

Architektur und Programmierung von Grafik- und Koprozessoren

Performanz von Computerprogrammen

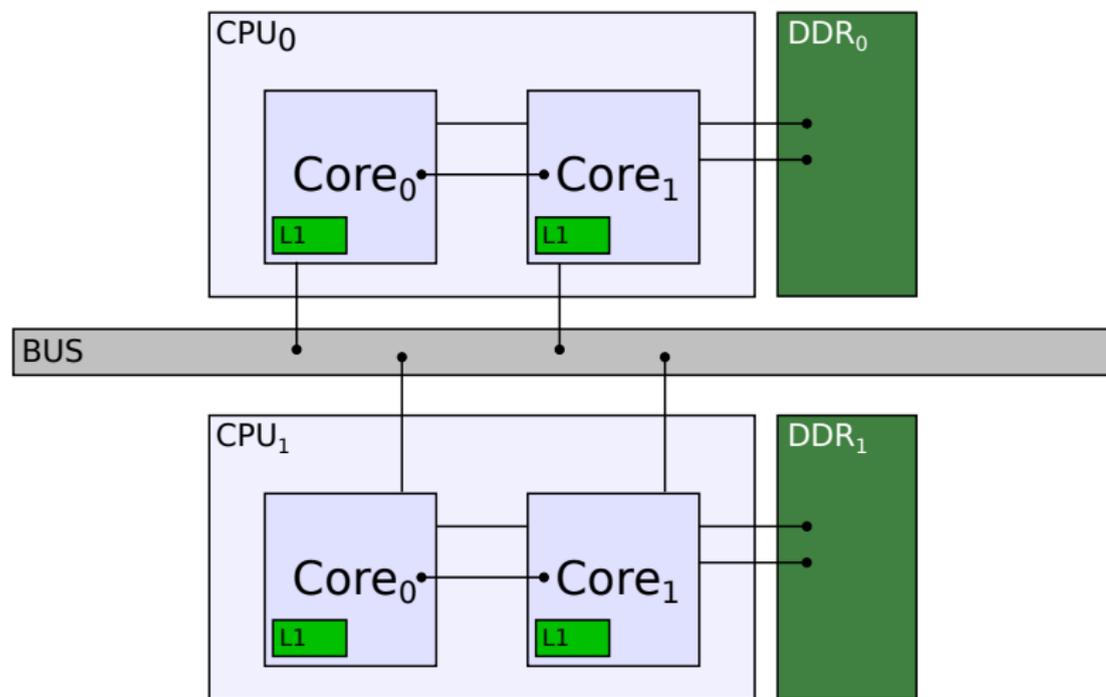
Stefan Zellmann

Lehrstuhl für Informatik, Universität zu Köln

SS2019

NUMA und Topologien

Einfache BUS Topologie



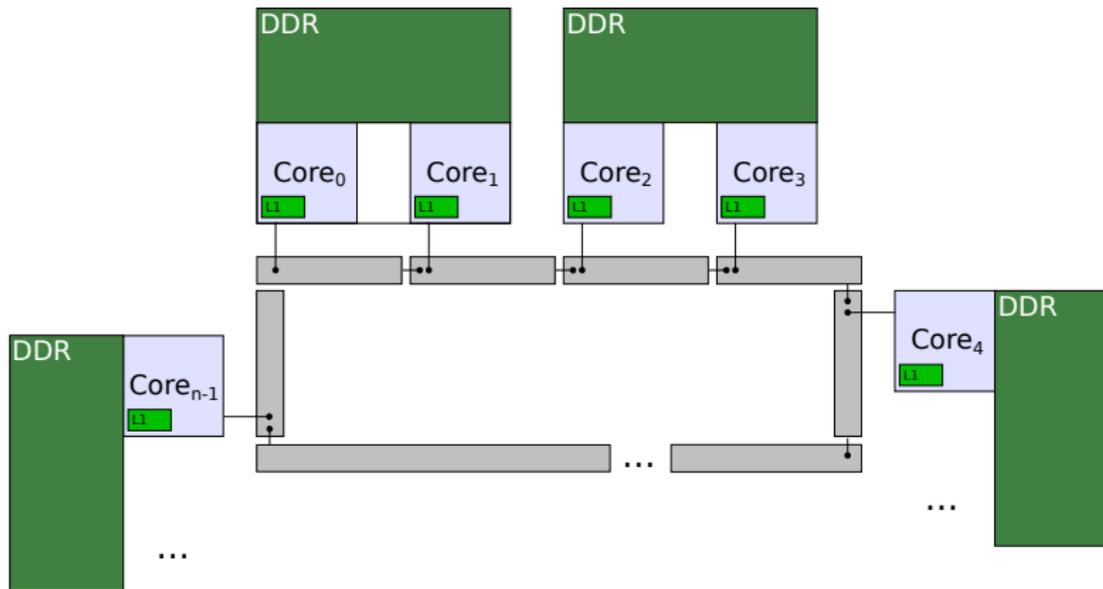
NUMA und Topologien

Einfache BUS Topologie

- ▶ Kommt bei vielen Multi-Core Architekturen vor (vereinfachte Darstellung, es gibt nur L1 Caches).
- ▶ Speicherzugriffe in den eigenen RAM müssen zwischen $Core_0$ und $Core_1$ synchronisiert werden. Ggf. verändert $Core_0$ ein Datum, das $Core_1$ im lokalen L1 Cache auch bereits geändert hat.
- ▶ Synchronisation zwischen Cores von CPU_0 und CPU_1 erfordert Kommunikation über den BUS \Rightarrow höhere Latenz. Dadurch ergeben sich i. Allg. nichtuniforme Speicherzugriffe.

NUMA und Topologien

Kompliziertere Topologien (z. B. Ring-BUS)



Legacy Chipsets

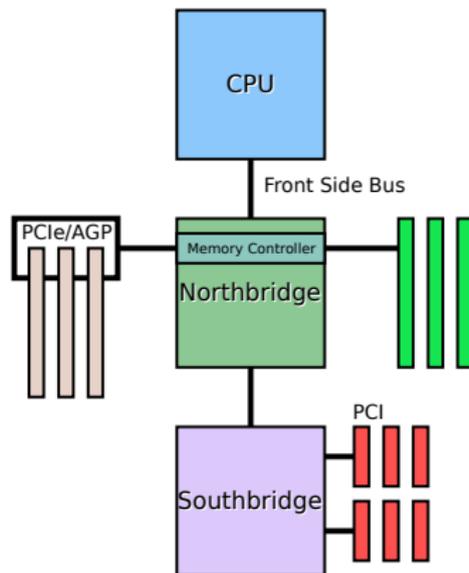


Abbildung: Memory Controller war Teil der *Northbridge* (separater Chip auf Mainboard). Später: Northbridge (und MC!) Teil der CPU.

Moderne Chipsets

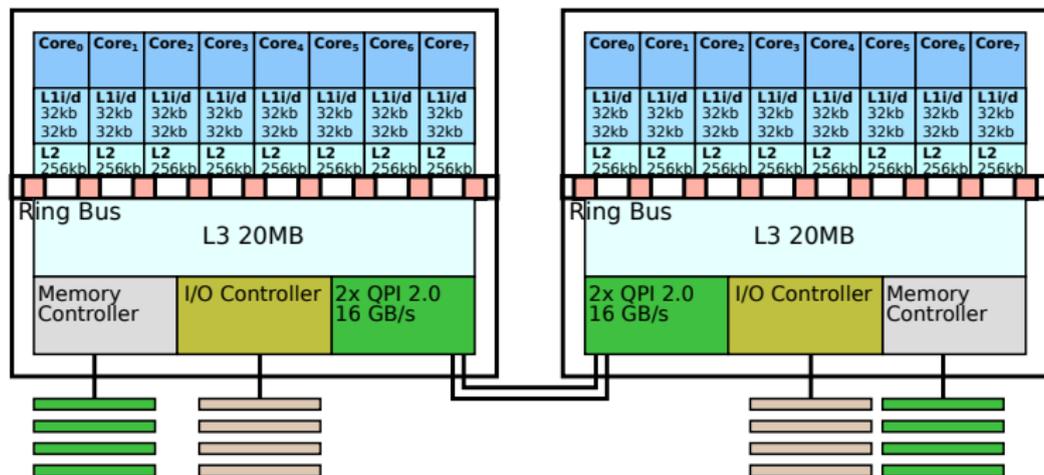


Abbildung: 2 × Intel “Sandy Bridge” Xeon E5-2670 Konfiguration, zwei QPI Links zwischen Prozessoren (8 GT/s bei 2 byte pro Transfer, bidirektional, jeder Link in jede Richtung 16 GB/s ⇒ total 32 GB/s.). Vgl.: Performance Evaluation of the Intel Sandy Bridge Based NASA Pleiades Using Scientific and Engineering Applications, Saini et al. 2013.

Cache-kohärentes NUMA

- ▶ Problem bei Architekturen mit Caches: was passiert, wenn beide Prozessoren das gleiche geteilte Datum im Cache speichern, und ein Prozessor ändert das Datum lokal?
- ▶ Die meisten Infrastrukturen (Hardware + OS) implementieren *cache-kohärentes NUMA* (ccNUMA).
- ▶ Cache-Kohärenz Protokolle lassen es für Entwickler so aussehen, als gäbe es keine lokalen Caches, sondern als wären alle Caches geteilt.

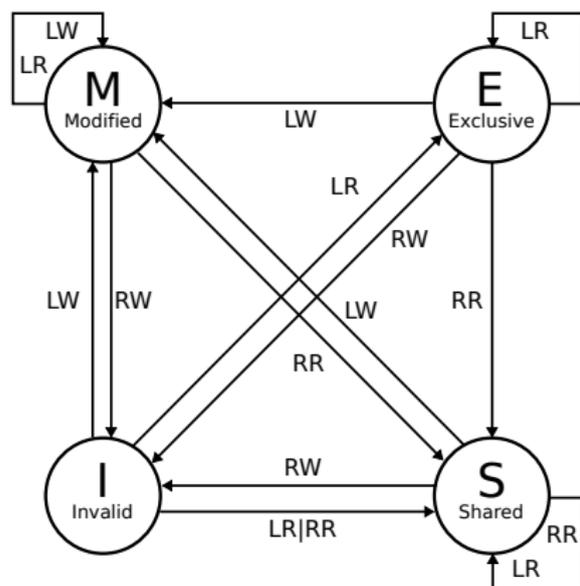
Cache-Kohärenz Protokolle

- ▶ Werden bspw. vom *Cache Controller* (CPU Funktionseinheit) umgesetzt – z. B. (“Snooping Protokolle”):
 - ▶ *write-invalidate*: der schreibende Prozessor sendet ein Token über den Interconnect, das den gerade beschriebenen Cache Block identifiziert und invalidiert. Dann kann die lokale Kopie angepasst werden.
 - ▶ *write-update*: der schreibende Prozessor verteilt via *Broadcast* das neue Datum, und alle Prozessoren aktualisieren die Kopie im lokalen Cache.

MESI Protokoll

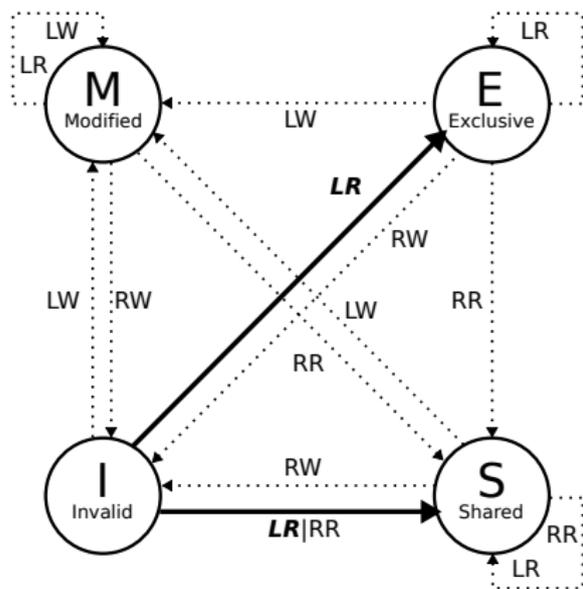
- ▶ Jeder Prozessor verwaltet seinen lokalen Cache.
- ▶ Protokoll modelliert *Zustandsautomat*. Cache Line in einem der Zustände
 - ▶ **Modified** - kein anderer Cache speichert die Cache Line, sie wurde außerdem bereits modifiziert.
 - ▶ **Exclusive** - kein anderer Cache speichert die Cache Line, sie kann modifiziert werden, wurde aber noch nicht modifiziert.
 - ▶ **Shared** - andere lokale Caches haben eine Kopie der Cache Line mit identischem Inhalt.
 - ▶ **Invalid** - Cache Line wird derzeit nicht benutzt.
- ▶ Die Prozessoren tauschen untereinander Nachrichten aus, um ihre lokalen Caches abzugleichen.
- ▶ Wichtig: modifizierte Cache Lines sind auch exklusive Cache Lines.

MESI Protokoll



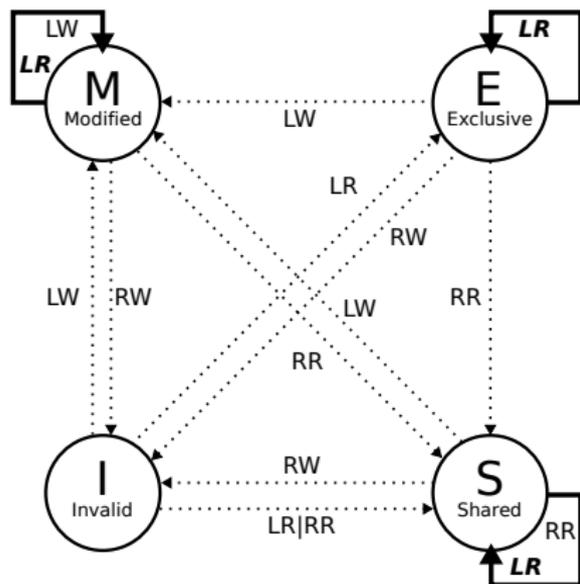
- Mögliche Zustandsübergänge aufgrund von *local read* (LR), *local write* (LW), *snooped remote read* (RR) oder *snooped remote write* (RW).

MESI Protokoll



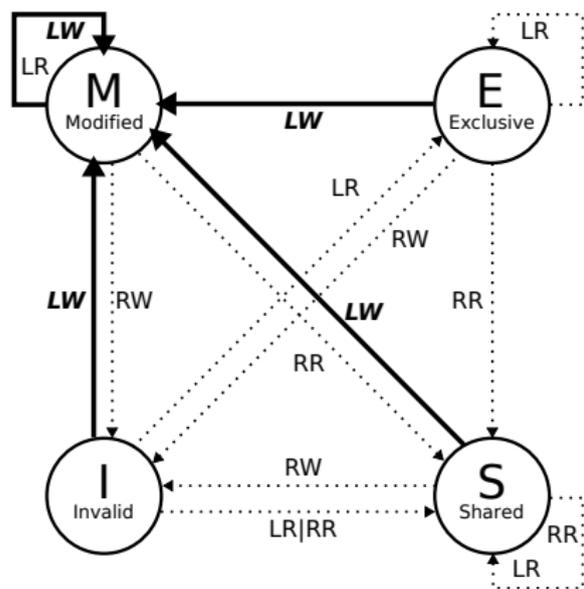
- ▶ **LR** Zustandsübergang von Invalid (Prozessor liest Cache Line): prüfe (über Bus), ob Cache Line geteilt. Falls ja: *Shared*, sonst: *Exclusive*.

MESI Protokoll



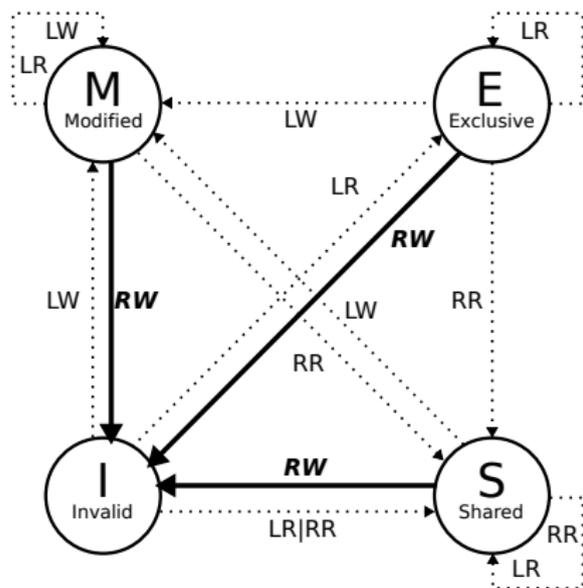
- ▶ **LR** Alle anderen Zustände:
Keine Veränderung.

MESI Protokoll



- ▶ **LW** Alle lokalen Schreiboperationen überführen Cache Line in Modified State.
- ▶ Falls vorher Invalid oder Shared, sende Broadcast Nachricht an alle anderen Prozessoren, diese invalidieren ihre lokale Kopie.

MESI Protokoll



- ▶ **RW** Schreiboperationen über Bus (also von einem anderen Prozessor!) führen dazu, dass Cache Line invalidiert und in den Speicher zurückgeschrieben wird.

MESI Protokoll

- ▶ Es kann nur in Cache Lines geschrieben werden, die im Modified oder Exclusive Zustand sind.
- ▶ Remote Writes werden mit sog. *read for ownership* implementiert: Core möchte in Cache Line schreiben, die im Shared Zustand ist. Dazu schickt er zuerst per Broadcast eine Nachricht über den Bus, dass alle anderen Cores ihre Kopie der Cache Line invalidieren. Dann *liest* der Core die Cache Line und versetzt sie lokal in den *Exclusive* Zustand. Danach kann geschrieben werden (modify).
- ▶ Hält ein Core eine Cache Line im *Modified* Zustand, muss er den Bus überwachen (*snooping*, ob andere Cores auf die Cache Line zugreifen wollen. *Dann erst* muss der Core das geänderte Datum erst in den Hauptspeicher zurückschreiben. Die entfernte Leseoperation kann bspw. direkt aus dem Cache bedient, oder zurückgehalten werden.

MESI Protokoll

- ▶ Prinzip: opportunistisch – Kommunikation über Bus nur falls nötig, modifiziere Speicher nur falls nötig.
- ▶ Lesen aus lokalem Cache ist billig.
- ▶ Lokale Schreiboperationen führen nur zu Datenkommunikation, falls Cache Line geteilt.
- ▶ Schreiben über Bus ineffizient.
- ▶ Andere Protokolle: MSI, MESIF, MOSI, etc.

Programmieren für NUMA Systeme

- ▶ Spezialisierte APIs, z. B. `libnuma` unter Linux, Funktionen wie `numa_alloc_local()`, `numa_bind()`.
- ▶ Tools wie `numactl`: “pinne” Prozess an bestimmten Core, etc.
- ▶ APIs und Tools exponieren Funktionalität des *Betriebssystems*.

NUMA: Atomare Instruktionen

```
atomic<int> counter(0); // shared variable
void parallel_func() {

    counter++;
}
```

- ▶ Atomare Operationen gehen durch die Cache Hierarchie.
- ▶ Was passiert, wenn Cores geteilte Variable atomar modifizieren?

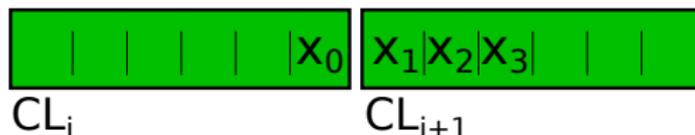
NUMA: Atomare Instruktionen

- ▶ Einfache Implementierung (vergleichbar mit Critical Sections!):
 1. Sende LOCK auf Bus (broadcast).
 2. Cores empfangen LOCK und schließen alle Speicheroperationen ab. Core, der LOCK initiiert hat, wartet darauf.
 3. When ACK ("acknowledge") von allen Cores empfangen, führe atomare Operation aus. Sende dann broadcast Nachricht an alle Cores, dass sie weiterarbeiten können.
 4. Dies ist ausgesprochen langsam.
- ▶ Besser: MESI Protokoll - read for ownership \Rightarrow Cache Line, die Datum beinhaltet ist in Exclusive Zustand und *kann* atomar modifiziert werden.

NUMA: Atomare Instruktionen

► Problem: Alignment

1. MESI Protokoll implementiert diese Art von “Locking” auf Cache Line Level. Funktioniert nicht, wenn das geteilte, zu modifizierende Datum zwei Cache Lines “überlappt”.
2. x86 bspw. implementiert den langsameren LOCK basierten Modus für *unaligned atomic* Instruktionen.



Default Alignment von C Datentypen mit GCC

- ▶ char: 1-byte.
- ▶ short: 2-byte.
- ▶ int: 4-byte.
- ▶ Zum Vergleich: Cache Line Alignment (x86): 64-byte.
- ▶ Zusammengesetzte Typen (struct) “erben” das Alignment vom ersten Member. Typen wie `std::atomic<>` unterstützen atomare Operationen auf zusammengesetzten Typen.
- ▶ Daher: stelle sicher, dass atomar genutzte Typen aligniert (z. B. `__attribute__((align(64)))`).

Packing Zusammengesetzter C-Typen

```
struct P1 {
    char c;
    int i[3];
    short s;
};

struct P2 {
    int i[3];
    short s;
    char c;
};

int main() {
    struct P1 p1;
    struct P2 p2;
    printf("%d,", (int)sizeof(p1));
    printf("%d\n", (int)sizeof(p2));
}
```

Output (Apple Clang ANSI-C Compiler, 7.0.2): 20,16 (!)

Alignment und Packing

Bemerkungen

- ▶ Auf Architekturen mit Caches sollten *inner loop* Typen cache-effizientes Alignment und Packing haben.
- ▶ Da atomare Operationen mit *Cache Line Granularität* durchgeführt werden, ist auf NUMA Systemen Alignment und Packing noch viel wichtiger!
- ▶ Mehrkernprozessoren wie Core i7 etc. sind heutzutage immer NUMA Systeme und implementieren MESI oder Varianten davon (\Rightarrow Relevanz).

NUMA “Takeaways”

- ▶ Multi-Core und Many-Core Architekturen \Rightarrow kompliziertere Speicherarchitekturen.
- ▶ NUMA hilft, Skalierungsprobleme zu vermeiden, da weniger Kommunikation auf dem Bus.
- ▶ Dadurch ist es schwieriger, Latenz von Speicherzugriffen einzuschätzen, viele entfernte Speicherzugriffe werden zu Bottlenecks.
- ▶ Caches nicht direkt adressierbar \Rightarrow zusätzliche Chipfläche und Software Cycles für Cache-Kohärenz Protokolle.
- ▶ NUMA Komplexität wird zwar von den meisten Architekturen durch Cache-Kohärenz versteckt, Entwickler von Multithreading Applikationen sollten jedoch über nichtuniforme Speicherzugriffe wissen.

Verteilte Speicherarchitekturen

Abgrenzung

- ▶ **Symmetric Multi-Processing** - ein Speicher, viele Prozessoren, die mit gleicher Latenz auf den Speicher zugreifen.
- ▶ **Non-Uniform Memory Access** - Kerne greifen mit niedriger Latenz auf lokalen Speicher zu.
- ▶ **Distributed Memory** - Prozessoren verfügen über lokalen Speicher und sind über Interconnect (Netzwerk) miteinander verbunden. Zugriff auf entfernten Speicher ist nicht direkt möglich. Vielmehr muss der dem entfernten Speicher zugeordnete Prozessor *benachrichtigt* werden, dass von entfernt auf Speicher zugegriffen werden soll.

Verteilte Speicherarchitekturen

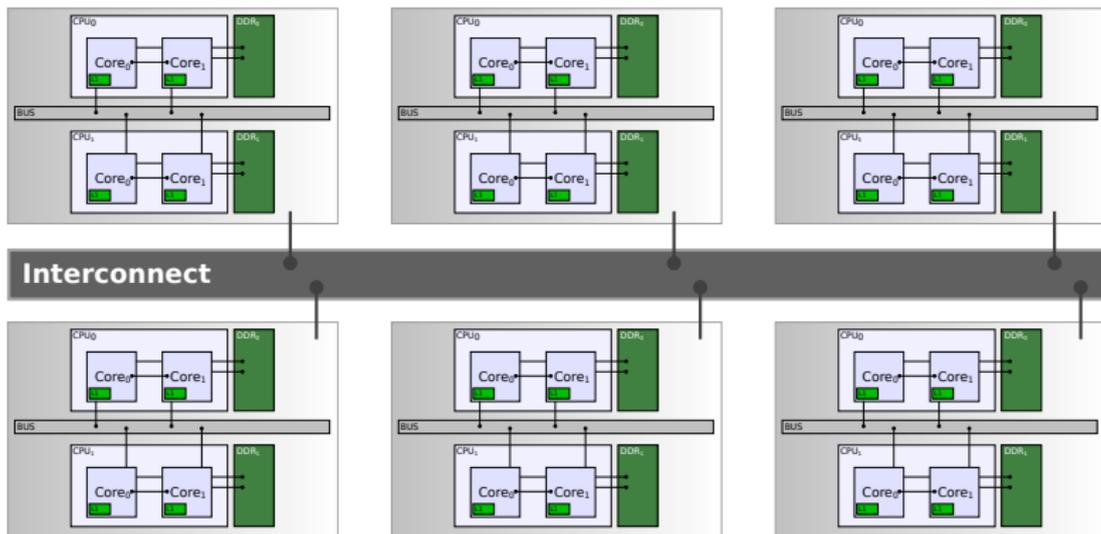
Abgrenzung

- ▶ Abgrenzung zwischen Architekturen teils schwierig. NUMA Systeme haben bspw. Charakteristiken von Distributed Memory Systemen, jedoch werden diese durch die Implementierung (z. B. Cache-Kohärenz) versteckt.
- ▶ Distributed Memory Systeme zeichnen sich üblicherweise durch räumliche Trennung der *Rechenknoten* aus.

Verteilte Speicherarchitekturen

Hybride Systeme

Distributed Memory Systeme sind in Wirklichkeit oft hybride Systeme: Rechenknoten sind per Interconnect verbunden und sind selbst SMP oder NUMA Systeme.



Verteilte Speicherarchitekturen

Programmiermodelle

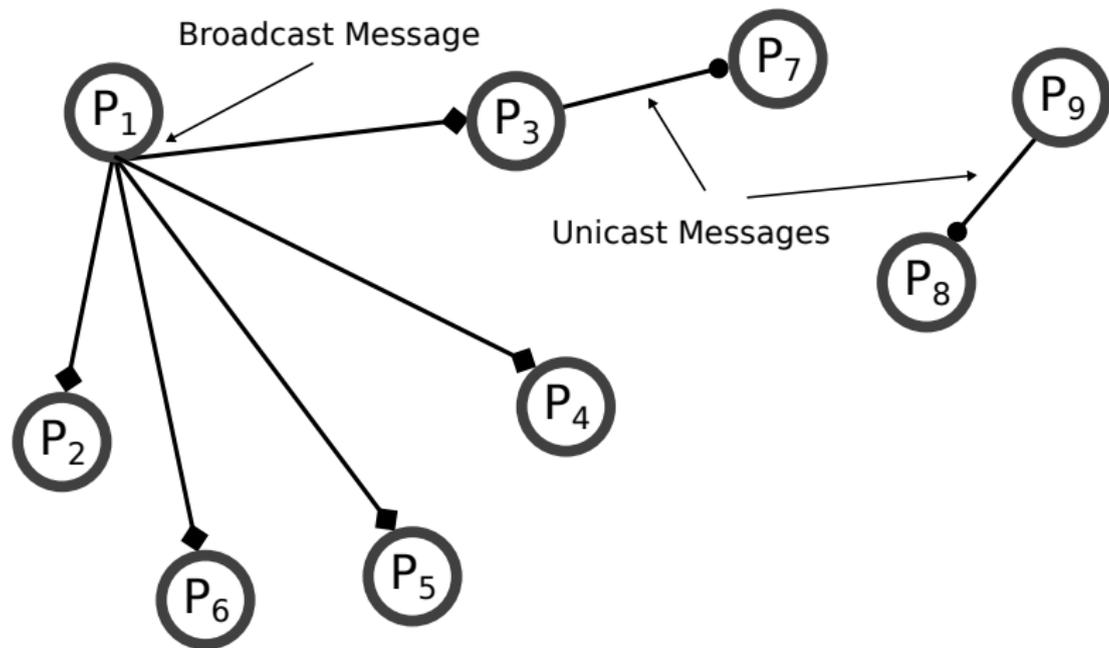
- ▶ Es gibt verschiedene Programmiermodelle für Distributed Memory Systeme, z. B. aus dem Umfeld Netzwerkprogrammierung mit Protokollen wie TCP/IP.
 - ▶ Programmierung mit Sockets.
 - ▶ Programmierung über Web Protokolle wie HTTP.
- ▶ Im High-Performance Computing (HPC) Umfeld spielt jedoch der *Message Passing Interface* (MPI) Standard eine wichtige Rolle.

Verteilte Speicherarchitekturen

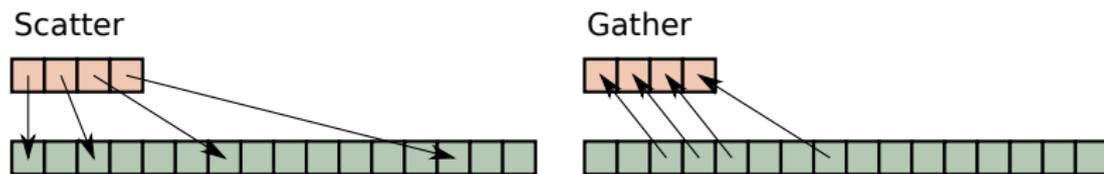
Message Passing Interface

- ▶ Sockets erlauben Zugriff auf Netzwerkschnittstelle auf sehr niedrigem Level und per direkter Kommunikation mit dem Betriebssystem.
- ▶ Im HPC Umfeld kommen jedoch häufig schnelle Interconnects (z. B. InfiniBand) zum Einsatz, die nicht IP basiert sind.
- ▶ Solche Netzwerke kann man teils plattformspezifisch programmieren, MPI ist jedoch ein standardisierter Ansatz um HPC Interconnect Architekturen plattformunabhängig zu nutzen.
 - ▶ InfiniBand bspw. kann man auch mit Sockets programmieren (sog. *IP over IB*), das vermeidet man aber, da IP Stack und OSI Schichtenmodell stark latenzbehaftet sind.

Message Passing



Message Passing



Scatter verteilt Daten, *Gather* sammelt Daten. Bemerkungen bzgl. Kohärenz und Lokalität gelten auch hier.

Message Passing

Message Typen

- ▶ Unicast: `MPI_send()`, `MPI_recv()`
- ▶ Broadcast: `MPI_Bcast()`
- ▶ Scatter/Gather: `MPI_Scatter()`, `MPI_Gather()`
- ▶ Reduktion: `MPI_Reduce()`
- ▶ ...

Kollektive Aufrufe wie Broadcasting müssen synchronisiert werden (`MPI_Barrier()`).

Struktur eines MPI Programms

```
#include <mpi.h>
#define MASTER 0
#define SLAVE 1
int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == MASTER) {
        char const* msg = "MSG";
        MPI_Send(msg, 4, MPI_BYTE, SLAVE, 1,
                 MPI_COMM_WORLD);
    }
    else {
        char str[32];
        MPI_Status s;
        MPI_Recv(str, 32, MPI_BYTE, MASTER, 1,
                 MPI_COMM_WORLD, &s);
    }
    MPI_Finalize();
}
```

Struktur eines MPI Programms

Übersetzung mit speziellem MPI Compiler, z. B.

```
mpicc hello_mpi.c -o hello_mpi
```

Führe MPI Programm auf dafür konfiguriertem Cluster aus, z. B.

```
mpirun -n 256 hello_mpi oder
```

```
mpiexec -n 256 hello_mpi
```

MPI Implementierungen

- ▶ Verschiedene offene und nicht-offene Implementierungen.
 - ▶ insb. MPICH und OpenMPI
- ▶ Ursprünglich Fortran und C.
- ▶ Adaptionen für verschiedene Sprachen, z. B. C++ Interface.

MPI Saxpy

```
#include <mpi.h>
int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int comm_size, rank;
    MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int N = atoi(argv[1]);
    float a, y[N], x[N];

    if (rank == 0)
        init(a, y, x, N);

    int N_local = N / comm_size;

    float al, yl[N_local], xl[N_local];
    // cont. on next slide
```

MPI Saxpy

```
// cont. from prev slide
MPI_Scatter(a, 1, MPI_FLOAT,
           a1, 1, MPI_FLOAT,
           0, MPI_COMM_WORLD);
MPI_Scatter(y, N_local, MPI_FLOAT,
           y1, N_local, MPI_FLOAT,
           0, MPI_COMM_WORLD);
MPI_Scatter(x, N_local, MPI_FLOAT,
           x1, N_local, MPI_FLOAT,
           0, MPI_COMM_WORLD);

saxpy(y1, a1, x1, N_local);

MPI_Gather(y1, N_local, MPI_FLOAT,
          y, N_local, MPI_FLOAT,
          0, MPI_COMM_WORLD);

MPI_Finalize();
}
```

Message Passing Bemerkungen

- ▶ NUMA Systeme (auch wenn sie in *einer* CPU verbaut sind) sind verteilte Systeme. Sie versenden untereinander Nachrichten.
- ▶ Cache-Kohärenz Protokolle (z. B. MESI) verstecken Message Passing. Cache-Kohärenz ist fehlertolerant, bringt jedoch Overhead mit. Alternative: nicht-cache-kohärente NUMA Systeme mit Message Passing.
- ▶ Mehrere MPI Prozesse können auf einer CPU ausgeführt werden!
 - ▶ Fabrique (z. B. *QuickPath Interconnect* (QPI)) ist dann das Netzwerk.