Parallelisierungsarten:

- Datenparallelität: gemeinsame Operation auf homogener Datenmenge
- Taskparallelität: weitgehend unabhängige Daten werden parallel verarbeitet
- Funktionale Parallelität: funktionalle Sprachen, Auswertung mathematischer Ausdrücke

Effizienzrechnung:

T_n: Laufzeit mit n Prozessoren | Speedup: T_1/T_n | Effizienz: Speedup/n

Amdahl's Law: p = paralleler Anteil | s = serieller Anteil | $s+p=1$ | n = # Prozessoren | $Speedup_{max} = T_1/T_n = \frac{1}{s+\frac{p}{n}}$

Gustafson's Law: (große p führen in Wirklichkeit zu höheren Speedups, da p von der #Prozessore abhängig ist) | s_n = serialisierbarer Anteil | s_p = parallelisierbarer Anteil |

$Speedup_{max} = T_1/T_n = \frac{s_n + p_n \cdot n}{s_n + p_n} = s_n + p_n = n + (1-n)s_n$

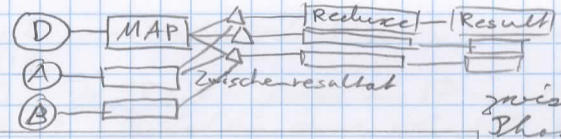
TPL (Task Parallel Library): + Kümmert sich um #Thread, Threadpool, Entwickler konzentriert sich auf den Task, einfache Handhabung (Rückgabewerte, Datenübergabe, Join, ...).

Race Condition: "kritischer Wettlauf". Zwei Systeme wollen den Wert 1 zeitlich parallel um 1 erhöhen. Erwartet wird 3 als Resultat, es kann aber eine 2 stehen wegen der Race Condition. A liest 1 ein, B liest 1 ein, A erhöht →2, B erhöht →2, A schreibt 2 B schreibt 2 ≠ 3 ☡

Parallelisierungshindernisse: - Aliasing: Vorhandensein mehrerer Bezeichner für eine Variable. Wird diese irgendwo geändert so ist dies nicht an anderen Stellen ersichtlich (Zeiger / Referenzen)
- Race Condition: ↑ .

Non-blocking: Es wird nicht geblockt: z.B. Spin-Locks: Der Thread "dreht" bis Lock frei wird. **Lockfree:** non-blocking, wenn ein Thread abgewiesen wird, so kann nicht ein anderer abarbeiten, Starvation möglich **Wait free:** lock free, jeder Thread kommt nach endlicher Anzahl Schritten zum Zug. **Dead Lock:** zwei Threads besitzen jeweils ein Lock, das der andere Thread benötigt → sie sperren sich gegenseitig. **LiveLock:** zwei oder mehr Threads wechseln ständig ihren Zustand ohne weiter zu kommen (spezielle Art des Deadlocks, da der Zustand nicht fest bleibt sondern ständig wechselt) **Starvation:** Bekommt nie die nötigen Ressourcen und kann die Aufgabe nie erledigen

MapReduce: Map + Reduce jeweils parallel

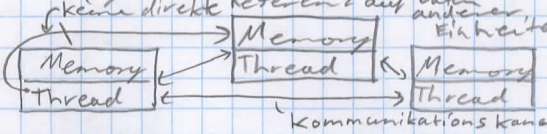


Daten (D, A, B) werden Map-Funktion übergeben. Diese legen Zwischenergebnisse ab (Δ). Ist Map-Phase fertig, so geht es zur Reduce-Phase die je ein Zwischenspeicher "reduced" und das Ergebnis ausgeben.

Beispiel: Eingabe: Texte. Jede Map-Instanz fügt eine 1 in den Zwischenspeicher für Wort "w", wenn es "w" findet. Ist es fertig, zählt Reduce-Instanzen, wie oft "w" vorkam → Ausgabe.

Reader/Writer Locks: Gemeinsame Datenstrukturen werden parallel gelesen/beschrieben. Damit mehrere Threads gleichzeitig lesen können, aber warten wenn etwas verändert wird (write locke komplett, bis fertig sind). Probleme: Stale Data (Reader lesen alte Daten), Starvation (Reader/Writer warten unendlich), Caveat: Nur Sinnvoll, wenn Protokollaufwand nicht alles auffrisst. **Linearisierbarkeit:** ("LIN") Objekt p ist LIN, wenn jeder Methodenaufruf genau einen Wirkungspunkt hat ("Linearisierungspunkt") und es gilt: jeder parallele Ablauf in p ist äquivalent zu einem sequenziellen, bei dem alle Wirkungspunkte nicht überlappender Methodenaufrufe in chronologischer Reihenfolge auftreten. **Sequenzielle Konsistenz:** ("SEQ") Ein Objekt ist SEQ, falls jeder parallele Ablauf äquivalent zu einem sequenziellen Ablauf ist, in welchem alle Wirkungspunkte in der im ausführenden Thread programmierten Reihenfolge auftreten. Schwächer als LIN

MPI:



- Identischer Prozess n Mal gestartet
- keine shared data
- Kommunikation durch explizite Send/Receive

- comm.Send(value, dest, tag);
- comm.Receive(source, tag, ref value);
- comm.SendReceive(sendValue, dest, sendTag, source, recvTag, ref recvValue)
- comm.AllReduce(value, operation); Combine all values using "operation" and send result everyone.
- comm = Communicator.world; (Typ: Intercommunicator)

Interlocked: Bietet atomare Operationen: .Add, .Decrement(), .Increment(), .Exchange(ref location, object newValue) (returns old value), .CompareExchange(ref location, object value, object comparand) (returns the original value): Führt wechselt nur aus, wenn Objekt in "location" == "comparand". Prüft auf erfolgreichen Wechsel returnvalue == newValue. **Pulse/PulseAll:** Pulse wenn alle Threads auf selbe Ressource warten und nur ein Thread kann fortfahren, sobald Ressource frei wird. **Volatile:** Verhindert lokales cachen einer Variable, die mit volatile "markiert" wurde. Auch wird Befehlsreihenfolge durch CPU/Compiler nicht angepasst.

... private int data; private volatile boolean done; void CalcData(){
int ReadData(){ data = 15; ↓④
while(!done); ↓② Reihenfolge in ③ und ② done = true; }
return data; } wichtig. ohne volatile könnte
zuerst "done = true;" gesetzt werden und dann erst
"data = 15;" => falsch

SMP: Symmetric Multiprocessor: ein Cache für alle Prozessoren **NUMA:** Non Uniform Memory Architecture: ein Cache pro Prozessor **Cache:** Hit falls Daten im Cache, Miss falls nicht und aus RAM/HDD gelesen werden müssen **Parallelität:** Ein Prozess wird durch parallel ablaufende Teilprozesse erledigt **Concurrency:** Gleichzeitig ablaufende Prozesse die eine gemeinsame Ressource konkurrieren. **Semaphore:**
int max; // # Ressourcen (gleichzeitige Zugriffe. Am Anfang initialisiert
public P(){ while(max <= 0){ /* wait */ } --max; // Zugriff erlaubt
public S(){ max++; // aufgerufen wenn fertig, 1 Ressource mehr frei
Es gibt auch solche, bei denen man bei P() angeben kann wieviele Ressourcen man reservieren möchte. Ist max = 1, so hat man ein Lock aus einer Semaphore. **CSP: Communicating Sequential Processes:** Konzept, bei dem Prozesse/Threads nur über Channels kommunizieren. **Monitor:** .Enter(object): aquire an exclusive lock on the specified object. .Exit(): release the lock. Pulse()/.PulseAll(), .Wait(object): gibt den Lock frei, wartet bis das Lock wieder zur Verfügung steht und fährt fort. Nützlich wenn Thread mit Lock auf einen Statuswechsel warten muss, der durch einen anderen Thread entsteht, aber dafür den Lock braucht. **.Net Semaphore:** new Semaphore(int initCount, int max); .Release()/.Release(int) -> S()↑, .WaitOne() -> P()↑ kann mehrmals aufgerufen werden. Die Klasse prüft nicht ob der Thread bei .Release() zu viel/wenig freigibt! **Aufgaben:** Sortieralgorithmen für parallelen Zugriff: Quicksort/Mergesort (pattern der Parallelität: Fork-Join) // Wann kann eine for-loop ersetzt werden durch Parallel.For()? -> Wenn die Iterationen unabhängig voneinander sind // Wann dürfen 2 Threads ohne Synchronisation auf einem Array operieren? -> Read-only, wenn es an unabhängigen Orten passiert (Array aufteilen), Monotonie updates // Wieso lohnt es sich bei p cores p+1 Threads laufen zu lassen? -> Cache
Lock-free: kein Sperr Lock, da immer irgendein Thread vorwärts kommt nach endlicher Schrittzahl. Starvation möglich, da keine Garantie welcher Thread vorwärts kommt. // Wait-free: kein Sperrlock und bei Starvation durch Garantie, dass jeder Thread nach endlich vielen Schritten vorwärts kommt. // Waiting for a condition lock(x){ while(!condition) Monitor.Wait(x); } \\ lock(x){ condition = true; Monitor.PulseAll(x); }

ABA Problem: Stack am Anfang: top -> A -> B -> C. T1 pop() und setzt currentTop = A (A ist top, aber gehört nicht mehr zum Stack). T1 wird unterbrochen. T2 macht 2x pop() und push(A). T2 fertig (Stack: top -> A -> C). T1 macht weiter mit CompareExchange: da currentTop und top gleich, wird top auf currentTop.next = B gesetzt. B werde aber von T2 gelöscht -> undef. Verhalten bzw. "zombie object" in .Net.

InvokeRequired: delegate void A(object b);
public void AMethod(object b){
if(form.InvokeRequired){ // im falschen Thread
var d = new A(AMethod);
Invoke(d, new object[]{ b });
} else { /* correct thread, do work */ }
}

nötig bei GUI Komponenten! Methoden sind bereitgestellt durch ISynchronize Invoke (ohne Bindestrich)

Backoff: Thread.Sleep(int) bei erfolglosem Lockversuch.

Lock <-> Monitor: lock(x){ ... } <-compiler-> Monitor.Enter(x); ...; Monitor.Exit(x);

Thread Zustände: Initialized —Start()→ Ready → Running —wait()→ Waiting —Pulse→ Running
Running → Terminated