Wir befinden uns immer noch im Abschnitt Parallelrechner. Das Kapitel ist jetzt überschrieben mit Klassifikation und Architektur. Um dieses Kapitel zu verstehen, ist es natürlich ganz gut, wenn Sie die GRA gehört haben. Ahnung von Speicherhierarchie haben, haben wir ja in dieser Vorlesung drüber gesprochen und was Instruction Level Parallelism bedeutet, abgekürzt mit ILP. Worum geht es? Wir wollen in dem Kapitel versuchen zu verstehen die unterschiedlichen Konzepte und auch die Klassifikationsmöglichkeiten von und für Parallelrechner. Dementsprechend geht es in diesem Kapitel um die Architektur von Hochleistungs- bzw. Parallelrechnern. grundsätzlich uns die Blockschaltbilder anschauen, was ist ein Shared Memory SIMT Rechner, was ist ein Distributed Memory SIMT Rechner, was ist das gleiche als MIMT Rechner, also Shared und Distributed Memory, was bedeutet eigentlich Cluster Computing, was ist ein CC NUMA Rechner, was ist Grid und Cloud Computing und wie ist so eine GPU als Parallelrechner aufgebaut oder als Parallelrechner zu sehen. Also die einfachste Klassifikation für Parallelrechner, die wir haben, die ich auch schon gebraucht habe, die Sie vielleicht auch schon woanders gehört haben, das ist SIMD-Rechner und MIMD-Rechner. Wofür stehen diese Kürzel? Diese Kürzel stehen für Single Instruction Multiple Data und Multiple Instruction und Multiple Data. Was bedeutet das jetzt? parallelität zu nutzen das heißt ich habe eine instruktion und diese instruktion soll für sehr viele also die gleiche instruktion soll für sehr viele daten verwendet werden also das beste beispiel dafür ist vektorverarbeitung das heißt ich habe zum beispiel ich bilde ein skalarprodukt und da habe ich also zwei vektoren die ich schnell miteinander im abwechseln multiplizieren und addieren muss um dieses skalar zu bilden und es ist also Ich kann das auch weitestgehend parallelisieren, weil diese ganzen Multiplikationen sind voneinander unabhängig. Dann muss ich die Ergebnisse alle aufaddieren. Auch das kann ich relativ zügig machen, parallel. Also im Wesentlichen ist SIMD so etwas ähnliches wie Vektorverarbeitung. Ganz anders dazu ist MIMD-Verarbeitung. Ich habe hier Multiple Instructions auf Multiple Data. Das bedeutet im Wesentlichen, dass jedes Processing-Element, also jeder Rechner sein eigenes Programm fährt, unabhängig von den anderen und dass ich halt über Synchronisation und Kommunikation dafür sorge, dass das Ganze ordentlich läuft, dass ich alle Datenabhängigkeiten berücksichtige. Zwischenergebnisse und bei den Zwischenergebnissen wird halt synchronisiert und kommuniziert. Das ist also eine ganz andere Parallelität als diese Vektorverarbeitung. Da gibt es natürlich in dieser Klassifikation auch SISD, also Single Instruction Single Data. Das ist also ein rein sequenzieller Rechner, das heißt ein ganz einfacher von Neumann Rechner, wie wir ihn kennengelernt haben, wo das Programm also von oben nach unten abläuft. kein Instruction Level Parallelism, nichts, also alles rein sequenziell. Und wenn ich dann alle Kombinationen ausprobe, dann gibt es natürlich auch noch MIST, MIST ist MIST, also Multiple Instruction Single Data, dafür gibt es eigentlich kein vernünftiges Beispiel, wo sowas Sinn macht. Also viele Instruktionen auf einem Datum, das kann ich nicht parallelisieren normalerweise, weil das ist ja schon im Begriff, dass ich dann die Daten habe, habe und die muss ich dann immer sequenziell ausführen. Also Mist, ich kenne keinen Mistrechner und macht auch keinen Sinn. Okay, jetzt kann ich mir überlegen,

wie verbinde ich denn meine Rechner? Wir haben also einmal das Prinzip des Shared Memory, das heißt, ich habe hier Netzwerk und über dieses Netzwerk kann ich auf den Speicher zugreifen. Und bei Shared Memory bedeutet das, dass jede CPU auf jedes Speichermodul zugreifen kann. Das ist also auch die CPU 1 kann hier auf das Speichermodul da hinten zugreifen, also das ist vollständig miteinander vernetzt und jede CPU kann auf jede Stelle des Speichers zugreifen. Und wenn ich jetzt in so einem System kommunizieren will, dann passiert das über gemeinsame Speicherbereiche. in eines dieser Module eine globale Variable. Diese globale Variable kann ich zum Beispiel als Semaphore-Variable benutzen oder als gemeinsames Ergebnis benutzen. Also die Kommunikation und auch Synchronisation passiert über gemeinsame Speicherbereiche. Im Gegensatz dazu habe ich beim Distributed Memory jede CPU direkten Zugriff auf ihren eigenen Speicher. Und wenn die miteinander kommunizieren wollen, Mündet es natürlich nicht über Speicherbereiche, weil jede CPU ihren privaten Speicher hat, sondern die Kommunikation läuft über ein Netzwerk. Man spricht in diesem Fall eben von Message Passing, weil die CPU sich Nachrichten über dieses Netzwerk zuschicken müssen, um miteinander zu kommunizieren. Warum gibt es jetzt diese unterschiedlichen Systeme? Memory ist besser oder Shared Memory ist besser? Naja, eigentlich ist intuitiv programmierbar besser Shared Memory. Geht auch ein bisschen schneller, wenn ich über gemeinsame Variable kommuniziere, als über explizite Nachrichten, weil Nachrichten muss ich einpacken, adressieren, alles ein bisschen komplizierter. Allerdings wissen wir, dass das Netzwerk oder der Bus oder was immer auch dazwischen steht, auf globale speicher zugreifen das heißt so ein system hier unten wird zwar schick zu haben ist aber einfach nicht skalierbar werden so ein system wie hier oben über ein schnelles netzwerk ein bisschen deutlicher skalierbar ist weil die speicherlast eben immer auf die private memorie auf die privaten memorie module geht und da kann ich components of the shelf nehmen da kann ich also einfach existierende motherboards nehmen die funktionieren da schon ganz gut ich muss nur dazu und eben so ein Interprozess-Kommunikationsnetzwerk, damit die miteinander kommunizieren können. Also es gibt eben wieder mal nicht die beste Lösung, sondern je nach Anwendung, je nach Anzahl der Prozessoren ist das eine oder das andere besser. Ja, wie sieht typischerweise so ein SIMD-Rechner aus? Der hat natürlich auch ein Shared Memory. Auf diesen Shared Memory können alle Verarbeitungseinheiten schnell zugreifen. Abwehr, spezielle Einheiten, also Vector Processing Units, die Vektorverarbeitung besonders schnell können. Alles läuft in einem globalen gemeinsamen Speicher. Daneben habe ich auch noch Skalareinheiten für Floating Point. Und ich habe auch noch ganz normal meine Integer Processing Units, also die ALUs, wie wir sie kennen. Und an diesem Bus hier, Systembus, da ist natürlich auch noch das IOS-System angeschlossen, über das ich dann auf die Peripherie zugreifen kann. schaltbild für so ein einzelvektorprozessor was mich ja nicht hindert dass ich so ein einzelvektorprozessor dann mit vielen anderen einzelvektorprozessoren zu den großen rechner zusammenschalte dann habe ich halt parallelität auf zwei ebenen ich habe einmal diese vektor verarbeitung auf dem system hier und dann die nebenläufige verarbeitung wenn ich mehrere also im Bild vorher war es ein Shared Memory, jetzt haben wir

ein Distributed Memory. Die Idee ist natürlich auch wieder, dass ich Vector- oder Matrixverarbeitung möglichst schnell mache. Und was mache ich? Ich lege also so ein Feld von Prozessoren fest, also diese Punkte hier, das sind meine Prozessoren. Und diese Prozessoren, die werden über Datenströme gefüttert. Zwischenergebnisse ablegen können. Die Prozessoren sind meistens über direkte Nachbarn in irgendeiner Art von Netzwerk miteinander verbunden. Also hier haben wir so ein typisches Nord-Süd-Ost-West-Netzwerk, also vier Nachbarn. Man kann sich da auch andere Strukturen vorstellen. Ich habe also hier so ein Interconnection-Network, damit die untereinander Daten austauschen können oder Daten halt durch die Gegend bewegen, also Data Movement verschieben können aufeinander. Ich habe eine zentrale Kontrollinstanz, einen sogenannten Und der gibt praktisch den Takt vor, der sagt so, jetzt mal alle addieren. Die Daten müssen natürlich auch festgelegt sein, die addiert werden sollen. Und dann habe ich in der Regel in so einem Feld als Prozessor nur zwei Möglichkeiten. Entweder ich mache bei der Addition mit oder ich mache nicht mit. Das heißt, ich habe keine eigene Intelligenz, wenn man so will. Ich bin streng getaktet mit allen anderen. Und welchen Befehl ich auszuführen habe, das kommt von einer zentralen Kontrollinstanz. diesen Befehl mit auszuführen oder eben nicht mit auszuführen. Das sind dann die Feldrechner. Ja und es ist klar, dass diese Hardware für bestimmte Algorithmen sehr toll funktioniert, also alles was Vektor-Matrix basiert ist, da ist diese Hardware natürlich toll dafür, aber im General Purpose Einsatz natürlich ihre Schwächen zeigt, ja also alles was eben nicht Matrix oder Vektor organisiert ist, Das heißt, eigentlich ist so eine Architektur eher zu sehen als Co-Prozessor für Vektorberechnungen und nicht als General Purpose Prozessor. So, jetzt gibt es dann noch die Shared Memory MIN Rechner. zentrale Kontrollinstanz habe, sondern ich habe hier unabhängige CPUs. Jede CPU berechnet ihr eigenes Programm und ist eben über das Netzwerk mit dem Speicher verbunden, wobei es für dieses Netzwerk eben ganz unterschiedliche Ansätze gibt. Einmal kann ich so ein Netzwerk natürlich aufbauen als Crossbar. Crossbar ist natürlich toll, weil jede CPU mit jedem Memory-Modul 1 zu 1 verbunden ist. viele CPUs können gleichzeitig mit dem Speicher arbeiten. Das einzige Problem, was ich beim Crossbar habe, das ist quadratisches Wachstum der Komponenten.