panikzettel.htwr-aachen.de

Einführung in High-Performance-Computing

Version 3 | 19.02.2024

Jonas Schneider

Inhaltsverzeichnis

1 Einleitung

Vorwort

Dieser Panikzettel ist über die Vorlesung „Einführung in High-Performance-Computing“.

Dieser Panikzettel ist Open Source auf https://git.rwth-aachen.de/jonas.max.schneider/panikzettel . Wir freuen uns über Anmerkungen und Verbessungsvorschläge (auch von offiziellen Quellen).

Warum High-Performance-Computing?

Es gibt mehrere Gründe:

- Viele Wissenschaftsfelder benötigen (häufig vor allem Simulationen) viel Rechenleistung und komplexe Algorithmen.
- Aber auch AI Anwendungen benötigen hohe Rechenleistung.
- Parallelprogrammierung wird immer wichtiger und lässt sich auch auf anderen Entwicklungen anwenden (z.B. Spiele).
- Sie sind cool.

Rankings verschiedener Supercomputer

Passiert mittels verschiedener *benchmarks*. Der Populärste (nicht unbedingt der aussagekräftigste) ist der *HPL Benchmark*[1]. Er wird auch für das berühmte TOP-500 List Ranking benutzt. Andere benchmarks sind z.B.: „STREAM“ für memory performance, HPCG für realistischere memory-intensive computations.

Sidenote: Moore's Law

Transistorzahl pro Chip wächst mit $1,59^{\text{year}-1959}$ Die Frequenz der Prozessoren nimmt Abwärme bedingt praktisch nichtmehr zu (nur noch ganz langsam), Da aber immer weitere Kerne zu einem Prozessor hinzugefügt werden, steigt die Transistorzahl trotzdem.

1.2 Prozessorarchitektur

High-Performance-Computing ist tief mit den einzelnen Prozessorfunktionalitäten verwurzelt. Deswegen müssen wir die uns hier auch anschauen.

Definition: *Von Neumann Architecture*

- Einheitlicher Speicher (Für Instruktionen und Daten)
- *Control unit* holt Instruktionen vom *Memory*
- *Arithmetic Logic Unit* (ALU) executes the given instruction on some data.

Heutige Prozessoren sind deutlich komplexer, halten sich aber immer noch an die Architektur. Eine CPU besteht aus *control unit* + *ALU*. Ein wichtiger Aspekt: Die *memory hierarchy* (hierzu später im Abschnitt 1.3).

Wenn ihr von x86-64(bzw. amd64), risc oder auch arm64 lest ist dies die *instruction set architecture*. Er zeigt welche Funktionen die CPU ausführen kann (also z.B. `ADD`, `Load`, `VFMADD132PD`.

Aufgeilt sind die verschiedenen Instructions sets in *Complex Instruction Set Computer* (CISC) z.B.`x86-64`. Die Instruktionen haben unterschiedliche Breite und hier können Instruktionen auch gut mal mehrere Unteroperationen ausführen.

und in *Reduced Instruction Set Computers* (RISC) z.B. `arm`, `risc`. Sehr viele „gleich große“ *instructions*.

[1] https://www.netlib.org/benchmark/hpl/

Definition: Little vs Big endian

Big endian **Most** significant bit an der Untersten Adresse
Little endian andersherum...

Pipelining

(Grob) während z.B. die *ALU* gerade eine Multiplikation berechnet, hat die *control unit* und der Speicher gerade wenig zu tun. Selbst intern in der ALU gibt es mehrere Zonen z.B. für Addieren, Multiplizieren, etc.

Das ***Pipelining*** überlappt mehrere Instruktionen die an getrennte *stations* ausgeführt werden. Die Instruktionen dauern natürlich gleich lang, aber sie werden halt nicht **nur** sequenziell abgearbeitet. Es können auch nur unabhängige Instruktionen gepipelined werden, somit muss zu Out-Of-Order execution gewechselt werden.

Positives (+)	Negatives (-)
• Höherer *throughput* (performance)	• Die Pipeline muss erstmalig gefüllt sein (*wind up-phase*) • Benötigt unabhängige Instruktionen • Benötigt *instruction scheduling* wegen der *Out-Of-Order execution.* • Komplexe ISAs können schwierige Instruktionen haben, die die Pipeline aufhalten d.h. *pipeline stalling*. Z.B. Sqrt.

Natürlich kann code für pipelining optimiert werden. Zum Beispiel können die einzelnen Instruktionen versetzt werden, um mehrere Unabhängige Instruktionen zu haben.

Superscalarity

Definition: Superscalarity

Ein Prozessor ist *superscalar*, wenn er dafür gebaut wurde mehrere Instruktionen pro *clock cycle* auszuführen. Es ist eine Form des *instruction level parallelism.* Moderne Prozessoren sind 3-6 fach *superscalar.*

Auch um das auszunutzen braucht es optimierten code. Z.B. können Abhängigkeiten entfernt werden.

SIMD

Ebenfalls ein tolles Thema.

Definition: Single Instruction Multiple Data (SIMD)

Mittels spezieller Vector Instruktionen und Registern kann eine Instruktion auf mehreren Daten (einem *vector*) ausgeführt werden, daher kommt der Name *Single Instruction Multiple Data* (SIMD). Die Instruktionen müssen nicht zwangsläufig parallel ausgeführt werden, sind aber transparent für den Nutzer.

Die SIMD instructions für x86-64 werden in folgende *instruction set extensions* aufgeteilt:
- SSE: 128bit Vector Register
- AVX(2): 256bit Vector Register

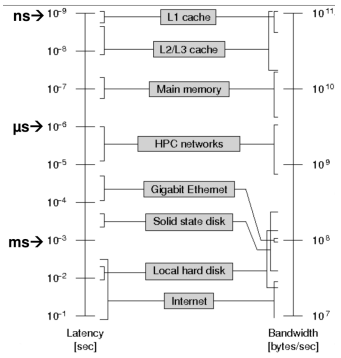

Abbildung 1: Memory Hierarchy

- AVX512: 512bit Vector Register

Eine beliebte Technik für SIMD Optimierungen sind die *loop unrollings*.

```
for(i=0; i<n; i+=4)
{
  C[i] = A[i]+B[i];
  C[i+1] = A[i+1]+B[i+1];
  C[i+2] = A[i+2]+B[i+2];
  C[i+3] = A[i+3]+B[i+3];
}
```

Hier kann dann aber der Compiler leichter sehen: ok diese 4 Operationen können in eine SIMD instruction zusammengefasst werden. Wir müssen den Compiler entweder mit Flags oder mit z.B. `#pragam omp simd` mittels Directives und OpenMP sagen, das er Vektorisieren soll.

1.3 Memory hierarchy

Da der Speicher (DRAM) nicht besonders an Geschwindigkeit zunimmt und besonders eine hohe Latenz (schon allein wegen der Entfernung zu CPU) hat, wurden einige *caches* eingefügt und eine Hierarchy gebildet, siehe Abbildung 1.

Zu sehen ist, dass Level-1 (L1) cache schneller ist als level Level-2 cache, dafür ist er aber auch wesentlich kleiner 32KB vs 256KB vs 30MB (L3) (unterschiedlich pro Prozessor und Generation).

Ganz so einfach ist es aber nicht. Am schnellsten sind natürlich die Register, auf denen die Berechnungen laufen. Der L1 cache ist dann aber auch pro Kern private, der (L2) L3 cache wird aber von allen Kernen geteilt. Die Caches sind meistens mittels *hash-tables* implementiert, da sie unterschiedliche Adressen in dem Speicher haben: *TAG RAM* ist die Speicheradresse und *DATA RAM* die Daten.

Hierbei gibt es zwei Strategien zu cachen:
Temporal Locality Ein Item was gerade referenziert wurde, wird bald bestimmt nochmal verwendet
Spacial Locality Wenn ich `a[0]` vom Memory hole, brauche ich bestimmt auch bald `a[1...]`.

Und auch die Daten im Cache zu ersetzen:
LIFO Last In First Out
FIFO First In First Out
Random ...
Most Recently Used ...
Not Recently Used ...

Cache-Lines kümmern sich hierbei um *spacial locality*. Wenn ich Daten vom Memory oder einem höheren cache hole, hole und cache direkt die nächsten Elemente mit. Die wird in der mit 64 Byte gemacht.

Die **Access-Time** kann dabei wie folgt berechnet werden:

$$G(\tau, \beta) = \frac{T_m}{T_{av}} = \frac{\tau}{\beta + \tau(1-\beta)}$$

mit T_m, T_c ist die Access-Time für den Cache/ dem Speicher. $T_{av} = \beta T_c + (1-\beta)T_m$ ist die Average Access-Time, wobei β die Hit-Rate, also der Anteil der Daten, die durch den Cache bereitgestellt werden. $\tau = \frac{T_m}{T_c}$ ist somit der Speedup bei Benutzung des caches.

Wir versuchen natürlich die Cache-Misses zu minimieren. Zum Beispiel mittels *prefetching*, indem wir Daten kontinuierlich vom Speicher holen.

Cache coherence

Wenn mehrere Kerne auf die gleichen Daten zugreifen/schreiben, müssen die einzelnen L1,2,3 caches gelehrt werden, da sonst Inkonsistenzen entstehen können. Dies passiert entweder durch *Directly based* indem an einem zentralen Punkt gespeichert wird, oder durch *Snooping* indem caches untereinander Kommunizieren.

2 Basic Optimization

Bei Optimierungen sollte immer nach dem Schema vorgegangen werden:
1. Datenerfassung (*profiling & monitoring*)
2. Datenanalyse (warum ist der Code langsam)
3. Optimierung der *hot spots*
4. Test and Repeat

Versucht nicht ernsthaft zu optimieren ohne *profiling*. Ihr könnt nur schätzen wo die Hotspots sind.

Sidenote: Optimierungs Tips by Jonas

Ganz grob sollte es bei Optimierungen folgende schritte geben:
1. Datenstrukturen und Algorithmen. Macht keine Arbeit die auch einfacher ginge. (Das macht meistens den größten speedup aus)
2. Serial Performance generell.
3. SIMD und Caching (kleine Tweaks meistens)
5. Multi-core performance

2.1 Monitoring

Profiling & Monitoring sind essentiell.

Event-driven Measurement Messe immer wenn bestimmte events passieren. (-Overhead bei vielen Events [und schwer zu berechnen])

Time-driven Measurement Messe immer in einem bestimmten Zeitabstand. (+Overhead ist berechenbar, - Wir könnten Events verpassen)

Definition: Sampling

Interrupting einer Applikation um deren Zustand zu erfassen. (*Time-driven*)

Definition: Instrumentation

Einfügen von monitoring in die Applikation.

Instrumentalisierung kann manuell, durch *pre-instrumented libraries* oder compiler-driven geschehen.

Definition: Profiling

Erstelle ein „performance profile“ von einer bestimmten Instanz. z.B. bei Funktion-Profiling von (manchen oder allen) Funktionen. Enthalten sind z.B. Metriken wie *runtime*, *bytes written*, etc.

Definition: Tracing

ein „Trace“ ist eine *time-ordered list* z.B. über den Programm Zustand. Braucht deutlich mehr Speicher als ein *profile*.

Hardware Counters

Die Hardware zählt teilweise selber manche Metriken wie z.B. L1-Hitrate, Bus-Transactions (= cache line transfers), Loads & Stores, etc.

2.2 Obviously

- Tue weniger. Z.B. `break` in loops.
- Weniger teure Operationen. Z.B. Lookup table
- Weniger Speicherverbrauch => Mehr cache hits
- Ausgliedern von (teuren) Berechnungen.
  Z.B. `a[i]+ a[i]+tmp` in einem loop statt `a[i] = a[i] + s + r*sin(x)`
- Avoid Branches. Branches müssen durch den Prozessor predicted werden (wegen dem prefetching der Instruktionen). Somit kann die CPU auch falsch liegen. Zudem verhindert es häufig eine Vektorisierung (also SIMD).
- SIMD nutzen. Hierfür braucht es *aligned* Daten gleicher Länge (z.B. 4 Byte). Und es hilft wenn sie im speicher an Adresse % (4,8,16) = 0 Adressen liegen. Es kommt noch vielmehr dazu und SIMD ist tricky.
- Compiler Optimierungen nutzen: Compiler sind wirklich gut, aber man sollte es ihnen einfach machen. (`-O3` z.B. aktivieren)
- Avoid Allocation/Deallocation: … Abschnitt 2.3

2.3 Daten Optimierung

Speicheroperationen sind sehr häufig der limitierende Faktor. Die Bandwidth (GB/s) und die Latenz des Speichers ist nicht so stark gestiegen wie die Prozessorleistung (selbst mit cache)

Definition: theoretical peak performance

$$P_{\text{peak}}[\text{FLOP/s}] = \#\text{cores} \cdot \text{frequency} \cdot \frac{\text{FP operations}}{\text{cycle}}$$

wir meinen immer double precision

Definition: Machine/Code Balance

$$B_m = \frac{b_s}{P_{\text{peak}}}$$

wobei B_m die Machine Balance ist und b_s die Memory Bandwidth [words/s = 64/8/s = 8byte/s], also das Verhältnis zwischen Speichergeschwindigkeit und Prozessorgeschwindigkeit.

$$B_c = \frac{\text{data transfers (LOAD, STORE)}}{\text{arithmetic operations}}$$

Für einen bestimmten Code abschnitt.

Im STREAM Triad benchmark `for(i=0;i<N;++i){ a[i] = b[i] + s * c[i]; }` $\Longrightarrow 1.5 \frac{\text{Words}}{\text{FLOP}}$

Definition: Lightspeed

Erwartete Peak Performance $l = \min\left(1, \frac{B_m}{B_c}\right)$.

Im STREAM Triad benchmark und dem Intel Xeon E5-2650v4: $0.015 = 1.5\%$ von P_{peak}

Mit diesen drei Definitionen erzeugen wir uns das ***Roofline Model***

$$P = P_{\text{peak}} \cdot l = \min\left(P_{\text{peak}}, \frac{b_s}{B_c}\right)$$

ist die maximal erreichbare peak performance

$$I := \frac{1}{B_c}$$

ist die operational intensity.

Das Roofline model ist dann

$$P = \min\left(P_{\text{peak}}, I \cdot b_s\right)$$

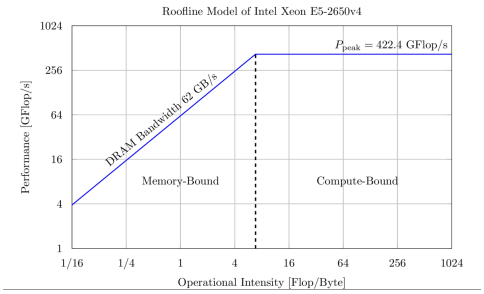

Abbildung 2: Roofline Model. Credit: HPC Folien

Die Abbildung 2 beschreibt ein Roofline-Model.

Memory-bound Probleme Der erste Teil der Abbildung. Probleme sind durch die Speichernutzung limitiert. (Häufiger)

Compute-bound Probleme Der zweite Teil. Die Prozessorleistung ist der limitierende Faktor.

Hierbei wird natürlich, bedingt durch den *lightspeed*, nur der langsamste Datenweg modelliert.

2.4 Algorithmen

Wir wenden hier die Big-O Notation an. Die sollte durchaus bekannt sein.[2]. Allerdings trennen wir erneut die Speicher und die Compute Zeit (⚠ zuerst kommt arithmetic operations). Beispiele:

- $O(N)/O(N)$ wie Vektor addition, SpMV mult., etc. Performance meistens „memory-bound“ da der Speicher langsamer ist. Compiler optimiert hier leicht
- $O(N^2)/O(N^2)$ dense matrix-vector multiply, matrix addition. Schwer zu optimieren
- $O(N^x)/O(N^2) \quad x > 2$. dense matrix-matrix Multiplikation. Großes Optimierungspotential. Zunächst mal nicht memory bound.

Sparse Matrices

Wenn fast alle Elemente von Vektoren und vor allem Matrizen = 0 sind heißt eine Matrix *sparse*.

Wenn wir eine sparse matrix speichern, können wir dies auszunutzen und nur die $\neq 0$ Werte speichern mittels des Compressed Row Storage (CRS) Formates

Als Beispiel $A = \begin{pmatrix} 1.1 & 1.2 & 0 & 0 \\ 0 & 2.2 & 0 & 0 \\ 0 & 3.2 & 3.3 & 0 \\ 0 & 0 & 0 & 4.4 \end{pmatrix}$ würde man in die Folgenden Arrays kodieren:

- Row pointer: `A.ptr: int[] = [0,2, 3, 5, 6]` (der letzte gibt das Ende der letzten Reihe an)
- Column index: `A.index: int[] = [0,1, 1, 1,2, 3]`
- Value: `A.value: double[] = [1.1,1.2, 2.2, 3.2,3.3, 4.4]`

Es gibt andere Formate und gute Algorithmen für sparse matrix Operationen.

3 Parallelprogrammierung

Es gibt grob zwei Arten der Memory Architektur:

- Distributed: Jeder hat sein eigenes Memory areal (eigenes *address spaces*)
- Shared: Teilen eines *address spaces*
- Hybrid: Eine Kombination aus beiden

	Single Instruction	Multiple Instruction
Single Data	SISD Serial Execution	MISD (nicht praktikabel)
Multiple Data	SIMD	MIMD (multi processor architecture)

3.1 Limits of scalability

[2]Sonst guck dir den DSAL Panikzettel an https://panikzettel.htwr-aachen.de/dsal.pdf

Satz: Amdahl

Part-I

$$\text{Speedup } S_{p(N)} = \frac{T(1)}{T(N)}$$

Hier ist $T(x)$ die Zeit die es braucht um das Programm mit x Prozessoren auszuführen.

$$\text{Effizienz } E_{p(N)} = \frac{S_{p(N)}}{N}$$

Part-II Amdahls Law (**strong scaling**)
Amdahl geht von einem perfekt parallelen Teil p und eine nicht parallelisierbaren Teil s aus $s + p = 1$

$$S_p = \frac{T(1)}{T(N)} = \frac{T(1)}{\left(s + \frac{p}{N}\right) \cdot T(1)} = \frac{1}{s + \frac{1-s}{N}}$$

Im besten Fall gibt es einen linear speedup. Also falls ich 2 Prozessoren habe, ist das Programm doppelt so schnell.

Zwei wichtige Punkte zu Amdahl:
- Kein Code ist perfekt parallelisierbar
- Das Workload nimmt höherem N zu. Bzw es bleibt Konstant pro Prozessor. Wir teilen also ein 100000 nicht auf, sondern jeder Prozessor berechnet 100000 Elemente.

Satz: Gustafson's Law

Wir teilen die Arbeit auf die unterschiedlichen Prozessoren auf. Erneut gilt $s + p = 1$.

$$S_p = \frac{(1-p) + Np}{(1-p) + p} = Np + s$$

$$\varepsilon_{p(N)} = \frac{(1-p)}{N} + p$$

Auch bekannt unter „weak-scaling“

Daraus folgt, das wenn wir N erhöhen, wird der Serial part immer unwichtiger. Und der Speedup geht gegen unendlich (natürlich unrealistisch)

3.2 Multithreading

Hyperthreading ist das wir zwei Control-Units und doppelte Register haben und zwei Threads laufen lassen. Die ALU wird aber geteilt. Somit wird ein Kern besser ausgenutzt.

3.3 Multi-Core Prozessoren

Ein Prozessor, der mehrere Kerne hat ist ein *Multi-Core Prozessor*. Die Kerne können auch unterschiedlich performant/effizient sein.

Hier muss natürlich auf Memory richtig zugegriffen werden:

UMA Unified Memory Architecture: Nur ein Memory wird via shared bus zugegriffen. Der Bus sollte cross-linken können, sodass mehrere Kerne gleichzeitig zugreifen können.

ccNUMA Memory ist verteilt. Per *interconnect* kann auf den anderen Memory zugegriffen werden. Meistens liegt ein NUMA node näher ein einem Kern (ist schneller). Die ganze Sache ist aber völlig transparent für den User.

3.4 Network

Bandwidth: $B_{\text{eff}} = \frac{N}{T_L + \frac{N}{B}}$. B ist die max. Bandwidth, N die Nachrichtenlänge in Bytes und T_L die Latenz. Die Network performance ist natürlich von der Netzwerktopologie (Bus, Ring, etc.) abhängig.

Topologie	Max degree	Edge connectivity	Diameter	Bisection Bandwidth
Bus	1	1	1	B
Ring	2	2	$\lfloor \frac{N}{2} \rfloor$	2B
Fully-Connected	$N-1$	$N-1$	1	$B\lfloor \frac{N^2}{4} \rfloor$
Fat-Tree	depends	1 w/o redundancy	$2 \cdot \text{level}$	depends
Mesh	$2d$	d	$\sum_{i=1}^{1}(N_i - 1)$	$B\left(\prod_{i=1}^{d-1} N_i\right)$

Tabelle 1: Topologieübersicht

4 Parallelisierungsoptimierungen

Concurrency Ausführung eines Tasks ist nicht fest-gelegt (kann auch gleichzeitig sein). Eigenschaft eines Programms

Parallel Berechnung können simultan geschehen. Eigenschaft einer Maschine.

Unterschied zwischen Prozessen und Threads sollte durch BUS und PSP bekannt sein.

Es gibt verschiedene Wege wie wir parallelisieren können. Auch Bibliotheken (z.B. OpenMP) unterstützen häufig mehrere Wege.

- SPMD: Single Program, Multiple Data
- Loop Parallelism: Wir teilen eine Loop in Teile auf und bearbeiten die in Threads
- Master/Worker: Ein Master vergibt an viele Worker aufgaben
- Fork/Join: Fork do stuff join back

All das waren Support Strukturen. Es gibt aber auch die Algorithmischen Strukturen die auf den Support Strukturen aufbauen.

- Tasking: Wir erschaffen Tasks die dependencies haben können.
- Divide/Conquer: logisch
- Geometric decomposition: ja keine Ahnung.
- Pipeline

Geometric decomposition

Wir teilen z.B. ein Bild oder eine Matrix auf. Zumindest solange die Indexe nicht-überlappen. Bzw. sie können überlappen (wie bei Gaussian Blurs) müssen dann aber unabhängig voneinander sein. Z.b. convolutions bei einem convolutional-neural-network

Divide&Conquer

Basically Merge-Sort.

Sidenote: Merge-Sort

Bei Merge-Sort lässt sich das *mergen* der Daten ebenfalls mit Divide&Conquer lösen. Aber beim splitten ist immer darauf zu Achten, dass es meistens einen Punkt gibt, wo ein Serieller Code schneller ist. Dort sollte man dan ein Threshold anlegen.

Pipelining
Gleiches Prinzip wie bei Hardware Pipelining, pipelines laufen sequenziell, aber können versetzt gleichzeitig ausgeführt werden.

Bei allen Parallelisierungen müssen wir darauf aufpassen die Berechnung ungefähr gleich zu verteilen, da wir sonst an Effizienz verlieren. Bei Loop Parallelisierungen z.B. in sparse matrix-vektor Multiplikation, dass in dem erstem Bereich ähnlich viele Non-Zeros wie in allen anderen Bereichen sind. OS Jitter ist in diesem Bereich der Begriff, dass das OS natürlich auch aufgaben zu tun hat und unser Programm unterbricht. Minimiere also das OS wenn möglich. Load Balancing ist ebenfalls möglich per runtime zu machen und dynamisch die Aufgaben zu partitionieren. Wie genau ist für den Panikzettel irrelevant (guckt euch die Folien an)

Zudem müssen wir uns um synchronization und somit auch um Korrektheit kümmern.

5 OpenMP

less goo. OpenMP ist eines der meist genutzten Bibliotheken im HPC Sektor. Sie ermöglicht es (leicht) parallelen C/C++ oder Fortran 77/90/95+? zu implementieren. Sie läuft hierbei über Direktiven, heißt wir annotieren nur unseren bereits geschriebenen Code und der OpenMP compiler (integriert in alle üblichen mit Flag `-fopenmp` oder `-qopenmp`) fügt dann den Code ein.

Bei OpenMP Programmierung nehmen wir eine shared-memory Modell an. Alle Kernen greifen also auf einen Memory.

OpenMP start mit einem einzelnen Thread dem *Master Thread*. Von dem werden mit Parallel Regionen dann weitere Threads gespawnt. Also sehr nach dem Konzept Fork/Join.

Beispiel

```
#pragma omp parallel
{
  // parallel stuff
}
```

Die Thread Zahl wird meistens per env variable `OMP_NUM_THREADS` gesetzt.

Oft nutzen wir aber auch Worksharing, also nach dem Konzept Loop-Parallelism:

```
#pragma omp parallel
#pragma omp for
for(int i=1; i < N; i++){
  a[i] = b[i]+c[i]
}
```

Sei `OMP_NUM_THREADS=4`, $N = 100$.

Hier wird das Standardmäßig aufgeteilt in $[0, 25), [25, 50), [50, 75), [75, 100)$. Kann aber mit `#pragma omp for schedule([static|dynamic|guided], [chunkSize])` geändert werden.

Die beiden Sachen können auch kombiniert werden:

```
#pragma omp parallel for
...
```

Für Synchronization stehen auch Direktiven zur Verfügung.

```
#pragma omp critical
```

Hier darf jeweils nur ein Thread gleichzeitig rein.

```
#pragma omp barrier
```

Hier warten alle bis alle Threads bei der Barriere angekommen sind.

```
#pragma omp single/master
```

Wir nur von einem Thread bzw. dem master erledigt.

5.1 Data scoping

Um Korrektheit zu bewahren müssen manche Daten für Threads privat sein, da sie sonst von anderen überschrieben würden.

Das alles ist ganz schön viel wir werden hier nur über ein paar Sachen berichten.

- `shared`: Daten werden über alle threads geteilt, Default für parallel Region.
- `private`: Daten sind private. D.h. eine neue Instanz wird erstellt für jeden Thread
  - `firstprivate`: Privat, aber vor dem ersten construct werden die Daten kopiert, Default für tasks
  - `lastprivate`: Privat, aber am Ende werden die Daten zurück kopiert.

Loop control vars sind `private`. `i` im Beispiel oben. Statische vars sind `shared`

Falls wir so einen Code hier haben

```
int i,s = 0
#pragma omp parallel for reduction(+s)
for(i=0; i < 100; i++){
  s = s + a[i]
}
```

Wir summieren also einfach nur `a` auf. Da das aber eigentlich ja schlecht parallelisierbar ist, gibt es die `reduction` clause. Hier wird dann für alle Intervalle $[0, 25), \ldots$ ein eigenes s_{priv} gemacht, dann $a[0] + \ldots + a[24]$ aufsummiert und dann die 4 s_{priv} summiert. Dies verhindert, wenn man es richtig anstellt, auch das False-Sharing, also das invalidieren der Caches.

Tasking

Zusätzlich zum Worksharing besitzt OpenMP auch Tasking Funktionalitäten.

```
int main(int argc, char *argv[]){
  #pragma omp parallel
  {
    #pragma omp single
    {
      fib(input)
    }
  }
}
int fib(int n){
  if (n < 2) return n;
  int x,y;
  #pragma omp task shared(x) if(n<30)
  {
    x = fib(n-1);
```

```
  }

  //hier könnten wir noch einen Task machen, brauchen wir aber nicht, da dieser Task
hier auch noch etwas berechnen kann
  y=fib(n-2)
  #pragma omp taskwait
  return x+y
}
```

Das Beispiel sollte verständlich sein. Die erste `parallel` Region machen wir auf, damit die anderen Tasks auf die Threads aufgeteilt werden. Dabei wird das `if` verwenden um z.B. einen Threshhold zu bauen, ab wann wir seriell arbeiten wollen. Warum das? Einen Task zu erstellen ist nicht wenig Arbeit (einige Tausend Instruktionen) noch besser wäre ein `if(n<30) return serialFib(n)` nach dem Rekursionsanker.

Die `taskloop` Direktive teilt eine Loop in tasks auf, also so wie for aber statt threads, tasks. Ebenfalls gibt auch noch `taskyield`. Der Tasks kann also suspendiert werden.

6 MPI

Um nicht nur Kerne eines Prozessors, sondern auch mehrere Prozessoren für Berechnungen zu nutzen müssen wir uns um die Kommunikation zwischen ihnen kümmern. Message Passing Interface (MPI) ist eine Bibliothek die uns Nachrichtenverkehr zwischen Prozessen ermöglicht. Egal ob die auf dem gleichen Prozessor laufen.

Dies ist für SPMD, Single Program, Multiple Data wichtig. Hier haben wie ein Programm und lassen via mehreren OS Prozessen auf verschiedenen Datenteilen laufen. Hierbei kommen häufige Nachrichten vor:

- `send(data, destination [id])`: Sende Daten
- `recv(data, src)`: Warte auf und empfange Daten
- `bcast(data, root)`: Broadcast Daten vom Root runter.
- `gather/scatter(data/subdata, subdata/data, root)`: Verteile/Sammle Daten zu subdaten in allen unter Root.

Teil von MPI sind unter anderem:

- Identifiers, also wie die Prozesse heißen.

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char **argv) {
  int rank, nprocs;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

  printf(“Hello, MPI! I am %d of %d\n”,
    rank, nprocs);

    MPI_Finalize();
  return 0;
}
```

Das `...Comm...` steht hierbei für Communicator. Ein Communicator teilt die Prozesse ein und verteilt dort Nachrichten. Zwei sind vordefiniert:
- `MPI_COMM_WORLD`: Alle gestarteten Prozesse
- `MPI_COMM_SELF`: Nur der aktuelle Prozess. (für IO Zeugs)

Fehler werden bei MPI mit Rückgabe von `MPI_SUCCESS` bzw genau dann nicht gelabeled.

Eine MPI Nachricht besteht aus:
- Message Content
- Envelope
  - Sender Rank
  - Receiver Rank
  - Tag (Zusatzinfos)
  - Communicator

Die Nachrichten werden in korrekter Reihenfolge übertragen.

Hierfür wird `MPI_Send(buf, count, datatype, dest, tag, comm)` verwendet. Die Datentypen werden mittels eigener Language Bindings eingetragen. Heißt soviel wie bei `int buf[count]` sagen wir `MPI_INT`.

Für Empfangen `MPI_Recv(buf, count, datatype, src, comm, status)` `count` ist die Anzahl die `buf` halten kann. Die Tatsächliche Größe kann mittels `MPI_Get_count` abgefragt werden.

Wenn aber beide Prozesse erst Datensenden und dann Austauschen wollen gibt es einen Deadlock. Hierfür gibt es die `MPI_Sendrecv([send args], [recv args], comm)`.

Oder man verwendet *non-blocking* message passing mittels `MPI_Isend, MPI_Irecv`. Hierbei ist `status` mit `req` getauscht, da wir über den Status natürlich erst später Bescheid wissen: `MPI_Wait(MPI_Request *request, MPI_Status *status)`. Mit `MPI_Waitany/Waitsome/Waitall` lassen sich mehrere *request* angeben und es wird gewartet bis einer/eine gegebene Anzahl/alle fertig ist/sind. MIt `MPI_Test` auch any,some and all, lässt sich non-blocking queries ob die Übertragung fertig ist.

Die einzige Synchronizationsmöglichkeit ist `MPI_Barrier(MPI_Comm comm)`.

MPI hat auch eigene Datentypen und implementiert die ganzen Broadcast/Gather/Scatter Sachen, aber das ist zu viel für diesen Panikzettel. Die Basics sind da.

6.1 Hybride Systeme

OS Prozesse sind natürlich etwas aufwendiger und MPI ist ja nur das Message Passing Interface, also die effiziente Ausnutzung der CPU ist nicht dabei. Hier können wir aber beides MPI und OpenMP verwenden. OpenMP für Multithreaded, SIMD und Cache effiziente Programme und MPI für die Multi-node Nutzung. Beispiel:

```
#pragma omp parallel for
for(k = 0; k < N; k++)
{
  // Parallel work done here by OpenMP
  // (e.g. updates over a local stencil)
}

// Halo exchange done by single thread via MPI
MPI_Irecv(halo data from –dir neighbor)
MPI_Isend(data to +dir neighbor)
MPI_Irecv(halo data from +dir neighbor)
```

```
MPI_Isend(data to -dir neighbor)
MPI_Waitall();
```

Unter Hybride Systeme versteht man aber genereller die zusammensetzung von mehreren Systemen. Z.B. Auch MPI + CUDA für GPU Programmierung etc.

MPI kann verschiedene Thread Unterstützungen. `MPI_THREAD_SINGLE/FUNNELD` (nur einer/der master kann), `MPI_THREAD_SERIALIZED` (nur einer gleichzeitig), `MPI_THREAD_MULTIPLE`(alle)

7 Accelerators

Werden immer Populärer (denke ich). Hierunter zählen auch GPUs (genauer General Purpose GPUs (GPGPUs)) aber auch FPGAS oder Intel Xeon PHI coprozessoren.

Ganz schneller GPU Überblick
lets go.

Eine GPU hat im Vergleich zu einer CPU viel viel mehr Kerne, die (teilweise) exakt gleichzeitig laufen, aber deutlich weniger können und meist langsamer sind.

> Wie ist eine GPU aufgebaut?

GPUs operieren nach dem Prinzip SIMT, Single Instruktion, Multiple Threads. Also mehrere Threads laufen einem Gemeinsamen Program Counter hinterher.

Es werden 32 Threads zu einem *WARP* zusammengefasst. Die Laufen 100% Parallel ein Programm ab. Die ThreadID (ein index 0-31) wird hierbei als Index für Kalkulationen genommen. Beispiel `a[threadIdx.x] = b[threadIdx.x] + c[threadIdx.x]`. Bei Branches werden manche Threads pausiert, falls die Bedingung da gerade nicht stimmt (ineffektiv)

Mehrere Warps wird dann zu einer Gruppe Threads (einer Thread Group) zusammengefügt. Der Warp Scheduler kann einzelne Warps dann auf einer Streaming Multiprocessor (SM) ausführen.

Diese Gruppen werden dann erneut gruppiert zu einem Grid (da teilweise mehrdimensional). Die werden dann auf der gesamten GPU ausgeführt, die aus mehreren SMs besteht, die wiederum aus vielen Kernen besteht.

Das war viel.

Wir programmieren das ganze entweder speziell für die GPU mittels OpenCL oder noch spezieller mit CUDA für NVIDIA GPGPUs oder mittels OpenMP offloading und der OpenMP compiler macht das für uns (teilweise natürlich ineffizienter). `#pragma omp target`

Vorteil von Accelerators sind häufig die schnelleren und optimierteren Speichergeschwindigkeiten. Neue GPUs nutzen z.B. Hight-Bandwidth-Memory HBM das um einiges Schneller ist als gewöhnlicher DRAM.

GPUs sind bei sehr vielen mit Lineare Algebra und somit auch KI echt schnell. Bei allem mit vielen Branches aber nicht (da sie keine Branch prediction haben, sondern dann ein Teil der GPU wartet).

Allerdings muss man bei GPUs darauf achten wann, welche und wie viele Daten von der CPU zur GPU übertragen werden, da die PCIe Connection vergleichsweise langsam ist. Es gibt aber auch distributed memory Konzepte, die aber natürlich nicht immer optimal sind.

`#pragma omp map(to:a[0:n]) / #pragma omp map(from:a[0:n]) / #pragma omp map(tofrom:a[0:n])` Bei `to` wird es vor Anfang des Codeblocks zur Graphik Karte kopiert. Bei `from` wird es am Ende auf den CPU speicher kopiert, bei to from beides einfach.

OpenMPs tasks können auch auf die GPUs warten mittels `depends()` z.B. `#pragma omp target map(...) nowait depends(out:gpu_data)`.

> Auf was außer *branching* sollte man noch achten bei GPUs?

Coalescing. Eine GPU greift mit einer Art cache-line von 128Byte auf den Speicher zu. Wir sollten also darauf achten keine „Lücken“ im speicher zu haben.

statt `struct {x,y}[]`, wo wir nur die x Koordinate brauchen, sollten wir lieber nur `struct{x[], y[]}` oder so nehmen.

8 Energieeffizienz

Computer können nur begrenzt effizient sein.

Satz: Thermodynamische Limits eines NAND Gates

$$E = \ln(2) \cdot k \cdot T$$

wobei $k = 1.380650424 * 10 - 23\frac{J}{K}$ die Bolzmann Konstante ist. Und T die Temperatur (Kelvin).

Ist nur gut zu wissen.

CPU braucht Strom:

$$P = V \cdot I = CV^2 f$$

Wobei V die Spannung ist, f die Frequenz, I die Stromstärke, C die Elektrische Kapazität.

Um eine höhere Frequenz zu erreichen, braucht die CPU eine höhere Spannung (also quadratisch mehr Power).

Ein Prozessor kann in unterschiedlichen Zuständen sein:
- P State: Unterschiedliche Leistung/Geschwindigkeit aber alle voll Funktional. (Und nichtmehr ganz akkurat)
- C State:
  - C0 CPU macht gerade etwas
  - C1 CPU ist IDLE: Clock wird nicht weitergegeben
  - C2 Stop Clock
  - C3, C4 Deep Sleep

Es lohnt sich also schnelle Programme zu schreiben, die die gesamte CPU ausnutzten.