Calcul Ştiinţific

Capitolul 1: Arhitecturi de calcul

Bogdan Dumitrescu

Facultatea de Automatică şi Calculatoare

Universitatea Politehnica Bucureşti

Introducere

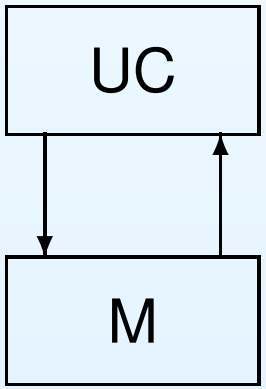
- Calculul ştiinţific este caracterizat de probleme de dimensiuni foarte mari
- Relativ puţine tipuri de probleme (e.g. ecuaţii cu derivate parţiale)
- Se doresc soluţii numerice (numere reale sau complexe !)
- Algebra liniară (sisteme de ecuaţii liniare, calculul valorilor proprii) apare frecvent
- Eficienţa implementării este crucială, în special prin adecvarea algoritmilor la arhitecturile de calcul
- Arhitecturile evoluează rapid: cum alegem/adaptăm algoritmii ?

Cuprins

- Arhitecturi standard şi caracteristicile lor:
  - structură
  - operaţii favorizate
- Cum se poate obţine eficienţa ?
- Algoritmi standard:
  - BLAS
  - LAPACK
  - SCALAPACK

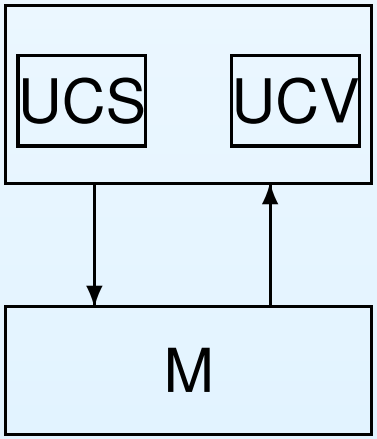
Arhitectura von Neumann

- Pentru fiecare operaţie, operanzii sunt aduşi din memoria M în unitatea centrală UC, rezultatul fiind depus din nou în M
- Instrucţiunile stau şi ele în M (alternativă: arhitectura Harvard, cu memorie separată pentru program)
- Unitatea de măsură standard pentru evaluarea timpului de execuţie: operaţia în virgulă mobilă (flop)
- Model relativ corect până în anii ’80



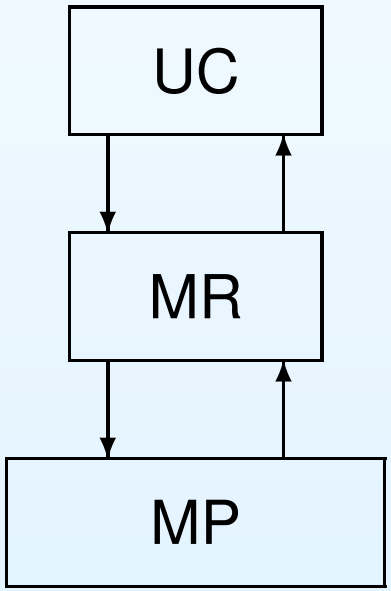
Arhitecturi vectoriale

- UCS—unitate de calcul obişnuită, pentru calcule scalare
- UCV—unitate de calcul vectorial, pe care operaţiile vectoriale de genul $x + y$, $x, y \in \mathbb{R}^n$, se execută rapid
- UCV funcţionează în regim pipeline, folosind şi moduri favorizate de transfer din memorie (e.g. memoria e partajată în blocuri şi există căi separate de alimentare a UCV pentru fiecare bloc)
- Operaţii eficiente: cu vectori de lungime suficient de mare



Arhitecturi cu memorie ierarhică

- Mai multe niveluri de memorie (cel puţin două)
- MR—memorie rapidă (cache), care poate alimenta UC cu date astfel încât să se efectueze un flop pe tact
- MP—memoria principală, în general mult mai lentă
- Operaţii eficiente: cele care refolosesc datele din MR



Memorie ierarhică—evoluţie

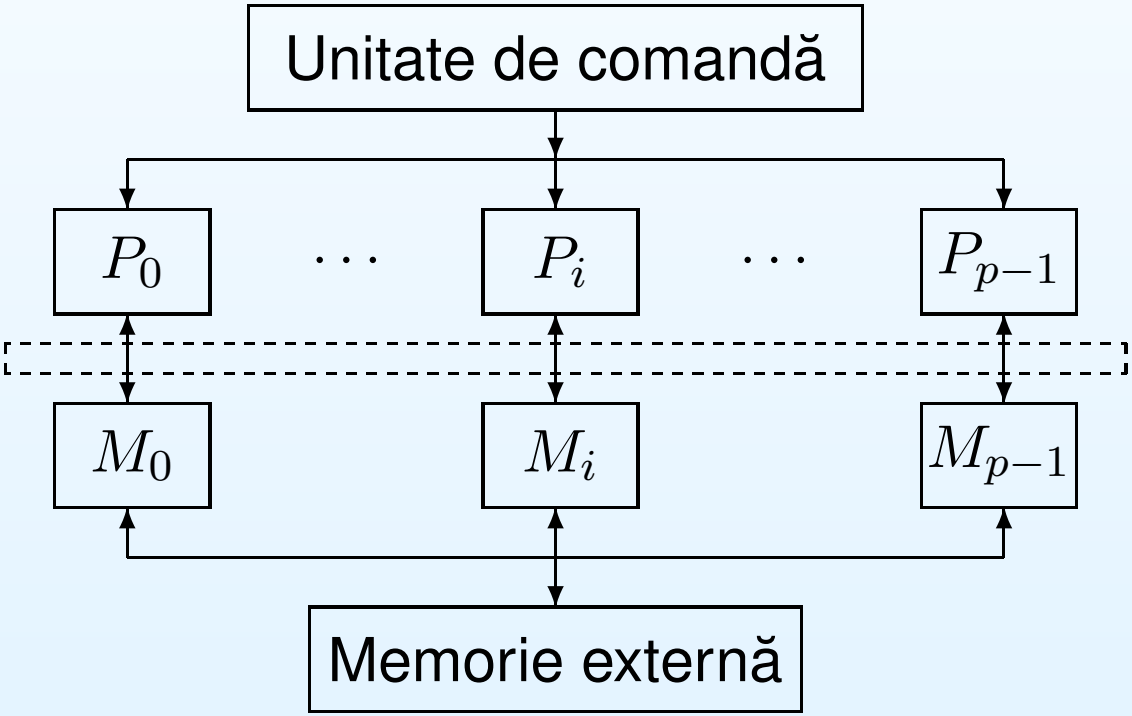
- Procesoare RISC (set redus de instrucţiuni), astfel încât decodificarea să fie rapidă
- Predicţie a datelor (pe mai multe căi) şi instrucţiunilor folosite, în încercarea de a aduce datele în MR înainte de a fi efectiv necesare
- Transferuri de blocuri de date din MP în MR
- Multicore—mai multe procesoare/UC, alimentate simultan (paralelism)
- Cheia eficienţei rămâne refolosirea datelor (sau refolosirea imediată a rezultatelor), ceea ce necesită reorganizarea algoritmilor (operaţii la nivel de bloc)

Calculatoare paralele

- Tipuri principale: SIMD, MIMD
- SIMD (Single Instruction Multiple Data): aceeaşi instrucţiune se execută simultan asupra mai multor date
- MIMD (Multiple Instruction Multiple Data): mai multe fluxuri de instrucţiuni se execută asupra mai multor date (fiecare flux cu datele lui)
- Tipuri de MIMD
  - cu memorie comună
  - cu memorie distribuită

Arhitectura SIMD (1)

- Procesoarele P_i execută aceeaşi instrucţiune, dictată de unitatea de comandă
- Fiecare procesor îşi ia datele dintr-o memorie locală M_i

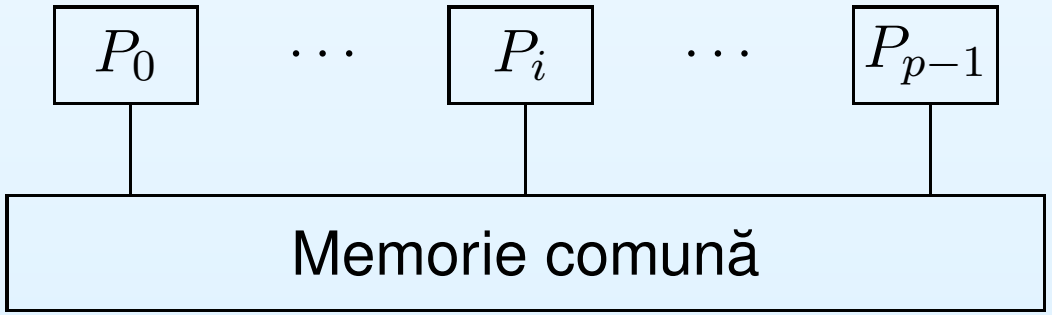


Arhitectura SIMD (2)

- Memoriile M_i sunt de obicei rapide şi comunică cu o memorie principală (eventual externă)
- Între procesoarele P_i şi memoriile M_i există o reţea de interconectare, care poate configura accesul (de exemplu: procesorul P_i ia datele din memoria M_{i+1})
- Procesoarele sunt de obicei foarte simple
- Operaţii favorizate: vectoriale, cu lungimea vectorului multiplu al numărului de procesoare
- Adunarea a doi vectori de lungime p se face într-un tact
- Problema critică: alimentarea cu date a $M_i \Rightarrow$ complicaţii arhitecturale

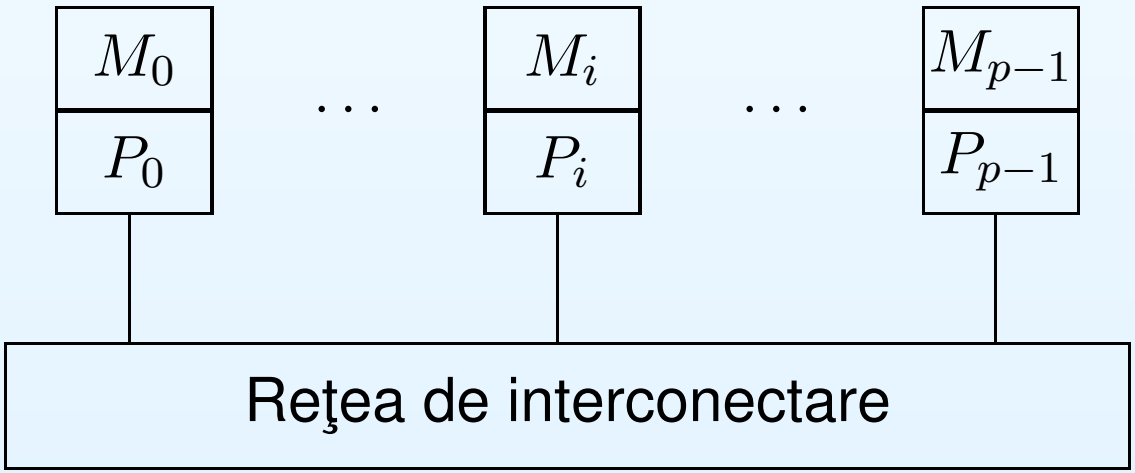
MIMD cu memorie comună

- Fiecare procesor execută instrucţiuni proprii
- Datele se află în memoria comună
- De obicei fiecare procesor are o memorie rapidă mică proprie (nereprezentată)
- Operaţii favorizate: paralele, la nivel de bloc
- Avantaj: comunicaţie simplă, prin intermediul memoriei
- Probleme: cu cât creşte numărul de procesoare, cu atât creşte probabilitatea conflictele de acces la memorie $\Rightarrow$ scade viteza de calcul



MIMD cu memorie distribuită (1)

- Fiecare procesor are memorie proprie (arhitectura locală cu RISC şi memorie ierarhică, de obicei)
- Comunicaţia se face printr-o reţea de comunicaţie, prin mesaje explicite
- Operaţii favorizate: paralele, la nivel de bloc
- Comunicaţia prin mesaje necesită algoritmi dedicaţi



MIMD cu memorie distribuită (2)

- Topologii ale reţelei de comunicaţie
  - fixă: inel, grilă (tor), hipercub, fat tree
  - configurabilă: switch, magistrale
- În calculatoarele actuale, topologia este de obicei transparentă pentru utilizator
- Biblioteci de funcţii de comunicaţie
- Numărul de procesoare poate fi foarte mare, de ordinul miilor şi peste (sute de mii)
- Putere de calcul maximă: $>$ 1 petaflop/secundă
- Exerciţiu: citiţi www.top500.org

Cum obţinem programe eficiente ?

- Scopul obişnuit în rezolvarea unei probleme de calcul ştiinţific este scrierea unui program cu timp de execuţie cât mai mic
- (Avem în vedere probleme care se rezolvă de multe ori, cu date diferite de fiecare dată)
- Totuşi, efortul de programare trebuie să fie rezonabil
- Dificultăţi
  - un cod eficient pe un calculator poate fi foarte lent pe altul
  - arhitecturile sunt din ce în ce mai complexe, deci greu de modelat
  - arhitecturile evoluează repede
- Soluţii: combinaţii ale celor expuse în paginile următoare

Soluţia 1: compilatorul perfect

- Idealul programatorului:
  - se ia un algoritm optim dintr-un punct de vedere general (e.g. număr minim de operaţii)
  - se implementează într-un limbaj de nivel înalt
  - prin compilare se obţine un cod (aproape) optim
- Compilatorul este optimizat pentru un anume calculator
- Deşi s-au facut progrese, dificultăţi majore:
  - descrierea precisă a arhitecturii
  - "înţelegerea" completă a programului de către compilator
  - timpul de compilare limitat (se pot obţine coduri aproape optime pentru unele programe scurte, dar complexitatea compilării explodează pentru coduri mai lungi)
  - timpul de optimizare a unui compilator ar putea depăşi durata de viaţă a calculatorului

Soluţia 2: nuclee optimizate

- Se identifică operaţiile "elementare" esenţiale pentru un domeniu
- De exemplu, în algebra liniară, acestea sunt înmulţirea de matrice, rezolvarea de sisteme triunghiulare, plus altele similare cu complexitate mai mică (produs matrice-vector, produs scalar, etc.)
- Pentru fiecare calculator, se scriu biblioteci optimizate pentru aceste operaţii, e.g. BLAS
- Programele sunt scrise cu cât mai multe apeluri la biblioteci, astfel încât sa fie portabile (după recompilare)
- Soluţia cea mai răspândită acum, deşi re-scrierea unei biblioteci optimale necesită multe luni de muncă pentru specialişti super-calificaţi
- Schimbări aparent mici în arhitectură pot necesita reoptimizarea bibliotecii

Soluţia 3: adaptare automată

- Se păstrează ideea de bibliotecă cu funcţii elementare
- Optimizarea bibliotecii pentru o nouă arhitectură se face automat, empiric
- Optimizarea se face prin intermediul unor parametri (e.g. dimensiuni de blocuri) într-unul sau mai multe coduri fixe sau prin generare de cod
- Pe baza unei proceduri de căutare, se măsoară timpii de execuţie pentru diverse valori ale parametrilor
- Avantaj: optimizarea se face exact pe calculatorul ţintă
- Dezavantaj: lipsă de robusteţe la variaţii mari ale arhitecturii
- Exemplu: ATLAS (Automatically Tuned Linear Algebra Software)

Structura BLAS

- BLAS—Basic Linear Algebra Subroutines (1973–1989)
- Rutinele sunt organizate pe trei nivele
- Nivel 1: operaţiilor vectoriale (sumă de vectori, produs scalar) care necesită $O(n)$ flopi. BLAS-1 este adecvat în special calculatoarelor vectoriale.
- Nivel 2: dedicat operaţiilor matrice-vector (produs, rezolvare de sisteme triunghiulare), care necesită $O(n^2)$ flops. BLAS-2 este eficient tot pe calculatoare vectoriale.
- Nivel 3: operaţii matrice-matrice, ca înmulţirea de matrice sau rezolvarea de sisteme triunghiulare cu parte dreaptă multiplă, care necesită $O(n^3)$ flops. BLAS-3 este eficient îndeosebi pe calculatoare cu memorie ierarhică.

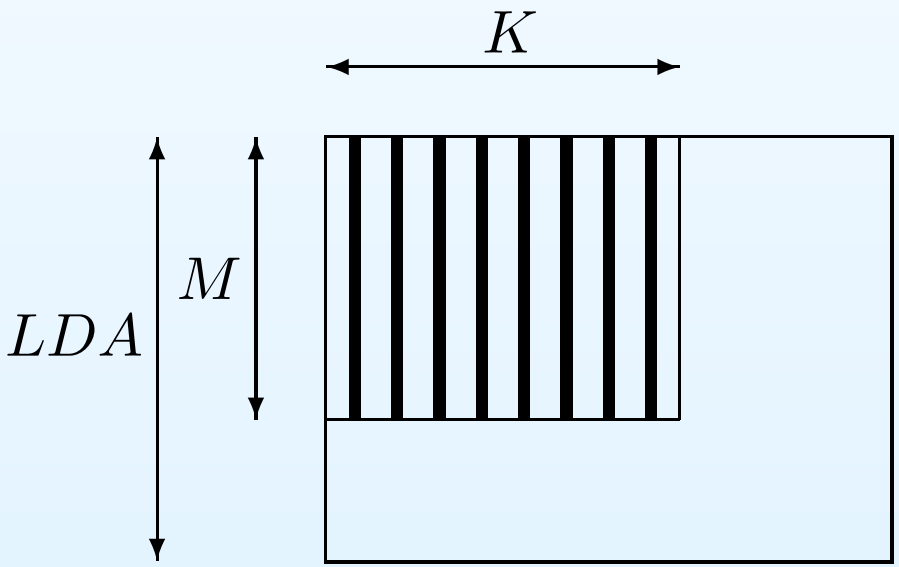
BLAS-1

Nume	Operaţie	Prefixe
xSWAP	$x \leftrightarrow y$	S, D, C, Z
xSCAL	$x \leftarrow \alpha x$	S, D, C, Z
xCOPY	$x \leftarrow y$	S, D, C, Z
xAXPY	$y \leftarrow \alpha x + y$	S, D, C, Z
xDOT	$dot \leftarrow x^T y$	S, D
xDOTU	$dot \leftarrow x^T y$	C, Z
xDOTC	$dot \leftarrow x^H y$	C, Z
xNRM2	$nrm2 \leftarrow \|x\|_2$	S, D, C, Z

S – real simplă precizie	C – complex simplă precizie
D – real dublă precizie	Z – complex dublă precizie

Memorarea unei matrice

- BLAS standardizează nu doar operaţiile aritmetice, ci şi modul de memorare
- (Sub)matrice $M \times K$, memorată pe coloane, într-un tablou al cărui număr de linii este LDA (dimensiunea principală)
- (Numărul de coloane al tabloului este nesemnificativ)



Exemplu (1)

- Valorile întregi reprezintă ordinea în care sunt considerate elementele submatricei
- Asterisc: valoare din tablou care nu face parte din submatrice
- $M = 3$, $K = 4$, $LDA = 5$

$$\begin{bmatrix} 1 & 4 & 7 & 10 & * & * & * \\ 2 & 5 & 8 & 11 & * & * & * \\ 3 & 6 & 9 & 12 & * & * & * \\ * & * & * & * & * & * & * \\ * & * & * & * & * & * & * \end{bmatrix}$$

Exemplu (2)

- Modul de descriere este identic în cazul în care submatricea este în interior
- Se modifică doar adresa de început a submatricei (adresa elementului din stânga sus)
- $M = 3$, $K = 4$, $LDA = 6$

$$\begin{bmatrix} * & * & * & * & * & * & * \\ * & * & * & * & * & * & * \\ * & 1 & 4 & 7 & 10 & * & * \\ * & 2 & 5 & 8 & 11 & * & * \\ * & 3 & 6 & 9 & 12 & * & * \\ * & * & * & * & * & * & * \end{bmatrix}$$

Exemplu de apel BLAS-1

- Operaţia $y \leftarrow \alpha x + y$ se apelează prin

  SAXPY(N, $ALFA$, X, $INCX$, Y, $INCY$)

- $INCX$ reprezintă distanţa, în memorie, între două elemente succesive ale vectorului al cărui prim element se găseşte la adresa X
- Dacă vectorul este coloană a unei matrice, $INCX = 1$
- Dacă vectorul este linie, $INCX = LDA$
- Dacă $LDA = N$, atunci mai multe coloane succesive sunt concatenate în memorie într-un singur vector $vec(A)$
- În cazul în care vectorul este format din mai multe "bucăţi", el trebuie copiat într-o zonă de memorie cu descrierea de mai sus (sau operaţia se face pe bucăţi)

Convenţii de nume

- Primul caracter al unui nume de rutină reprezintă tipul datelor
- Pentru BLAS-2 şi -3, tipul matricelor este reprezentat cu două caractere (pe poziţiile 2 şi 3 din nume)
- Restul caracterelor reprezintă numele operaţiei

GE - generală	GB - generală bandă	
SY - simetrică	SB - simetrică bandă	SP - simetrică împachetat
HE - hermitică	HB - hermitică bandă	HP - hermitică împachetat
TR - triunghiulară	TB - triungh. bandă	TP - triungh. împachetat

BLAS-2

- BLAS-2 conţine rutine pentru trei operaţii
- xyyMV: produs matrice-vector de forma $y \leftarrow \alpha Ax + \beta y$ (toate combinaţiile yy din tabel sunt permise)
- xyySV: rezolvarea sistemelor (inferior sau superior) triunghiulare, iar yy poate fi TR, TB sau TP
- xGER: produsul exterior (actualizare de rang 1) $A \leftarrow \alpha x y^T + A$
- Există şi rutine pentru actualizarea de rang 2, vezi explicaţia operaţiei la BLAS-3

BLAS-3

- Două operaţii de bază
  - înmulţirea de matrice, în câteva variante
  - rezolvarea de sisteme triunghiulare cu parte dreaptă multiplă
- BLAS-3 a apărut mult după BLAS-2, o dată cu calculatoarele cu memorie ierarhică
- Acoperă practic BLAS-2
- În paginile următoare, A, B, C sînt matrice oarecare, cu dimensiuni oarecare, dar adecvate operaţiilor, sau simetrice şi pătrate, T este o matrice triunghiulară, superior sau inferior, iar α şi β sînt scalari

GEMM

- xGEMM (GEneral Matrix Multiplication): înmulţirea matrice-matrice, în cazul general

  xGEMM($TRANSA$, $TRANSB$, M, N, K, $ALFA$, A, LDA, B, LDB, $BETA$, C, LDC)

- C este întotdeauna de dimensiune $M \times N$
- Transpunerea nu este lăsată utilizatorului, deoarece poate fi mare consumatoare de timp dacă se execută explicit
- Operaţiile efectuate de rutină sunt

	$TRANSA$ = 'N'	$TRANSA$ = 'T'
$TRANSB$ = 'N'	$C \longleftarrow \alpha AB + \beta C$	$C \longleftarrow \alpha A^T B + \beta C$
	A e $M \times K$, B e $K \times N$	A e $K \times M$, B e $K \times N$
$TRANSB$ = 'T'	$C \longleftarrow \alpha AB^T + \beta C$	$C \longleftarrow \alpha A^T B^T + \beta C$
	A e $M \times K$, B e $N \times K$	A e $K \times M$, B e $N \times K$

Rutine cu matrice simetrice

- xSYMM (SYmetric Matrix Multiplication): înmulţire matrice-matrice, cu una din matrice simetrică

$$C \leftarrow \alpha AB + \beta C$$

- xSYRK (SYmmetric Rank-K update): actualizare de rang k a unei matrice simetrice; A are dimensiune $n \times k$

$$C \leftarrow \alpha AA^T + \beta C$$

- xSYR2K: actualizare de rang $2k$ a unei matrice simetrice

$$C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$$

Rutine cu matrice triunghiulare

- xTRMM (TRiangular Matrix Multiplication): înmulţire matrice-matrice, cu una dintre matrice triunghiulară:

$$B \leftarrow \alpha T B$$

- Există şi varianta în care T este la dreapta; T poate fi inferior sau superior triunghiulară
- xTRSM (TRiangular system Solver, with Multiple right hand term): calculează soluţia unui sistem liniar triunghiular, cu parte dreaptă multiplă ($TX = B$):

$$X \leftarrow \alpha T^{-1} B$$

- Există versiuni în care necunoscuta este în stânga (de genul $XT = B$); T este fie superior, fie inferior triunghiulară

De ce BLAS-3 ?

- De ce se obţine eficienţă cu BLAS-3 pe calculatoare cu memorie ierarhică ?
- Presupunem că vrem să calculăm $C = AB$, cu matrice $n \times n$
- Dacă $3n^2$ este mai mic decât dimensiunea memoriei rapide, atunci se execută $2n^3$ flopi (necesari pentru MM) şi doar $3n^2$ accese la memoria principală lentă (pentru a aduce matricele în MR)
- Deci fiecare dată este adusă o singură dată în MR, iar acolo este (re)utilizată de n ori
- Dacă matricele sunt mari ($3n^2 > \dim$ MR), atunci operaţiile se desfăşoară pe blocuri de dimensiune potrivită (vezi algoritmii din capitolul următor)
- Concluzie: rutinele BLAS sunt rapide pe matrice suficient de mari

Evaluarea eficienţei pe arh. cu mem. ierarhică

- Pentru a evalua cât de rapid este un algoritm pe un calculator cu memorie ierarhică, introducem *ponderea operaţiilor de nivel* 3 prin raportul

$$P_3(n) = \frac{N_3(n)}{N_{total}(n)}$$

- $N_{total}(n)$ reprezintă numărul total de flopi necesari execuţiei algoritmului
- $N_3(n)$ este numărul de flopi executaţi în rutinele din BLAS-3
- Datorită eficienţei operaţiilor la nivel de bloc (optimizate în rutinele BLAS), un algoritm este cu atât mai bun cu cât $P_3(n)$ este mai apropiată de 1

LAPACK

- LAPACK (Linear Algebra PACKage) este o bibliotecă de rutine ce rezolvă probleme standard de algebră liniară
- Algoritmii implementaţi sunt structuraţi astfel încât să apeleze cât mai mult rutine BLAS, deci ating maximul de performanţă pe calculatoare cu memorie ierarhică
- Sursele LAPACK sunt disponibile gratuit: dacă există o implementare BLAS eficientă, se poate spera obţinerea de performanţe bune
- LAPACK e urmaşa LINPACK şi EISPACK, două biblioteci cu algoritmi cu bune calităţi numerice, optimizaţi după numărul de operaţii

Probleme rezolvate

- Rezolvare de sisteme liniare determinate $AX = B$, cu A pătrată
- Rezolvare de sisteme liniare în sens CMMP (matricea A este dreptunghiulară)
- Probleme CMMP generalizate, de exemplu $\min \|c - Ax\|_2$, cu restricţiile $Bx = d$
- Calculul valorilor şi vectorilor proprii
- Calculul valorilor şi vectorilor singulari
- Valori proprii sau singulare generalizate

Tipuri de rutine

- Numele rutinelor LAPACK au forma xyyzzz, unde x codifică formatul de reprezentare a datelor, yy reprezintă tipul matricei (ca la BLAS, plus altele), iar zzz reprezintă operaţia
- Categorii de rutine
  - rutine *driver*, care rezolvă o problemă completă, de exemplu aflarea soluţiei unui sistem liniar
  - rutine *de calcul*, care rezolvă subprobleme sau completează rezolvarea unei probleme, de exemplu calculul factorizării LU sau rafinarea iterativă a soluţiei unui sistem liniar
  - rutine *auxiliare*

Rutine driver

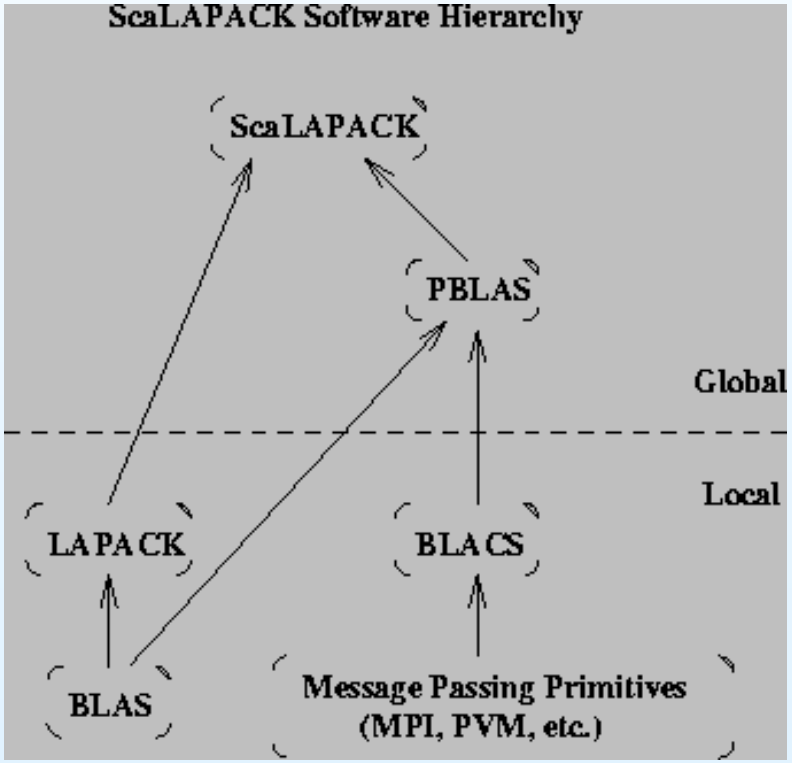
- Driverul *simplu* rezolvă problema cu datele utilizatorului, furnizând (fără "comentarii") rezultatul numeric
- Exemplu: xyySV rezolvă sistemele cu parte dreaptă multiplă $AX = B$ sau $A^T X = B$. xGESV se utilizează pentru matrice A oarecare, xPOSV se utilizează când matricea A este simetrică pozitiv definită, etc.
- Driverul *expert* încearcă să modifice datele astfel încât soluţia numerică să fie mai precisă, evaluează amplitudinea erorilor posibile, etc.
- Exemplu: xyySVX rezolvă sistemul, dar şi
  - scalează matricea A dacă este necesar
  - estimează numărul de condiţionare al matricei A
  - rafinează iterativ soluţia

ScaLAPACK

- ScaLAPACK (Sca de la scalable) este o bibliotecă paralelă ce rezolvă o submulţime (semnificativă) din problemele rezolvate de LAPACK
- Apelează rutine LAPACK şi (deci) BLAS
- Este destinată în special calculatoarelor MIMD cu memorie distribuită
- Comunicaţia efectuează prin mesaje, utilizând biblioteci standard
  - MPI—Message Passing Interface
  - PVM—Parallel Virtual Machine

Structură software

- Module specifice noi:
  - BLACS—Basic Linear Algebra Communication Subroutines
  - PBLAS—Parallel BLAS



Repartizarea matricelor

- ScaLAPACK presupune că calculatorul ţintă are o topologie (reală sau virtuală) de grilă
- Matricele (date sau rezultate) sunt distribuite în memoriile locale ale procesoarelor în mod *bloc ciclic*
- Un procesor are aproximativ n^2/p elemente matrice
- Considerăm o matrice 5×5, partiţionată în blocuri 2×2

$$\left[\begin{array}{cc|ccc} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ \hline a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ \hline a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{array}\right] \equiv \left[\begin{array}{ccc} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{array}\right]$$

Repartizare bloc ciclică—exemplu

- Blocurile sunt repartizate procesoarelor, unul câte unul, începând din stânga sus şi urmând structura grilei, cu repetiţie
- O matrice bloc 3×3 pe 2×2 procesoare $\begin{bmatrix} P_{00} & P_{01} & P_{00} \\ P_{10} & P_{11} & P_{10} \\ P_{00} & P_{01} & P_{00} \end{bmatrix}$
- Matricea 5×5 este deci repartizată astfel

$$\left[\begin{array}{ccc|cc} a_{11} & a_{12} & a_{15} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{25} & a_{23} & a_{24} \\ a_{51} & a_{52} & a_{55} & a_{53} & a_{54} \\ \hline a_{31} & a_{32} & a_{35} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{45} & a_{43} & a_{44} \end{array}\right] \equiv \left[\begin{array}{cc|c} A_{11} & A_{13} & A_{12} \\ A_{31} & A_{33} & A_{32} \\ \hline A_{21} & A_{23} & A_{22} \end{array}\right]$$

Descriptor de matrice

- M_A, N_A—dimensiunea globală a matricei
- MB_A, NB_A—dimensiunea unui bloc
- $RSRC\_A$, $CSRC\_A$—linia, respectiv coloana, pe care se află procesorul care deţine primul bloc al matricei (cel din stânga sus)
- LLD_A—dimensiunea principală a tabloului local în care e memorată matricea locală
- Altele: tipul matricei, contextul (este distribuită pe toate procesoarele sau pe o submulţime)

Descrierea unei submatrice

- O submatrice (distribuită) a unei matrice este descrisă de următoarele informaţii
- M, N—dimesniunea (globală) a submatricei
- A—adresa tabloului local care conţine matricea
- IA, JA—indicii de început ai submatricei în matricea globală
- $DESCA$—descriptor al matricei globale

- Manipularea datelor este principala dificultate a utilizatorului de ScaLAPACK
- Apelul rutinelor este în rest similar cu LAPACK