Teorie

Saturday, January 25, 2025 4:57 PM

Sisteme paralele: cuplare stransa, sincronicitate, sistem de comunicare foarte rapid si fiabil, spatiu unic de adresare; coopereaza pentru a rezolva mai eficient probleme dificile; shared memory, performanta.

Sisteme distribuite: mai independente, comunicare mai putin frecventa si mai putin rapida (asincrona), cooperare limitata, nu exista ceas global; au scopuri individuale si activitati private; scalabilitate, memorie distribuita, comunicare.

Clasificare Flynn

- SISD - simple instruction stream, simple data stream (procesoare clasice, arhitectura von Neumann)
- SIMD - simple instruction stream, multiple data stream (mai multe PU - processing units care executa aceleasi instructiuni pe date diferite)
- MISD - multiple instruction stream, single data stream (procesoare pipeline)
- MIMD - multiple instruction stream, multiple data stream

MIMD

Clasificare in functie de tipul de memorie:

- Partajata (toate procesoarele pot accesa intreaga memorie - global address space - UMA si NUMA)
  - UMA - Uniform Memory Access, CC-UMA Cache Coherent
  - NUMA - Non-Uniform Memory Access, se obtine deseori prin unirea a 2 sau mai multe arhitecturi UMA, care comunica printr-un Bus Interconnect
- Distribuita
- Hibrida

Shared Memory Multiprocessors (SMP) : Mai multe procesoare comunica cu memoria printr-un Interconnect.

Avantaje: global address space, partajare rapida si uniforma.
Dezavantaje: lipsa scalabilitatii, sincronizare in sarcina programatorului, costuri mari.

Parametrii de performanta corespunzatori accesului la memorie

Latenta = timpul in care o data ajunge sa fie disponibila la procesor dupa ce s-a initializat cererea.
Largimea de banda (bandwith) = rata de transfer a datelor din memorie catre procesor.

Cache foarte important ptr SMP, dar necesita Cache Coherency (MESI Protocol).

Arhitecturi cu memorie distribuita

Retea de interconectivitate (communication network), procesoare cu memorie locala.
Avantaje: memorie scalabila, cost redus - retele.
Dezavantaje: responsabilitatea programatorului de a rezolva comunicatiile, acces ne-uniform la date, dificil de a mapa structuri de date mari

Hybrid Distributed-System Memory: retea de SMP-uri care comunica printr-un intercommunication network => cluster.

MPP - Massively Parallel Processor

Fiecare nod este un sistem independent ce are local: memorie fizica, spatiu de adrese, disc local si conexiuni la internet, sistem de operare.

COW - cluster of workstations

Curs 3 bits

Procese in sistemul de operare: ID, stare (activitate curenta), context (se face context switch cand scheduler-ul schimba procesul executat), memorie (cod sursa, date globale/statice, stiva si heap).
Stari: new, ready, running, blocked, terminated.

Threaduri, componente private: ID, stare, context, memorie(doar stiva); componente partajate: cod program, date globale, heap.
Stari: runnable, waiting, timed waiting, terminated.

Race condition = o conditie care poate sa apara intr-un sistem in care comportamentul este dependent de ordinea in care apar anuminte evenimente.

Critical race condition = atunci cand ordinea 'in care se modifica variabilele' determina starea finala a sistemului.
Non-critical race condition = atunci cand ordinea NU are impact asupra starii finale a sistemului.

Data race = o situatie cand 2 sau mai multe threaduri vor sa acceseze aceeasi variabila concurent, si cel putin unul dintre accesuri este de scriere.

Instructiune atomica = instructiune a carei executie nu poate fi intercalata cu cea a altei instructiuni inainte de terminarea ei.

Critical section = o sectiune de cod unde poate avea loc un "data-race"

MPI

Forme de interactiune intre procese/threaduri

- Comunicare
- Sincronizare

Forme de sincronizare

- Excludere mutuala: se evita utilizarea simultana de catre mai multe procese a unei resurse critice.
- Sincronizare pe conditie: se amana executia unui proces pana cand o anumita conditie devine adevarata.

DEADLOCK = situatia in care un grup de procese/threaduri se blocheaza la infinit pentru ca fiecare proces asteapta dupa o resursa care este retinuta de alt proces care la randul lui asteapta dupa alta resursa.

STARVATION = situatia in care unui thread nu i se aloca timp de executie pe CPU pentru ca alte threaduri folosesc CPU.

"Fairness" - toate threadurile au sanse egale de folosire la CPU.

LIVELOCK = situatia in care un grup de procese/threaduri nu progreseaza datorita faptului ca isi cedeaza reciproc executia.

Mecanisme de sincronizare

OPERATII ATOMICE = operatii care sunt efectuate cu o singura unitate, indivizibil.

SEMAFOR = Primitiva de sincronizare de nivel inalt. Are 2 operatii:

- P(s)/down, este apelata cand un proces dorecte sa primeasca acces la o regiune critica, asteapta pana cand V(s) > 0.
- V(s)/up, semnifica eliberarea sectiunii critice pentru alte procese.

Poate fi definit ca o pereche {v(s), c(s)}.
Un semafor slab - c(s) este o multime de asteptare, are operatiile P(s)/down si V(s)/up. Pot aduce starvation.
Un semafor puternic - c(s) este o coada de asteptare, are operatiile P/down si V/up.

Semafor binar - mutex.

Un MONITOR consta din:

- Un set permanent de variabile ce reprezinta resursa critica
- Un set de proceduri ce reprezinta operatii asupra variabilelor (resursele critice pot fi accesate doar prin intermediul monitorului)
- Un corp de initializare
    - Apelat la lansarea 'programului'
    - Apoi monitorul este accesat numai prin procedurile sale

! Numai una dintre procedurile monitorului poate fi executa la un moment dat.

Un monitor poate folosi variabile conditionate pentru sincronizare conditionala.
O VARIABILA CONDITIONALA consta dintr-o coada de blocare si 3 operatii atomice: wait, signal, is_empty.

Thread safety si Shared Resources + Sincronizare Java

Codul care poate fi apelat simultan de mai multe threaduri si produce intotdeauna rezultatul dorit se numeste **thread safe**.
Thread safe => nu contine critical race conditions.

Variabilele locale, daca nu sunt partajate alte threaduri, sunt considerate thread safe.

Thread-safe class, daca comportamentul instantelor sale este corect chiar daca sunt accesate din threaduri multiple, fara sa fie nevoie de sincronizari aditionale (sincronizarile sunt incapsulate in interior).

Forme de sincronizare Java: excludere mutuala (cu synchronized), synchronized static methods, variabile atomice, locks, conditions, semaphore, exchanger, read write lock, cyclic barrier.

Future-Promise

- Accesare sincrona (blocking) sau asincrona (non-blocking).

- The two sided of an asnyc operation: consumer/caller vs producer/implementor, a caller will get a future, the implementor must return a future.

C++
future, promise, async, packaged_task
shared_future - permite accesarea din mai multe threaduri

Java
Task, Executor, ThreadPool, Runnable, Callable (returns a value), Future, FutureTask, CompletableFuture, runAsync, supplyAsync (the task yields a result), thenApply, thenAccept

OpenMP

Metode de evaluare a performantei programelor paralele. Granularitate. Scalabilitate.

$$t_p = \left(\max i : i \epsilon 0..p-1 : T^i{}_{calcul} + T^i{}_{comunicatie} + T^i{}_{asteptare}\right)$$

Sau in cazul incarcarii echilibrate pe fiecare procesor

$$t_p = \frac{1}{p}\sum_{0}^{p-1}(T^i{}_{calcul} + T^i{}_{comunicatie} + T^i{}_{asteptare})$$

Timpul total de executie
$T_{all} = p * T_p$ - timp total
T_s - timp serial
$T_O = T_{all} - T_s$ - timp overhead

Accelerare (speed-up)
Reprezinta raportul dintre timpul de executie al celui mai bun algoritm serial, executat pe un calculator monoprocesor si timpul de executie al programului paralel echivalent, executat pe un sistem de calcul paralel.

$$S_p(n) = \frac{t_1(n)}{t_p(n)},$$
$n - \dim datelor\ de\ intrare, p - nr\ de\ procesoare\ folosite, t_1$
$- timp\ executie\ program\ serial$

$S_p = p - linear\ speedup$
$S_p > p - superlinear\ speedup$

Eficienta
Paramteru care masoara gradul de folosire al procesoarelor.

$$E = \frac{S_p}{p}$$

Legea lui Amdahl
Accelerarea procesarii depinde de raportul partii secventiale fata de cea paralelizabila.

$$Speedup = \frac{1}{seq + \frac{par}{p}},$$
$seq + par = 1$
seq−fractia calculului secvential, par−fractia calculului paralelizabil

Legea lui Gustafon
Atunci cand dimensiunea problemei creste, partea seriala se micsoreaza in procent.
m - dimensiunea problemei

$T_p = seq(m) + par(m) = 1$
$T_s = seq(m) + p * par(m)$

$$Speedup = \frac{T_s}{T_p} = seq(m) + p * par(m)$$

Evaluare timp comunicatie
$t_{comm} = t_s + l * t_h + t_w * m, startup\ time, per - hop\ time, per - word\ transfer\ time$

Sau o varianta simplificata, fara 'cut-through routing'
$t_{comm} = t_s + t_w * m$

Costul
Produsul dintre timpul de executie si numarul maxim de procesoare care se folosesc.
O aplicatie paralela este **optima** daca costul este de acelasi ordin de marime ca timpul celei mai bune variante secventiale, si este **eficienta** daca costul este doar/maxim logaritmic mai mare decat varianta secventiala.

$C_p(n) = t_p(n) * p$

Evaluare **overhead** in cazul **multithreading**: timp crearea, gestiune, oprire threaduri si timp de sincronizare.

Scalabilitate
Scalabilitatea este un parametru calitativ care caracterizeaza atat sistemele parelele (numar de procesoare, unitati de memorie) cat si aplicatiile paralele.

Scalabilitatea aplicatiei = abilitatea unui program paralel sa obtina o crestere de performanta proportionala cu numarul de procesoare.

Scalabilitatea masoara modul in care se schimba performanta unui anumit algoritm in cazul in care sunt folosite mai multe elemente de procesare.

Un indicator important pentru aceasta este numarul maxim de procesoare care pot fi folosite pentru rezolvarea unei probleme.

Scalabilitatea unui sistem paralel (aplicatie software + tip arhitectura) este o masura a capacitatii de a livra o accelerare ca si o functie crescatoare cu parametru numarul de procesoare folosite.

Granularitate
Granularitatea (grain size) este un parametru calitativ care caracterizeaza atat sistemele

paralele cat si aplicatiile paralele.

Granularitatea aplicatiei se defineste ca dimensiunea minima a unei unitati secventiale dintr-un program, exprimata in numar de instructiuni.

Granularitate medie = media tuturor granularitatiilor.
Granularitatea medie a unui algoritm poate fi aproximata ca fiind raportul dintre timpul total de calcul si timpul total de comunicare.

Pentru un sistem paralel, exista o valoare minima a granularitatii aplicatiei, sub care performanta scade semnificativ. Aceasta valoare de prag este cunoscuta ca si **granularitatea sistemului** respectiv.

De dorit:
- un calculator paralel sa aiba o granularitate mica, astfel incat sa poate executa eficient o gama larga de programe.
- programele paralele sa fie caracterizate de o granularitate mare, astfel incat sa poata fi executate eficient de o diversitate de sisteme.

DOP Degree of parallelization - Gradul de paralelism
DOP al unui algoritm este dat de numarul de operatii care pot fi executate simultan:
- Fina (numar mare de operatii executate in paralel)
- Medie
- Bruta

DOP al unui program: numarul de procese/operatii care se executa in paralel intr-un anumit interval de timp.
DOP al unui sistem: numarul de procesoare care pot fi in executie in paralel intr-un anumit interval de timp.

Strategii de Partitionare/Descompunere. Sabloane de proiectare.

Partitionarea

Problema partitionarii are in vedere impartirea problemei in componente care se pot executa concurent.

Strategii de partitionare:
- Descompunerea domeniului de date. Aplicabila cand domeniul datelor este mare si regulat. Ideea centrala este de a imparti domeniul de date in componente care pot fi manipulate independent. Tehnici: prin taiere sau prin incretire - corespund distributiilor liniara, respectiv ciclica.
- Descompunerea functionala = o tehnica de partitionare folosita atunci cand aspectul dominat al problemei este functia, sau algoritmul, mai degraba decat operatiile asupra datelor.

Sabloane de proiectare

Master-Slave (master-worker)

Un task manager imparte taskuri independente catre workeri.

Descompunere Recursiva - Divide&Conquer

In general pentru probleme care se pot rezolva prin divide et impera.

Data Decomposition (geometric) - Descompunere geometrica (a datelor?)

Exista dependente, dar comunicarea se face intr-un mode predictibil (geometric), cu 'vecini'.
Input sau Output.
Owner computes rule - procesul care are data asignata lui este responsabil cu calculele asociate acelei date.

Pipeline

Pipeline - o secventa de stagii care transforma un flux de date.
Unele stagii pot sa stocheze date. Datele pot fi "consumate" si produse incremental.

Paralelizarea pipeline se face prin:

- Executia diferitelor stagii in paralel.
- Executia multiplelor copii ale stagiilor fara stare in paralel.

CUDA

Host = CPU si memoria asociata
Device = GPU si memoria asociata

SIMT = Single Instruction Multiple Threads
Threadurile sunt foarte lightweight, au putin overhead.

Un grid este procesat de intreg device-ul, un block este procesat de catre un SM (streaming multiprocessor), un thread este procesat de catre un SP (streaming processor).

Design for parallel programming

Decomposition/Partition -> Agglomeration (Granularity) -> Orchestration and Mapping

- Skipped a lot of stuff here

Distributed computing patterns

Un **sistem distribuit** poate fi definit ca fiind format din componente hardware si software localizate intr-o retea de calculatoare care comunica si isi coordoneaza actiunile doar bazat pe transmitere de mesaje.

Fault tolerance - how to ensure the correct execution of the application in the presence of faults? The execution should terminate. It should provide the correct result.

Patterns

Client-Server

O componenta de tip server care furnizeaza servicii catre mai multe componente client.
O componenta client care cere servicii de la componenta server.

Stateless sau stateful servers.

Tranzactiile trebuie sa fie:

- Atomice si sa asigure consistenta starii
- Izolate
- Durabile

Fault handling => starea mentinuta la nivelul clientului implica faptul ca toata informatia se va peirde in cazul in care clientul esueaza.

Securitatea poate fi afectata daca starea se mentine la nivel de client pentru ca informatia se transmite de fiecare data.

Scalabilitatea poate fi redusa daca starea se mentine la nivelul serverului 'in-memory'.

Peer-to-peer pattern

Un nod (peer) poate functiona ca si un client - care cere servicii de la alte componente sau ca si server care furnizeaza servicii pentru altii.

Atat clientii cat si serverele folosesc in mod uzual multithreading.

Performanta creste atunci cand numarul de noduri creste, dar scade atunci cand sunt prea putine.

Avantaje:

- Nodurile pot folosi capacitate intregului sistem, chiar daca fiecare are o capacitate proprie limitata
- Overheadul de administrare este scazut.
- Asigura scalabilitate foarte buna si este rezilienta la esecul componentelor individuale.
- Configurarea sistemului se poate schimba dinamic (un peer poate intra sau pleca in timp ce sistemul functioneaza)

Dezavantaje:

- Nu exista garantia calitatii serviciilor
- Nu exista garantia securitatii

Forwarder-Receiver

Furnizeaza in mod transparent comunicarea inter-proces pentru interactiunea de tip peer-to-peer.

Pipe-Filter Pattern

Blackboard Pattern

Mai multe subsisteme specializate asambleaza cunostiinte pentru a construi partial sau aproximativ solutia.

Messaging Pattern

Conectarea print-un canal de mesaje care sa permita transferul de date asincron. Acesta suporta loose-coupling.

Publisher-Subscriber Pattern

Infrastructura de propragare a informatiei, unde editorii inregistreaza ce tipuri de evenimente publica si abonatii inregistreaza de ce tipuri de evenimente sunt interesati.

Event-Bus Pattern

Sursele de evenimente (Event sources) publica mesaje pe canale particulare pe un (event bus). Event listeners se aboneaza la anumite canale.

Broker Pattern

Client-Proxy Pattern

Some random stuff

MPI

```
/* Used in: Rank */
#define MPI_PROC_NULL (-1)
#define MPI_ANY_SOURCE (-2)

/* Used in: Tag */
#define MPI_ANY_TAG (-1)
```

Sintaxa:

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm)
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm
comm, MPI_Status *status)
int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,
int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,
int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
int MPI_Bcast( void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
```