privater und öffentlicher Attribute und Methoden gibt dem Programmierer einer Klasse mehr Freiheit bei deren konkreter Umsetzung. Für die Verwendung der Klasse ist lediglich ihre Funktionalität relevant.

2.2. **Konstruktoren.** Konstruktoren sind spezielle Methoden einer Klasse, die die direkte Initialisierung eines Objekts bei seiner Deklaration erlauben. Der Name dieser Methode stimmt mit dem Klassennamen überein und sie besitzt keinen Rückgabewert auch nicht den leeren Rückgabewert `void`. Konstruktoren sollten außerhalb der Klassendefinition verfügbar sein, das heißt unter dem Zugriffsmodifikator `public` definiert werden. Im obigen Beispiel 2.1 könnte folgende Methode hinzugefügt werden, die den Konstruktor der Klasse realisiert:

```
class BankKonto{
  public:
    BankKonto(int num, double anf_betrag){
      kontonummer = num;
      kontostand = anf_betrag;
    }
    void eroeffnen(int num, double anf_betrag){
      ...
    }
    ...
};
```

Die gleichzeitige Deklaration und Initialisierung des Objekts `konto_albert` im obigen Beispiel erfolgt dann mittels der Anweisung

```
BankKonto konto_albert(123456,72.15);
```

Die Verwendung der Methode `eroeffnen` in dem in Abbildung 3 gezeigten Beispielprogramm ist damit überflüssig. Destruktoren sind Methoden, die beim Löschen eines Objekts oder dem Beenden eines Programms aufgerufen werden. Ihr Name entspricht dem Klassennamen mit einem vorangestellten Tilde-Symbol `~`, im obigen Beispiel also `~Bankkonto()`; damit wird insbesondere die unnötige Blockierung von Speicherplatz vermieden.

2.3. **Templates.** Die Verwendung von *Templates* beziehungsweise Schablonen erlaubt die variable Definition von Funktionen und Klassen für unterschiedliche Variablentypen, das heißt beispielsweise dass der Typ einer Variable der Funktion oder eines Attributs einer Klasse nicht bei deren Definition festgelegt werden muss. Ein oder mehrere variable Datentypen werden dann symbolisch durch die Voranstellung der Anweisung

```
template <class T1, class T2, ..., class Tp>
```

verwendet. Der konkrete Variablentyp wird beim Aufruf der Funktion implizit oder bei der Deklaration eines Objekts explizit spezifiziert und muss die erforderliche Funktionalität bereitstellen. Die in Abbildung 4 gezeigte Funktion bestimmt das Maximum zweier Variablen gleichen Typs, für den eine Vergleichsoperation `<` definiert ist. Beispiele dafür sind die Typen `double`, `int` sowie `char`, wobei im letzten Fall der lexikographische Vergleich verwendet wird.

```
1  // compile: g++ template_funktion.cc -o template_funktion
2  #include <iostream>
3
4  template <class T>
5  T maximum(T x, T y){
6    if (x<y)
7      return y;
8    else
9      return x;
10 }
11
12 int main(){
13   int i1 = 3, i2 = 2;
14   double d1 = 3.0, d2 = 4.15;
15   char c1 = 's', c2 = 'a';
16   std::cout << maximum(i1,i2) << std::endl;
17   std::cout << maximum(d1,d2) << std::endl;
18   std::cout << maximum(c1,c2) << std::endl;
19 }
```

ABBILDUNG 4. Definition einer Funktion unter Verwendung eines variablen Datentyps.

Ein Beispiel für die Verwendung variabler Datentypen bei der Definition einer Klasse ist die Klasse `std::vector` zur Verwendung von Vektoren variabler Länge. Diese erlaubt die Deklaration von Vektoren mit Einträgen allgemeiner Datentypen beispielsweise mittels

```
std::vector<double> v;
```

Einige Methoden der Klasse sind die Funktionen `push_back` und `size`, zum Anhängen von Einträgen und Bestimmen der Länge, sowie der Zugriff auf Einträge mittels eckiger Klammern wie etwa durch `v[5]`. Der Konstruktor der Klasse erlaubt die Spezifikation der Länge und der Einträge. Mit dem Kommando

```
std::vector<char> w(10,'z');
```

wird ein Vektor `w` deklariert und initialisiert, der zehn Einträge besitzt, die jeweils durch den Buchstaben `z` gegeben sind.