→ Vorstellung typischer Programmiermodelle für die Parallelverarbeitung inkl. entspr. Programmierumgebungen und -sprachen

3.1 Einführung

3.2 Programmiermodelle für gemeinsamen Speicher
parallelisierende Compiler, Threads, OpenMP, Unified C, ...

3.3 Programmiermodelle für verteilten Speicher
MPI, PVM, Occam, Datenparallelität, TCP/IP, RPC, Corba, ...

3.4 GPU-Programmierung

3.5. Hybride Programmiermodelle

3.1 Einführung (I)

- Entwicklung paralleler Programme orientiert sich stark an dem zugrunde liegenden parallelen Rechnersystem
  (RS = HW + BS + Compiler + Laufzeitbibliotheken + ...)

- die gleiche HW kann mit verschied. SystemSW zu versch. RS führen
  z.B. HW xyz + Thread-Bibliothek → RS mit gemeins. Speicher
  HW xyz + MesagePassing-Bibliothek → RS mit verteilt. Speicher

- dadurch können parallele Programme für denselben seq. Algorithmus für versch. RS sehr unterschiedlich sein

- Ziel bei Entwicklung: weitgehende HW-Unabhängigkeit,
  d.h. Lösung für eine ganze Klasse von RS, z.B.:
  - für RS mit gemeinsamem Speicher
  - für RS mit verteiltem Speicher

  →dafür jeweils typische Programmiermodelle

3.1 Einführung (II)

Programmiermodelle

- beschreiben ein paralleles RS aus Sicht einer Programmier-Umgebung (bzw. -sprache)
- entspricht Sicht des Programmierers auf das parallele RS
- möglichst einfach mit gewissem Abstraktionsgrad
- aber für den realen Rechner trotzdem möglichst „passend“, so dass eine hinreichend effiziente Implementierung möglich ist

3.1 Einführung (III)

Kriterien paralleler Programmiermodelle

- welche Art (Ebene) der Parallelität kann ausgenutzt werden?
- Ob und wie kann der Programmierer diese Parallelität spezifizieren? (implizite/explizite Parallelität)
- In welcher Form kann der Programmierer die Parallelität spezifizieren? (unabh./kooperierende Tasks/Threads,…)
- Wie erfolgt die Abarbeitung der parallelen Einheiten? (SIMD, MIMD, synchron/asynchron, …)
- Wie erfolgt Informationsaustausch zwischen den parallelen Teilen? (gemeinsamer Speicher, Message Passing)
- Welche Synchronisationsmöglichkeiten gibt es?
- …

→ Praktisch sind zahlreiche Kombinationsmöglichkeiten vorhanden …

3.2 Programmiermodelle für gemeinsamen Speicher

3.2.1 Grundlagen

- Annahme:
  gemeinsam benutzbarer Speicher zur Kommunikation vorhanden
  - entweder physikalisch gegeben
  - oder als verteilter gemeinsamer Speicher (DSM) organisiert

  (siehe später: Architekturen von Parallelrechnern)

- Konsequenz:
  - Konkurrenzsituationen/Wettlaufsituationen der parallelen Prozesse/Threads beim Zugriff auf gemeinsame Daten möglich
  - daher Koordinierungsmittel zum wechselseitigen Ausschluss nötig

Modell kann (ggf. mit Leistungseinbußen) auch für Systeme mit verteilt. Speicher benutzt werden

3.2.1 Grundlagen (II)

- typ. Begriff: PGAS (Partitioned Global Address Space)
- Alle Prozessoren besitzen privaten Speicher, der teilweise privat, aber auch andernteils mit anderen Prozessoren geteilt nutzbar ist
- Vorteil: große Datenmengen, die von den einzelnen Prozessoren nicht lokal gespeichert werden können, passen in die verteilten Speicher vieler Prozessoren (einheitliche Datenbeschreibung bleibt bestehen)
- PGAS-Sprachen unterscheiden zwischen privatem und verteiltem Speicher
- Beispiele: Unified Parallel C, Co-array Fortran (CAF), Fortress, Titanium (Java based), Chapel (Cray), X10 (IBM)

3.2.2 Parallelisierende Compiler (I)

- Programmierer schreibt nur sequentielles Programm, aber der Compiler findet darin die parallelisierbaren Codesequenzen selbst!
- Compiler erzeugt dann parallelen Code für Multithreading-Umgebungen für mehrere Prozessoren
- Grundlage: Datenflussanalyse des Compilers
- Versuch, Datenabhängigkeiten zw. Befehlen zu minimieren, z.B. durch Code-Umordnung, Einführung zusätzl. Variablen typisch z.B. für Schleifen (ggf. ist Reihenfolge der Durchläufe egal)
- datenunabhängige Anweisungen können parallel ausgeführt werden
- durch Codeoptimierung kann z.B. optimale Ablauffolge für Anweisungen ermittelt werden, ebenso optimale Platzierung von Variablen in CPU-Registern

3.2.2 Parallelisierende Compiler (II)

- Vorteile:
  - Programmierer braucht nicht (parallel) „umdenken“
  - Programmierer braucht kein Wissen über die parallele HW
  - Programme lassen sich u.U. leicht portieren
- Nachteile:
  - evtl. wenig Codeeffizienz, da HW-Besonderheiten ggf. nicht berücksichtigt werden können
  - explizit vorhandene Parallelität im Problem wird ggf. erst im sequ. Programm „vernichtet“ und später vom Compiler wieder erzeugt ...
- Beispiel-Sprachen/Compiler: FORTRAN, C

3.2.3 fork/join-Modell

- dynamische Erzeugung paralleler Prozesse aus einem Hauptprozess „Vaterprozess“ + (viele) „Sohn-Prozesse“

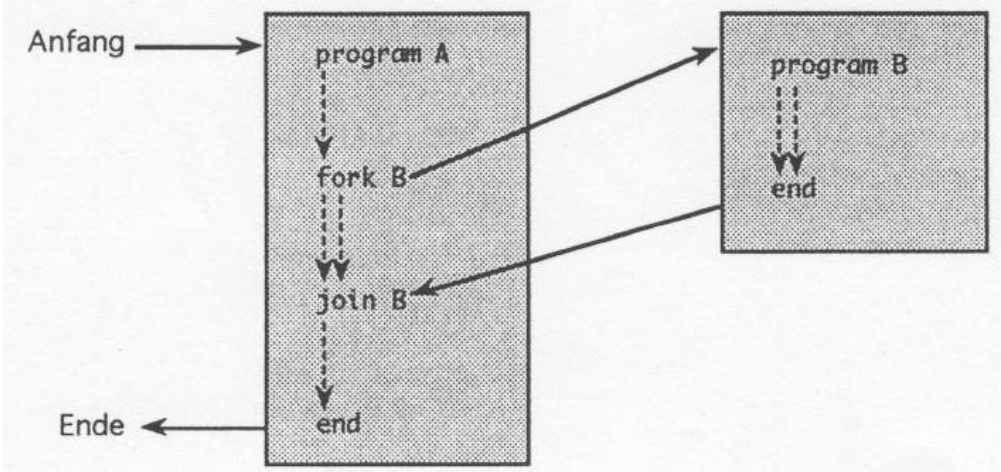


Bild-Quelle:
Bräunl:
Parallele Programmierung.
Vieweg, 1993

- günstig zur Implementierung des Master-Worker-Schemas (Vater verteilt Aufgaben/Daten zur parall. Bearbeitung durch Kinder)
- Bsp.: UNIX/Linux (fork/wait), dazu ggf. expl. Organisation von gemeins. Speicher sowie Nutzung von Semaphoren zum WA, ebenso Mittel zum Datenaustausch vorhanden (Pipes, Mess. Queues)

3.2.4 Threads (I)

- gehören jeweils zu einem Prozess
- benutzen Code- und Datenbereiche des Prozesses sowie weitere Ressourcen gemeinsam
- besitzen individuell nur Register (inkl.Programmzähler) und Stack, daher schnellere Umschaltung Kontextwechsel) mögl.
- sind ideale Basiseinheit zur CPU-Nutzung, daher ist dies das klassische Modell zu MIMD-Systemen
- explizite Synchronisation des Datenzugriffs nötig!
  sie basiert ursprünglich auf dem Monitor-Konzept
  Monitor = Objekt, enthält gemeinsame Daten, Funktionen zum (wechselseit. ausgeschlossenen) Zugriff darauf sowie Wartelisten, praktische Umsetzung z.B. durch Bedingungsvariablen, Semaphore/Mutexe, u.ä. unter Verantwortung des Programmierers

3.2.4 Threads (II)

- verschied. Implementierungsmöglichkeiten
  User-Level-Thr. ←→ Kernel-Level-Threads
  mit entspr. Zuordnungen
  und daraus resultierenden Zuteilungsmöglichkeiten für die CPUs
  (vgl. Kap. 5.3), z.B. M:N-Zuordnung
- wird heute durch viele
  BS direkt unterstützt
- alternativ: POSIX-Threads
  (Laufzeitbibliothek)
  sowie Integration in
  Programmiersprachen (Java)

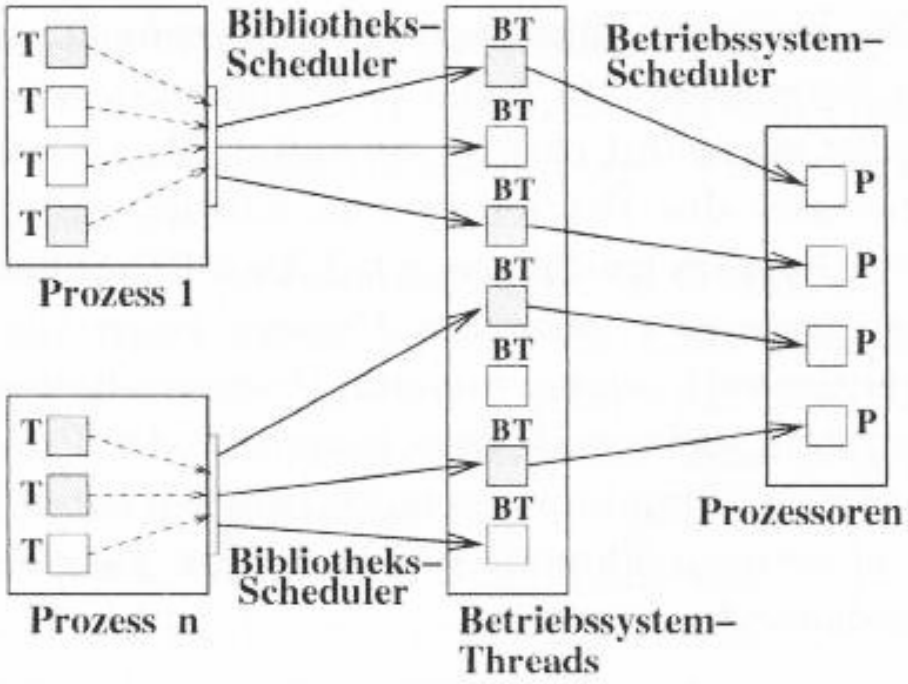


Bild-Quelle: Rauber/Rünger: Multicore: Parallele Programmierung. Springer, 2008

3.2.5 Open Multi-Processing (OpenMP) (I)

- Industrie-Standard-API für Multiprozessoren mit gemeins. Speicher
- www.openmp.org
- für versch. BS (Windows, Linux)
- zur Programmierung in FORTRAN und C/C++
- auch Java-Interface verfügbar (JOMP)
- nutzt Threadbibliotheken und das fork/join-Modell
- durch OpenMP-Compilerdirektiven kann serieller Code durch den Compiler in parallelen Code transformiert werden
  (das Verhalten des seriellen Codes wird nicht verändert, Compiler übernimmt das Thread-Handling zur Ausführung der parallelisierten Teile)
- bisher vor allem für eng gekoppelte Multiprozessoren/Multicores
  Verwendung für Cluster erfordert hybriden Ansatz mit MPI, z.B.
  - OpenMP innerhalb eines (mehrkernigen) Clusterknotens
  - + MPI für die Kommunikation via Verbindungsnetzwerk des Clust.

3.2.5 Open Multi-Processing (OpenMP) (II)

Steuerung der Parallelisierung durch *Compiler-Direktiven (Pragmas):*

- Pragma = spez. Steueranweisung an den Compiler
- Der Programmierer muss damit angeben
  - welche Coderegionen parallel ausgeführt werden sollen
  - wo eine Synchronisation erforderlich ist
- ein Programm mit OpenMP-Direktiven kann übersetzt werden:
  - mit einem OpenMP-Compiler in ein paralleles Programm
  - mit einem Standard-Compiler in ein sequentielles Programm (OpenMP-Direktiven werden ignoriert, Ergebnis des sequ. Programms ist dasselbe, Laufzeit ist aber natürlich höher)

zusätzlich sind in OpenMP verfügbar:

- *Bibliotheksfunktionen bzw. Klassen*
  (z.B. zur Ermittlung der akt. Threadanzahl zur Laufzeit,…)
- *Umgebungsvariablen bzw. Properties*
  (z.B. für Thread-ID, gewünschte Threadanzahl zur Parallelisierung,…)

3.2.5 Open Multi-Processing (OpenMP) (III)

Bsp.: *der folgende Code*

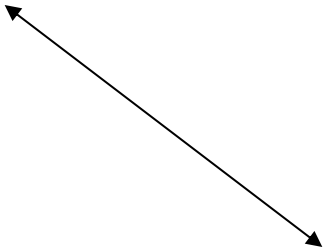
```
main() {
printf("Seriell\n");
#pragma omp parallel
{
printf("Parallel\n");
}
printf("Seriell\n");
}
```

führt nach Compilierung bei Aufruf von

```
$ export OMP_NUM_THREADS=3
$ ./my-omp-prog
```

zu folgender Anzeige:

```
Seriell
Parallel
Parallel
Parallel
Seriell
```



Steuerung durch Compilerdirektiven:
= Mittelweg zw. „vollständig manueller“ und „automatischer“ Parallelisierung

Programmierer legt fest, welche Coderegionen parallel ausgeführt werden können!

Er ist aber auch für die Korrektheit inkl. Synchronisation verantwortlich!

3.2.5 Open Multi-Processing (OpenMP) (IV)

Aufteilung von Schleifeniterationen

- Falls es im Schleifenkörper keine Datenabhängigkeiten gibt, können die einzelnen Durchläufe parallelisiert werden
- Lastverteilung
- mittels **`for`**-Direktive
- am Ende der parallelen Region (Schleife) wieder Barrieren-Synchr.
- Bsp.: Annahme: Parallelisierung für 3 Threads vereinbart

  das Codestück …

```
#pragma omp parallel
#pragma omp for
   for (i=1; i<13; i++)
       c[i]=a[i] + b[i];
```

… teilt dann die Schleifendurchläufe 1-4 dem Thread 1,
die Schleifendurchläufe 5-8 dem Thread 2 und
die Schleifendurchläufe 9-12 dem Thread 3 zu!

3.2.5 Open Multi-Processing (OpenMP) (V)

Weitere Möglichkeiten von OpenMP:

- Parallelisierung von Anweisungen/Blöcken (**`sections`**)
- ausdrücklich keine Parallelisierung von Anweisungen (**`single`**)
- Synchronisationsmittel:
  - kritische Abschnitte
  - Lock-Funktionen
  - Barrieren
- aktuell OpenMP-4.5 (Vektorisierungsdirektiven, Thread affinity...,),
- auch für GPU-Programmierung geeignet (siehe 3.4)

3.2.6 Unified Parallel C (UPC)

- explizit parallele Programmiersprache auf Basis von ISO-C
- Anzahl der parallelen Threads ist statisch festzulegen,
  z.B. beim Compileraufruf als Option **`-Threads 4`**
- alle Threads führen dasselbe UPC-Programm (**`main`**) aus
  = Single Program Multiple Data (SPMD) – Modell
- jeder UPC-Thread kann private und gemeinsame Daten haben
- Lastverteilung auf Threads durch Parallelisierung von unabhängigen Iterationen einer Zählschleife möglich (**`upc_forall`**)
- Sperrfunktionen zur Synchronisation mittels Lock-Variablen sowie Barrieren

3.2.7 Fortress

- Open Source Forschungsprojekt von Sun zur Entwicklung einer modernen Nachfolgesprache für FORTRAN (= klassische Sprache für numerisches Hochleistungsrechnen), aber nicht kompatibel dazu
- *„To do for FORTRAN what Java did for C“* (G.L.Stelle jr.)
- neue Syntax, orientiert sich nahe an mathematischen Formeln z.B. Verwendung von math. Symbolen wie α, Ω direkt im Quellcode
- verschiedene Sprachkonstrukte sind bereits implizit parallel (z.B. werden alle for-Schleifen durch parallel Threads bearbeitet), Programmierer braucht sich um Organisation der Parallelität (durch Threads) nicht zu kümmern
- zusätzl. explizite parallele Konstrukte zum Starten eines Thread (**`spawn`**) und zur Synchronisation (nur wechselseitiger Ausschluss (**`atomic`**)) vorhanden
- Objekte (Felder und Methoden dazu)

3.2.8 MPI-3 shared memory

- MPI = Message Passing Interface
- entwickelt für „Nicht shared memory Systeme“ (siehe 3.3.2)
- ab Version MPI-3 aber shared memory feature (Alternative zu OpenMP)

3.2.9 Weitere Beispiele:

- Cilk (MIT), Cilk Plus (Intel) mit Task-Parallelismus (fork/join)

3.3 Programmiermodelle für verteilten Speicher

3.3.1 Überblick

- *nebenläufige Modelle*
  - nachrichtenbasiert
    - MPI
    - PVM
    - Occam
  - Datenparallelität
    - High Performance Fortran
    - High Performance Java
- *kooperative Modelle*
  - nachrichtenbasiert
    - TCP/IP
    - Java Message Service
  - entfernte Aufrufe
    - RPC
    - CORBA
    - RMI, …

3.3.2 Nachrichtenbasierte nebenläufige Modelle (I)

a) Message Passing Interface MPI (I)

- Kommunikationsbibliothek für C/C++, Fortran, (Java)
- Ende der 1980er Jahre: viele verschiedene Kommunikations-bibliotheken (z.B.: NX, PARMACS, PVM, P4, ...)
  dadurch parallele Programme schwer portierbar
- Festlegung eines Quasi-Standards durch MPI-Forum (ab 1994)
  mit MPI 2.0 (1997) und aktuell: MPI 3.0 (2012) bzw. 3.1
- http://www.mpi-forum.org
- sehr weit verbreitet, durch IEEE standardisiert
- MPI definiert nur das API (d.h. die Programmier-Schnittstelle)
  daher verschiedene Implementierungen,
  sowohl kommerzielle als auch Open Source,
  z.B. MPICH, LAM/MPI, MPJ, ...
- bietet insbes. auf homogenen PR bzw. Clustern effiziente und schnelle Kommunikation

3.3.2 Nachrichtenbasierte nebenläufige Modelle (II)

a) Message Passing Interface MPI (II)

Einige Eigenschaften von MPI:

- bei MPI-1 feste Zahl von Prozessen für eine MPI-Anwendung, ab MPI-2 und 3 dynamische Erzeugung neuer Prozesse möglich
- API von MPI enthält zahlreiche Fkt. zum Senden/Empfangen v. Nachr.
- MPI-2 hat auch Schnittstelle zu einem allg. parallelen Dateisystem (kann auf vert. Dateisysteme wie z.B. NFS abgebildet werden)
- grundlegende Kommunikation: Punkt-zu-Punkt (1:1), aber auch Broadcast (1:m) und weitere Formen der Gruppen-Kommunikation möglich
- Barrieren zur Synchronisation
- ab MPI-3 auch shared memory Extension, Fortran-Binding

3.3.2 Nachrichtenbasierte nebenläufige Modelle (III)

b) Parallel Virtual Machine (PVM) (I)

- Kommunikationsplattform zur Verbindung von Windows- oder UNIX/Linux-Rechnern zu einem virtuellen Systemverbund bzw. (heterogenen) Cluster
- Entwicklung seit etwa 1989
- nicht vordergründig auf hohe Leistung orientiert
- Bibliotheken für C und Fortran, inzwischen auch für Java (JPVM), Perl und C++ (CPPVM)
- auch Projekte zur Verschmelzung von PVM und MPI (z.B. HARNESS)

3.3.2 Nachrichtenbasierte nebenläufige Modelle (IV)

b) Parallel Virtual Machine (PVM) (II)

Einige Eigenschaften von PVM:

- PVM besteht aus
  - PVM-Dämon-Prozess (`pvmd3`) ist auf jedem Rechner der Steuermodul für die Prozesse
  - Bibliothek (`libpvm3.a`) mit Fkt. zur Taskverwaltung und zum Nachrichtenaustausch
  - PVM-Konsole bietet Kommandos für Installation, Start, Beenden, Überwachen, Anzeigen usw. (auch grafisch: XPVM)
- PVM-Anwendung besteht aus Tasks, die die Parallelisierung erlauben, z.B. in folgenden Formen:
  - jede Task bearbeitet eine andere Funktion (funktionale Zerlegung)
  - jede Task bearbeitet dieselbe Fkt. mit anderen Daten (Datenzerl.)
  - Mischform
- Tasks und auch Rechner können dynamisch hinzugefügt/entfernt werd.

3.3.2 Nachrichtenbasierte nebenläufige Modelle (V)

b) Parallel Virtual Machine (PVM) (III)

- Tasks und auch Rechner können in PVM dynamisch hinzugefügt/ entfernt werden
- Fkt. zum Senden und Empfangen von Signalen
- Fkt. zum Senden von Nachrichten über einen Puffer
- Barrierensynchronisation

3.3.2 Nachrichtenbasierte nebenläufige Modelle (VI)

c) Occam

- ursprünglich von Inmos in de 80er Jahren für den Transputer entwickelte Progr.-sprache
- baut auf „Communicating Sequential Processes (CSP [Hoare]) auf
- Transputer „ausgestorben“, Occam aber noch für einige Plattformen (z.B. Intel) verfügbar, sowie Java-Implementierung für Occam
- einfache blockstrukturierte Sprache mit Funktionen/Prozeduren
- enthält statische parallele Prozesse mit Kommunikationsmöglichkeiten
- `PAR`-Konstrukt zur Parallelisierung mittels fork/join-Modell
- `PLACED`-Konstrukt zur Platzierung eines Progr. auf einem Prozessor

3.3.3 Modelle mit Datenparallelität (I)

a) High Performance Fortran (HPF) (I)

- seit 1993, basiert auf Fortran 90
- Implementierungen sowohl für SIMD*)- als auch MIMD-Systeme
- Hauptanwendung: wissenschaftliches Rechnen
- bessere Ausnutzung der Datenparallelität: parallele Ausführung derselben Befehle über mehreren gleichartigen Datenelementen
- bei HPF: Datenparallelität bzgl. Bearbeitung von Arrays (durch mehrere Prozesse auf verschiedenen CPUs parallelisierbar)
- Quellcode hat einfache sequentielle Semantik
- keine explizite Spezifikation der Kommunikation zwischen den Ausführungseinheiten nötig, Verteilung der Daten auf die Speicher der Prozessoren wird durch Compiler-Direktiven gesteuert (Compiler baut die erforderliche Kommunikation ein, ebenso Datenplatzierung)
- Struktur des Programms hängt z.T. vom Compiler ab

*) **Für SIMD existieren neben Fortran90 noch weitere spez. Sprachen wie z.B. C*, MPL, Parallaxis u.a.**

3.3.3 Modelle mit Datenparallelität (II)

a) High Performance Fortran (HPF) (II)

Bsp. zur Parallelität in HPF

- explizite Parallelität (direkt vom Compiler erkennbar)

```
integer A(m,n), B(m,n), C(m,n)
A = B * C  ! punktweise Multiplikation
```

Voraussetzung: Arrays haben gleiche Form und Größe

- implizite Parallelität (nicht direkt erkennbar)

```
do i = 1,m
   do j = 1,n
      A(i,j) = B(i,j) * C(i,j)
   enddo
enddo
```

benötigt Compileranalyse
wird auf verteilten Rechnern zu einem SPMD-Programm

(Quelle: http://www.infosun.fim.uni-passau.de/cl/passau/ppp2005/hpf.pdf)

3.3.3 Modelle mit Datenparallelität (III)

b) High Performance Java (HPJava)

- Programmierumgebung für wiss.-techn. Rechnen unter Java, als strikte Erweiterung von Java
- unterstützt gemeinsamen und verteilten Speicher
- Datenparallelität ähnlich wie bei HPF für Felder (auch multidimensional)
- www.hpjava.org

- weiteres ähnliches Projekt: Java Grande Forum (JGF) www.javagrande.org

3.3.4 Kooperative Modelle

3.3.4.1 Grundlagen kooperativer Modelle (I)

- Kooperation → Dienstleistungsverhältnis
  Prozesse benutzen Dienst eines anderen Prozesses
  typ. Client-Server-Verhältnis in verteilten Umgebungen
  Server verwaltet das „Gemeinsame“ der verteilten Clients

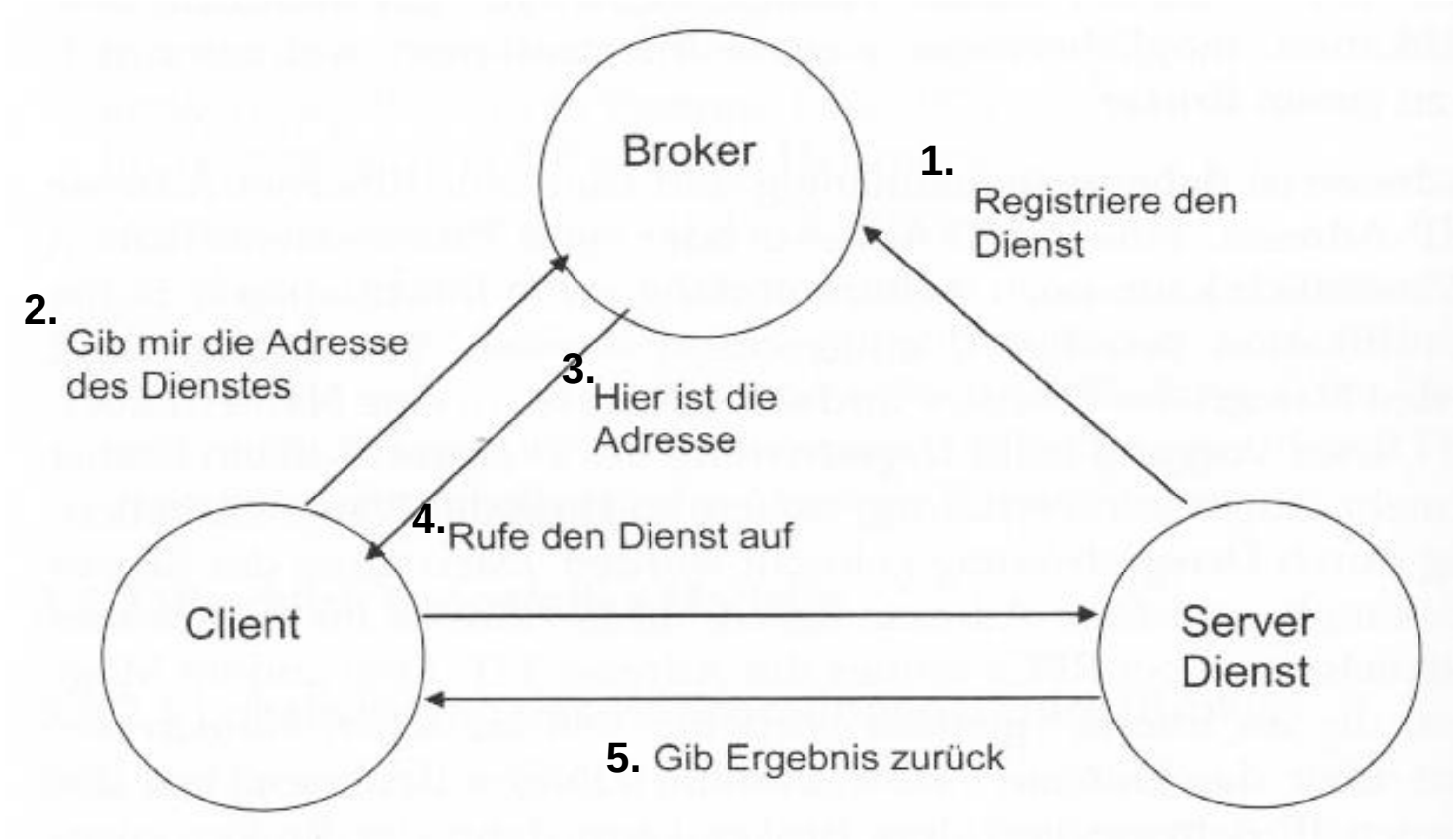
- Voraussetzung zur Kommunikation:
  Lokalisierung des Kommunikationspartners
  - *statisches Binden*, d.h. durch direkte Angabe der Rechneradresse
  - *dynamisches Binden*, d.h. Umwandlung des log. Namens des Koop.-partners in eine Rechneradresse durch einen Broadcast oder über einen Vermittler (Broker)

3.3.4.1 Grundlagen kooperativer Modelle (II)

- statisches Binden
  - bereits zur Übersetzung des aufrufenden Programms
  - falls sich Änderungen beim Koop.-partner ergeben (Wechsel des Rechners, andere Schnittstelle,…), ist Neu-Übersetzung nötig
  - Anwendung ist von konkr. Umgebung/Rechneradressen abhängig
- dynamisches Binden
  - Zu Beginn oder während Programmausführung
  - Zwischenschalten eines Brokers erlaubt einfachere Anpassung an Änderungen beim Koop.-partner ohne Beeinflussung das anderen
  - Unterstützung dyn. Rekonfigurationen und mobiler Partner möglich
  - Server muss sich beim Broker registrieren
  - Broker:
    - muss eindeutig bekannt sein (feste Adresse oder Ermittlung via Domain Name Service DNS)
    - verwaltet Tabelle mit Namen der Dienste und Adressen der zugehör. Server

3.3.4.1 Grundlagen kooperativer Modelle (III)

Allg. Ablaufschema bei kooperativen Modellen:



Quelle: Bengel u.a.: Masterkurs parallele und verteilte Systeme. Vieweg/Teubner 2008 (angepasst)

3.3.4.2 Kooperative nachrichtenbasierte Modelle (I)

a) TCP/IP

Netzwerkprogrammierung unter Nutzung der TCP/IP Sockets mit entspr. API-Fkt. zum synchronisierten Senden und Empfangen sowie Absicherung via Secure Sockets Layer (SSL)

Sockets = „Kommunikationsendpunkt“ zur Herstellung einer Kommunikationsverbindung zwischen (verteilten) Prozessen

- sind de-facto-Standard, werden von allen übl. BS angeboten
- Socket-API wird von zahlreichen Programmiersprachen unterstützt (z.B. C, Java, Python, Perl, Ruby, Tk/Tcl, …)
- Socket-Verbindung ist in beiden Richtungen nutzbar
- Sockets unterstützen neben TCP/IP auch andere Komm.-Protokolle
- verschied. Socket-Typen:
  - Datagram: verbindungslos (unzuverlässig)
  - Stream: verbindungsorientiert
  - Raw: zum direkten Zugriff einer Anwendung auf das Netzwerk-Protokoll
  - Packet: zuverlässiger, paketorientierter Dienst

3.3.4.2 Kooperative nachrichtenbasierte Modelle (II)

b) Java Message Service (JMS) (I)

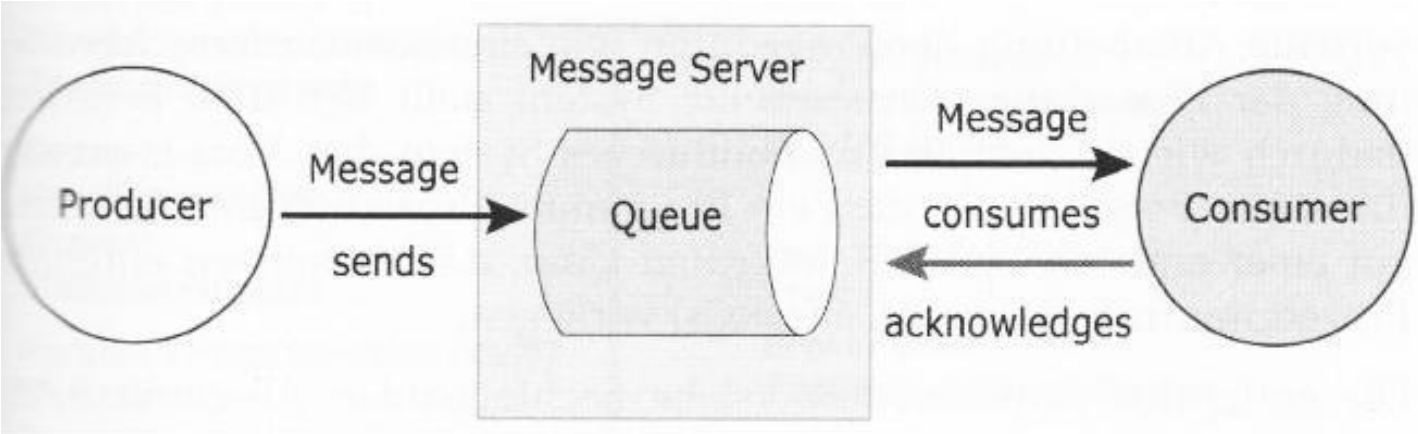
- asynchrones Kommunikationsmodell:
  Sender schickt Nachricht für den Empfänger an einen Message Server, der die Nachricht an den Empfänger weiterleitet, der Sender kann ohne Synchronisation fortfahren
- Sender und Empfänger sind damit nur indirekt gekoppelt
  brauchen nicht die gleiche Technologie verwenden
- in JMS heißt der Message Server „JMS Provider“
- JMS-Applikation besteht aus vielen JMS-Clients und einem Provider
- Komponenten einer JMS-Applikation:
  - Producer = Teil der Applikation, der eine Nachricht erzeugt und versendet
  - Destination = Objekt zur Angabe des Zielortes einer Nachricht
  - Consumer = Teil der Applikation, der eine Nachricht empfängt und verarbeitet

3.3.4.2 Kooperative nachrichtenbasierte Modelle (III)

b) Java Message Service (JMS) (II)

bietet 2 Nachrichtemodelle (Messaging Domains) an:

1.) Point-to-Point (PTP, 1:1)
- Erzeuger verschickt Nachricht über virtuellen Kanal („Queue“)
- nur ein Empfänger kann die Nachricht erhalten
- Empfänger kann Queue vorher inspizieren



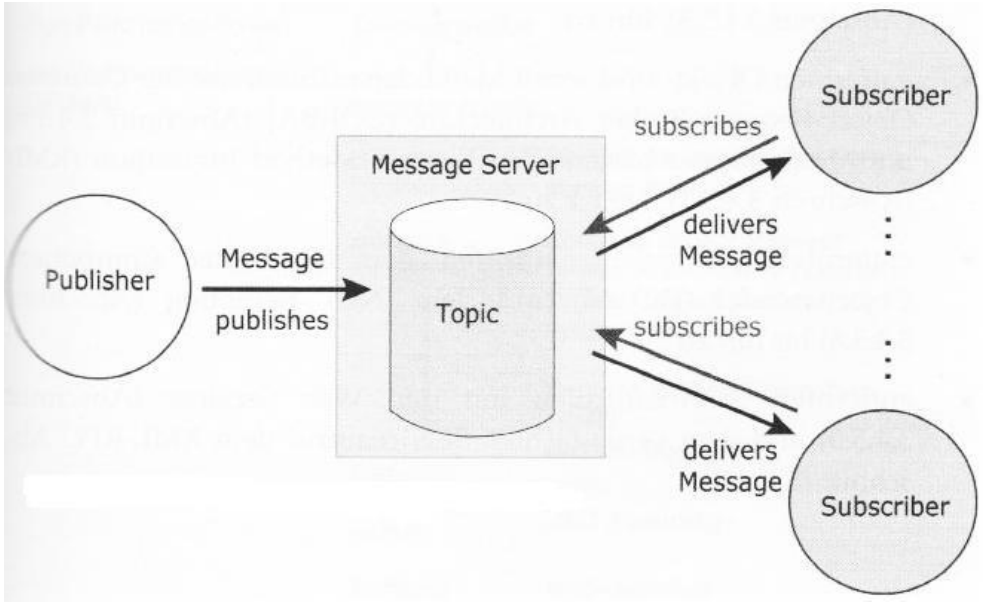
Quelle: Bengel u.a.: Masterkurs parallele und verteilte Systeme. Vieweg/Teubner 2008

3.3.4.2 Kooperative nachrichtenbasierte Modelle (IV)

b) Java Message Service (JMS) (III)

2.) Publish/Subscribe (Pub/Sub, 1:m)

- Erzeuger schickt Nachricht über virt. Kanal („Topic“) = publish
- ein oder mehrere Empfänger können sich zum Topic verbinden (=subscribe) und eine Kopie die Nachricht erhalten



Quelle: Bengel u.a.: Masterkurs parallele und verteilte Systeme. Vieweg/Teubner 2008

3.3.4.3 Kooperative Modelle mit entfernten Aufrufen (I)

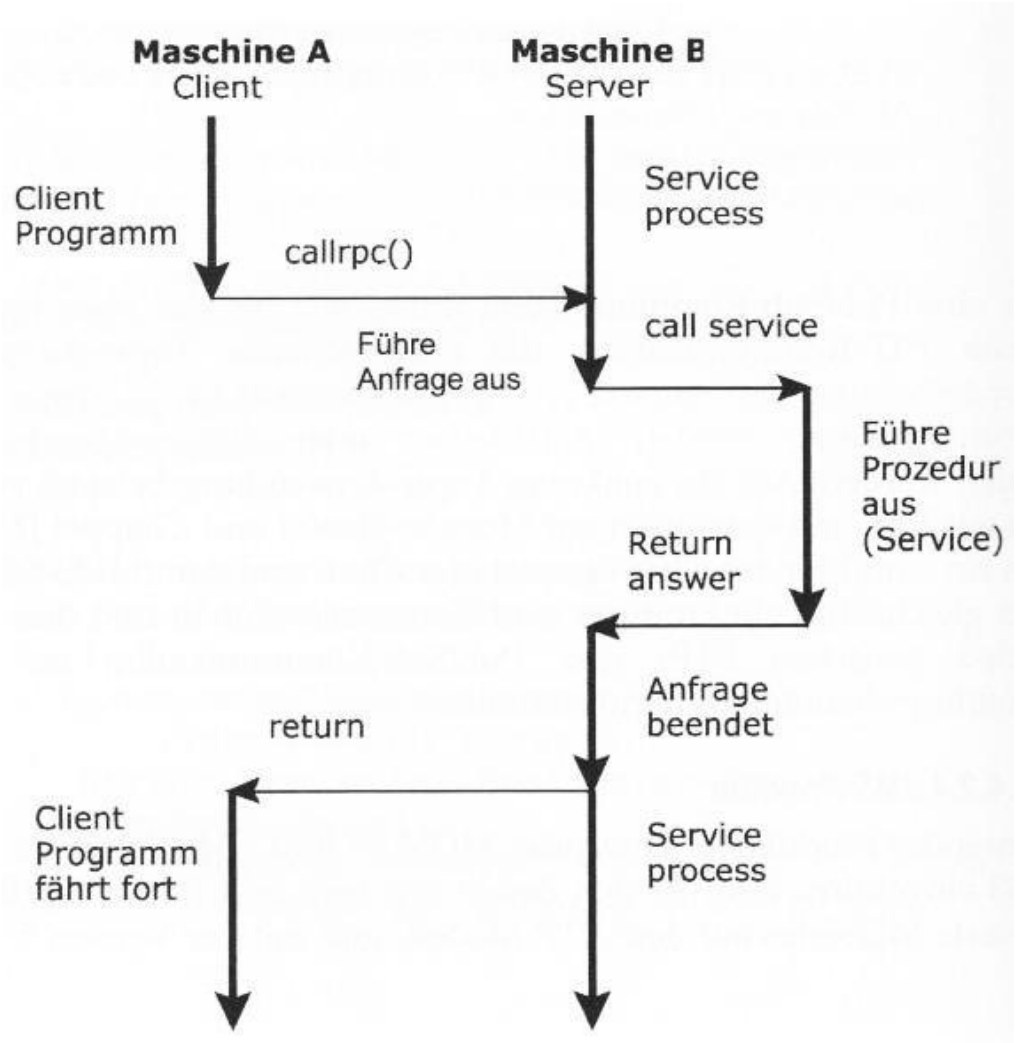
- Nutzung, wenn die Anwendung noch nicht klar in Client-Server bzw. Sender/Empfänger aufgeteilt ist
- eine noch nicht verteilte (monolith.) Anwendung ist quasi eine Sammlung von Funktionen/Prozeduren
- Diese Prozeduren kann man einteilen in
  - Prozedur-Aufrufer (= Clients) und
  - die Prozedur selbst (wird dadurch zum Server)
- dadurch „Aufprägung“ einer C/S-Struktur auf monolith. Struktur
  → Verteilung der Prozeduren auf mehrere Rechner
- Verteilte Prozeduren werden scheinbar zentral abgewickelt,
  → Benutzer sieht das verteilte System gar nicht
- Voraussetzung dafür: ein Programm (Prozedur) muss eine Prozedur auf einem entfernten Rechner aufrufen können!
  → dazu sind entfernte (Prozedur-)Aufrufe (Remote Calls) nötig!

3.3.4.3 Kooperative Modelle mit entfernten Aufrufen (II)

Ablauf entfernter Aufrufe:

Beispiel:
Prozess auf Rechner A (Client) ruft Dienst von Rechner B (Server) auf:

- Prozess A wird blockiert
- entfernter Prozedur-Aufruf (ggf. mit Parametern)
- Ausführung der Dienst-Prozedur auf Rechner B
- Ergebnisrückgabe aus der Prozedur an Prozess A
- Fortsetzung von Prozess A

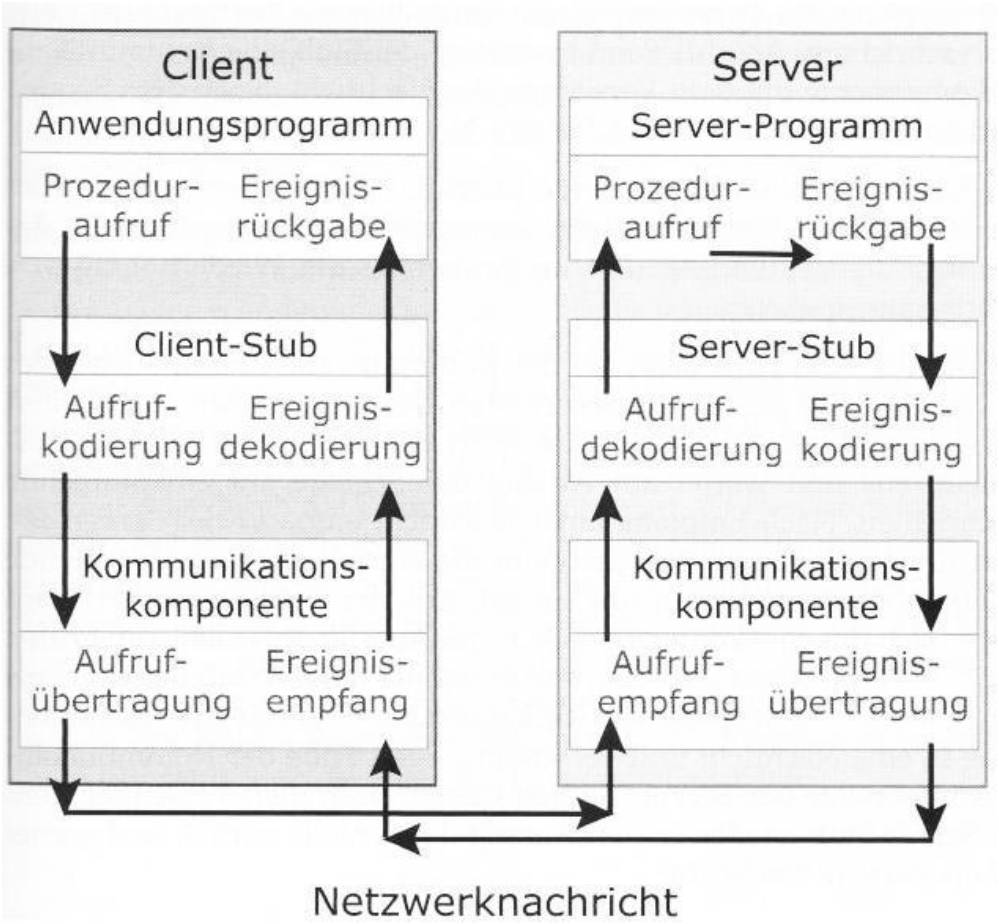


Quelle: Bengel u.a.: Masterkurs parallele und verteilte Systeme. Vieweg/Teubner 2008

3.3.4.3 Kooperative Modelle mit entfernten Aufrufen (III)

Abbildung des entfernten Aufrufes auf Nachrichten (am Bsp. RPC):

Die vom Client aufgerufene Prozedur steht i.allg. nicht lokal zur Verfügung!
Das RPC-System muss also den Prozeduraufruf umwandeln in eine Kommunikation mit dem Server (Codierung und Übermittlung des Aufrufs inkl. Parameter und Ergebniswerten).
Ggf. wird auch die Lokalisierung des Servers, Behandlung von Übertragungsfehlern und Ausfällen usw. übernommen.
Dies alles ist für den Client (die aufrufende Anwendung) transparent!



Quelle: Bengel u.a.: Masterkurs parallele und verteilte Systeme. Vieweg/Teubner 2008

3.3.4.3 Kooperative Modelle mit entfernten Aufrufen (IV)

Stubs = lokale Prozeduren beim Client oder beim Server

- Client-Stub
  - wird bei Transformation des lokalen Prozeduraufrufes benutzt, um aus den Prozedurparametern eine Nachricht an den Server aufzubauen (p*arameter marshalling*) und diese danach an eine Kommunikationskomponente zum Versand zu übergeben.
  - Danach blockiert der Client-Stub und wartet auf Rückantwort.
  - Beim Eintreffen der Antwort decodiert der die Nachricht und übergibt die Ergebnisse als Ergebnis der aufgerufenen Prozedur.
- Server-Stub
  - läuft i.allg. in Endlosschleife und wartet auf Nachrichten
  - Nach Empfang decodiert er die Nachricht, bestimmt die auf dem Server aufzurufende Prozedur und ruft sie auf (Diensterbringung)
  - Nach Ablauf der Prozedur wandelt er die Ergebnisse in eine Nachricht um und versendet diese unter Nutzung einer Kommunikationskomponente als Antwort an den Client

3.3.4.3 Kooperative Modelle mit entfernten Aufrufen (V)

Verschiedene Formen entfernter Aufrufe:

- Remote Procedure Call (RPC)
  - ONC-RPC (Open Network Computing) oder Sun RPC
  - DCE-RPC (Microsoft)
- entfernte Objekt- bzw. Methodenaufrufe
  - mittels CORBA (Common Object Request Broker Architecture)
  - Remote Method Invocation (RMI, bei Java)
- entfernte Komponentenaufrufe
  - Distributed Component Object Model (DCOM, Microsoft) und .NET Remoting
- entfernte Service-Aufrufe
  - mit Web-Services (SOAP)
  - XML-RPC für entfernte Prozeduraufrufe via Internet

3.4 GPU-Programmierung

General Purpose Computation on Graphics Processing Unit (GPGPU)

= Nutzung von GPU(s) für (allgemeine) Berechnungen
(damit „andere“ Nutzung der GPU im Vgl. zu ihrer eigentlichen Aufgabe)

Bsp.: Berechnungen zu techn. oder wirtschaftlichen Simulationen

Parallelität führt hier zu hohen Geschwindigkeitssteigerungen im Vergleich zum „gewöhnlichen“ CPUs

Ursprung der GPGPU: *Shader* (zur Berechnung von Schatten-Effekten, Rendering, auf Pixelbasis in der 2D/3D-Grafik)
Moderne GPUs haben viele (~1000) solche parallel arbeitende Shader-Einheiten

3.4 GPU-Programmierung

- GPUs sind Prozessoren mit einem anwendungsspezifischen Design,
  - daher hier z.T. „exotische“ Datentypen (9 Bit, 12 Bit mit Festkommastelle)
  - Dafür fehlen oft für CPUs typische Registerbreiten von 32, 48, 64, 80 Bit.
  - Gleitkomma-Berechnungen (z.B. nach IEEE 754) sind bei GPUs oft nicht im Befehlssatz vorhanden und müssen relativ aufwändig per Software emuliert werden.
  - Daher eignen sich GPUs vor allem zur Berechnung von Datentypen, die mit vergleichsweise geringen Bit-Breiten arbeiten.
- Es gibt inzwischen erweiterte GPUs, die neben den von der GPU benötigten Datentypen auch universelle Datentypen/Operationen z.B. zur direkten Berechnung nach IEEE 754 beinhalten (z.B. Nvidia Fermi mit 32-Bit-Integer, und einfache/doppelte Genauigkeit bei Gleitkomma-Datenformaten)
- Problem: Anbindung der GPU an die Rechnerarchitektur erfolgt meist über PCIe, mit höheren Latenzzeiten und geringeren I/O-Durchsatzraten.

→ Nutzen der GPU-Berechnung im Vgl. zu CPU-Berechnungen vorher abschätzen!

3.4 GPU-Programmierung

Programmiermodelle/-sprachen für GPGPU:

Voraussetzung für ausreichenden „Effekt“:
→ massive Parallelität in den Anwendungen

- OpenCL: einheitliche Schnittstelle zur Umsetzung von GPGPU-Berechnungen, offener Standard, für viele HW-Plattformen,
- CUDA: proprietäres Framework für Nvidia-GPUs
- C++ AMP: von Microsoft initiierte Spracherweiterung von C++, relativ HW-unabhängig (ggf. inkl. „Ausweichen“ auf CPU); AMP soll eine deutliche Trennung von Algorithmus und HW unterstützen

Um Programme auf einer GPU auszuführen, ist ein Hostprogramm zur Steuerung des Informationsflusses nötig. Meist wird zur Laufzeit der in einer C-ähnlichen Sprache formulierte GPGPU-Code auf Anweisung des Hostprogrammes compiliert und an den Grafikprozessor zur Weiterverarbeitung gesandt, der dann die errechneten Daten an das Hostprogramm zurückgibt.

3.4 GPU-Programmierung

OpenCL (***Open C****omputing* ***L****anguage*)

- Schnittstelle für Rechner mit CPU(s) und GPU(s),
- urspr. von Apple entwickelt
- für versch. Plattformen implementiert
- ein OpenCL-System besteht aus
  - 1 Host, er verteilt die sog. Kernel zur Laufzeit auf die verfügbaren Geräte. Kernel sind in einer geeigneten Sprache*) geschrieben, werden zur Laufzeit vom OpenCL-Compiler übersetzt und danach von einem Gerät ausgeführt
  - mind. 1 OpenCL-Gerät. Ein Gerät besteht aus einer oder mehreren unabhängigen Recheneinheiten („compute unit“, CU, z.B. die Kerne eines Mehrkernprozessor, oder die Shader-Einheiten einer GPU). Eine CU besteht damit aus ein oder mehreren ausführenden Prozessing-Elementen (PEs).

•*) zugehörige Programmiersprache „OpenCL C mit Erweiterungen und Einschränkungen bzgl. Standard-C (auf Basis von C lt. ISO C99)

3.4 GPU-Programmierung

OpenCL (***Open C****omputing* ***L****anguage*)

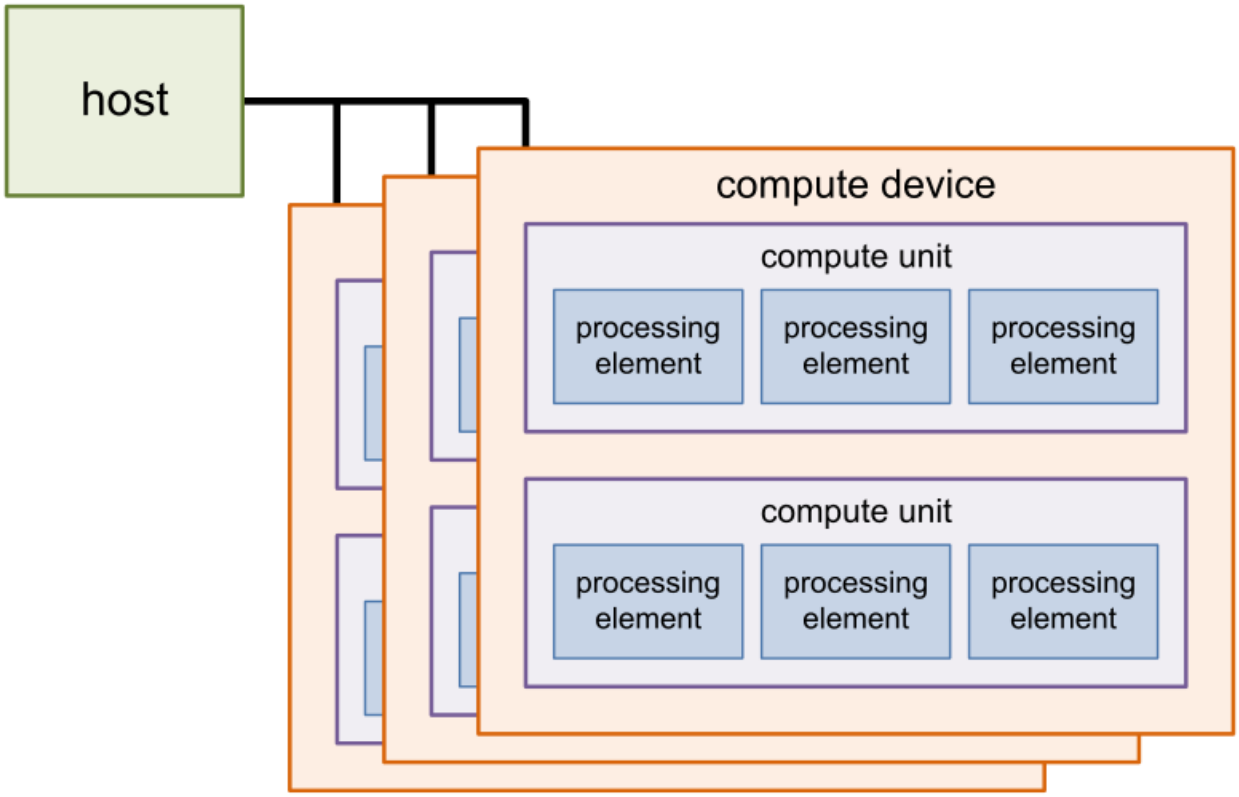


Bild-Quelle: de.wikipedia.org

3.4 GPU-Programmierung

CUDA (früher Compute Unified Device Architecture genannt)

- proprietäre Entwicklung von Nvidia
- zur Entwicklung von Programmteilen, die durch GPU(s) abgearbeitet werden.
- GPU wird mittels CUDA-API quasi als Co-Prozessor eingesetzt
- wird vor allem bei wiss.-techn. Berechnungen verwendet
- Programmierung mittels *C/C++ for CUDA* (C/C++ mit Nvidia-Erweiterungen)
- es existieren auch Wrapper für Perl, Python, Java, Fortran und .NET
- Unterstützung durch Tools (Compiler, Libs, Debugger, Optimizer...)
- Nachteile:
  - proprietäre Lösung
  - aktuelle GPUs sind oft via PCIe an die Hardware angebunden, daher höhere Latenzzeiten und geringere E/A-Durchsatzraten als bei direkter Anbindung „gewöhnlicher“ CPUs

3.4 GPU-Programmierung

OpenAcc (Open Accelerators)

- gemeinsamer „Standard“ von Cray, CAPS, Nvidia, PGI
- zur Entwicklung von SW für heterogene Systeme aus CPUs + GPUs
- vgl.-bar mit OpenMP (Kap. 3.2.5)
- Compilerdirektiven und zusätzl. Fkt. zur Steuerung der Parallelität
- Unterstützung durch versch. Compiler von Herstellern und OpenSource (z.B. für C, C++, Fortran)

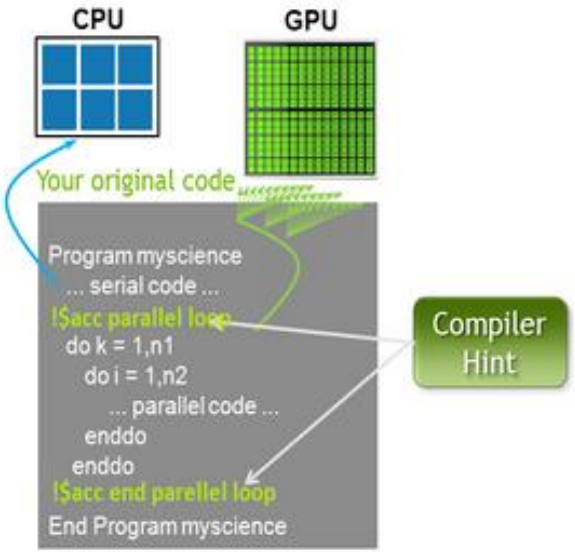
- Bsp. für Compiler-Direktiven in OpenAcc:

```
#pragma acc parallel
#pragma acc kernels
```

Definition der parallelen Berechnung von kernels auf einem Accelerator

```
#pragma acc data
```

Definition und Kopieren von Daten zu/vom Accelerator.



https://developer.nvidia.com/openacc

3.5 Hybride Programmiermodelle

- aktuelle Entwicklungen zur parallelen Programmierung für Cluster aus Multicores
  - sowohl parallel auf Knoten-Level (nodes), z.B. mit MPI
  - als auch innerhalb eines Knotens, z.B. mit OpenMP oder MPI-3 shared memory Feature
  - „MPI+x“