Procesare Paralelă

Capitolul 1: Arhitecturi paralele

Bogdan Dumitrescu

Facultatea de Automatică şi Calculatoare

Universitatea Politehnica Bucureşti

Introducere

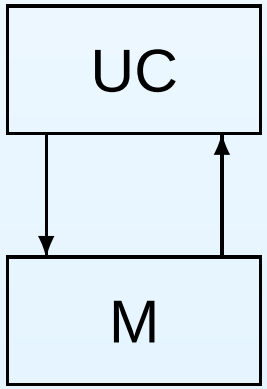
- Probleme de dimensiuni mari apar în multe domenii, atât ştiinţifice cât şi tehnice
- Relativ puţine tipuri de probleme; tipic: ecuaţii cu derivate parţiale
- Algebra liniară (sisteme de ecuaţii liniare, calculul valorilor proprii) apare frecvent
- Puterea totală de calcul disponibilă e uriaşă, dar fiecare procesor e "relativ" lent; calculul paralel e soluţia naturală
- Eficienţa implementării este crucială, în special prin adecvarea algoritmilor la arhitecturile de calcul
- Arhitecturile evoluează rapid: cum adaptăm algoritmii ?

Cuprins

- Arhitecturi standard şi caracteristicile lor
  - structură
  - operaţii favorizate
- Arhitecturi paralele
  - SIMD
  - MIMD (memorie comună, memorie distribuită)
  - Reţele de comunicaţie—topologii
- Cum adaptăm algoritmii la arhitecturi—strategii

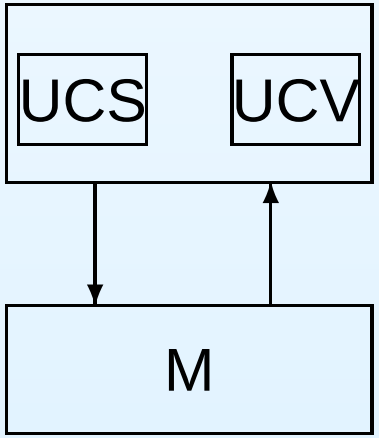
Arhitectura von Neumann

- 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



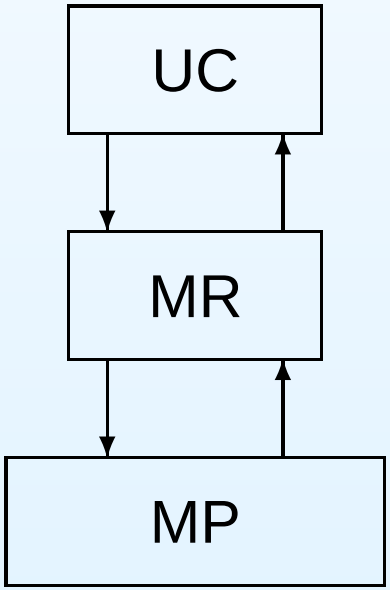
Arhitecturi vectoriale

- 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

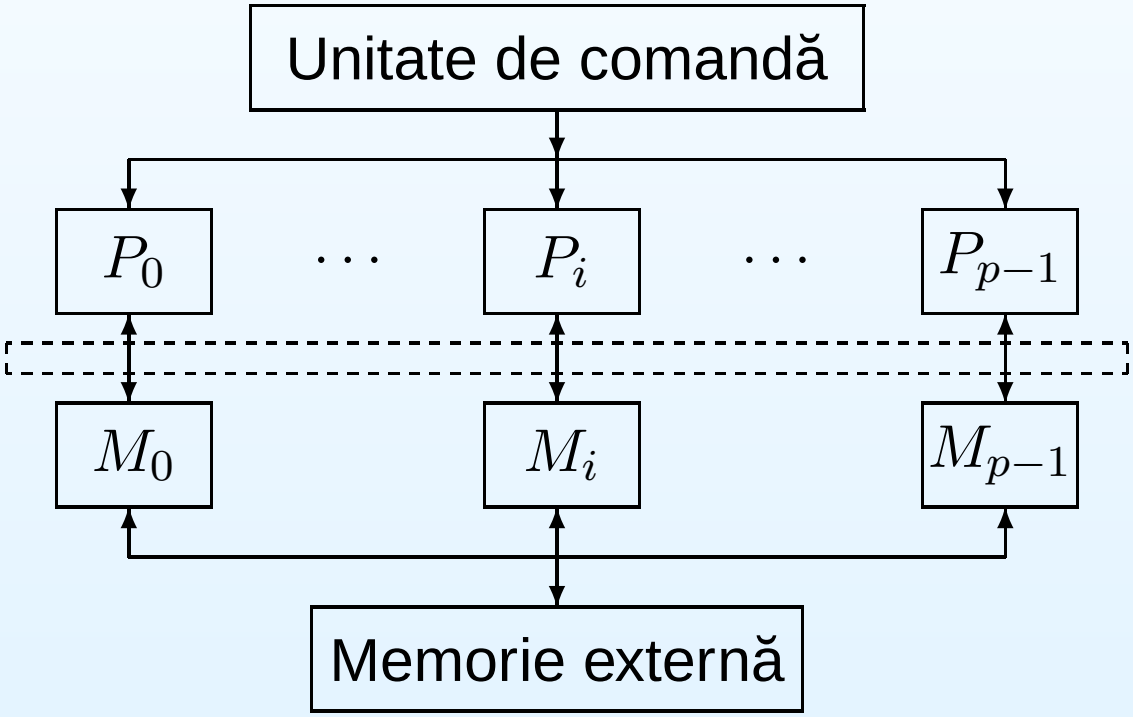
- Procesoare RISC (set redus de instrucţiuni), astfel încât decodificarea să fie rapidă
- Predicţie a datelor (pe mai multe căi) şi instrucţiunilor folosite, în încercarea de a aduce datele în MR înainte de a fi efectiv necesare
- Transferuri de blocuri de date din MP în MR
- Multicore—mai multe procesoare/UC, 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

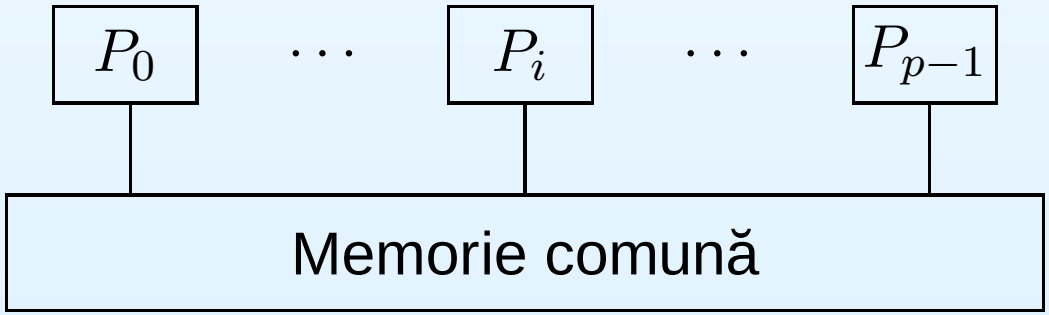


Arhitectura SIMD (2)

- 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

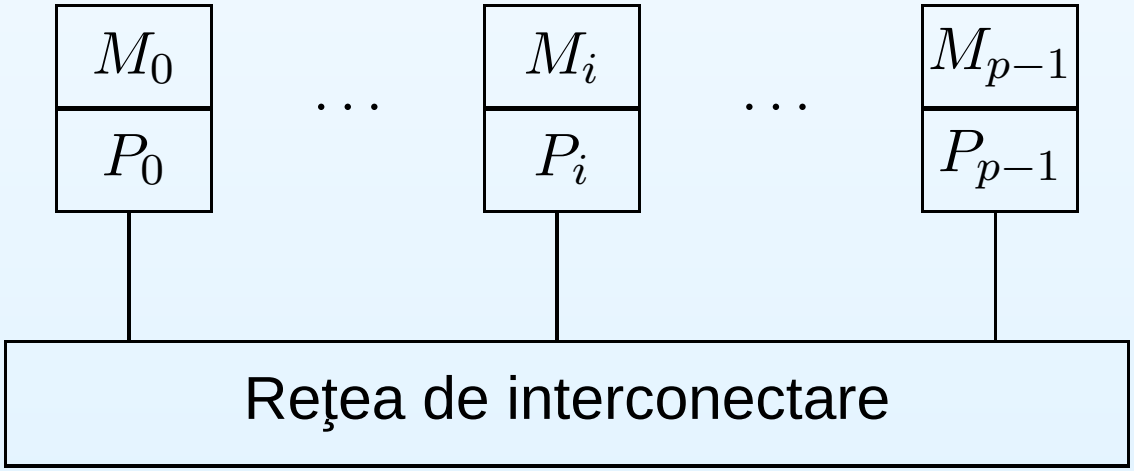
MIMD cu memorie comună

- 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: $>$ 10 petaflop/secundă
- Exerciţiu: citiţi www.top500.org

Grafuri

- O reţea de comunicaţie se modelează natural printr-un graf neorientat
- Procesoarele sunt nodurile, iar liniile de comunicaţie arcele
- Un *graf* neorientat este $G = (V, E)$, unde V e mulţimea nodurilor, iar $E \subset V \times V$ cea a arcelor
- Numărul de noduri ale unui graf: *ordin* (p, numărul de procesoare)
- Graf *simplu*: nu are bucle

Grafuri

- Două noduri sunt *adiacente* (vecine) dacă există un arc între ele
- *Gradul* unui nod: numărul de vecini ai nodului, adică numărul de arce plecând din nod
- *Gradul* unui graf (Δ): maximul gradelor nodurilor componente
- Graf *regulat*: gradele tuturor nodurilor sunt egale
- Numărul de arce al unui graf regulat este $p\Delta/2$

Grafuri

- *Drum* (sau *cale*) între două noduri $x, y \in V$: o secvenţă de noduri x_0, x_1, . . . , x_k, astfel încât două noduri consecutive sunt adiacente şi $x_0 = x$, $x_k = y$.
- Drum *elementar*: nodurile componente x_1, . . . , x_{k-1} sunt distincte
- Într-o arhitectură cu memorie distribuită, un drum este o cale de comunicaţie între două procesoare
- *Lungimea* unui drum: numărul de arce ale sale, k în notaţia de mai sus
- *Distanţa* dintre două noduri, $dist(x, y)$, este lungimea celui mai scurt drum între acestea
- *Diametrul* grafului (D): maximul tuturor distanţelor între perechi de noduri ale grafului, adică distanţa dintre cele mai depărtate noduri.

Grafuri

- *Ciclu*: un drum elementar pentru care nodul iniţial şi cel final sunt identice
- *Ciclu hamiltonian*: un ciclu trecând (exact o dată) prin fiecare nod al grafului; lungimea sa este egală cu ordinul grafului
- Graf *conex*: există (cel puţin) un drum între oricare două noduri ale sale
- *Arbore*: graf conex fără cicluri

Deziderate topologice: hardware

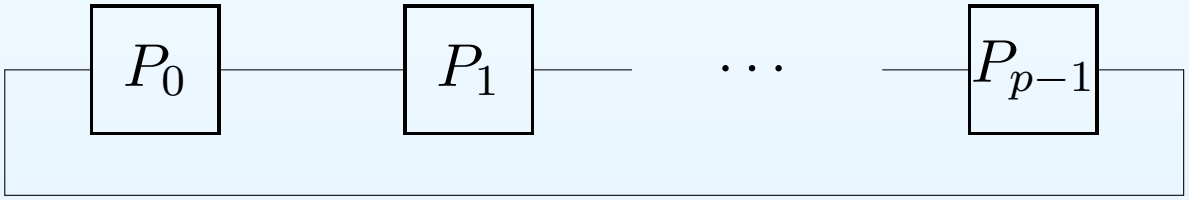
- Graf conex
- Grad mic şi graf regulat: procesoare identice cu doar câteva canale de comunicaţie, deci uşor de realizat
- Număr total de arce relativ mic, fiecare arc însemnând o conexiune fizică realizată printr-un traseu electric pe o placă şi, eventual, între plăci.

Deziderate topologice: software

- Diametru mic, astfel încât distanţele între procesoare să fie limitate, iar comunicarea cât mai facilă
- Cele două tipuri de deziderate sunt antagonice: cu cât gradul e mai mic, cu atât sunt mai puţine arce, deci scad şansele să existe drumuri scurte între oricare două procesoare.
- Extremă: graful *total interconectat* (sau *complet*), în care există arce între oricare două noduri; $D = 1$, $\Delta = p - 1$, numărul de arce $p(p-1)/2$
- La cealaltă extremă se află inelul

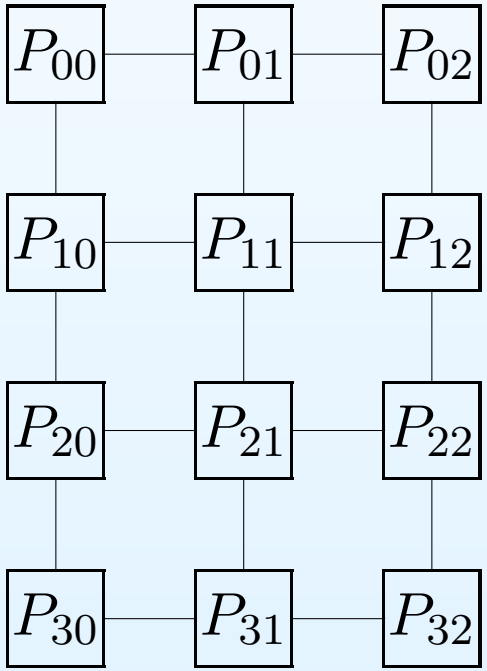
Inelul

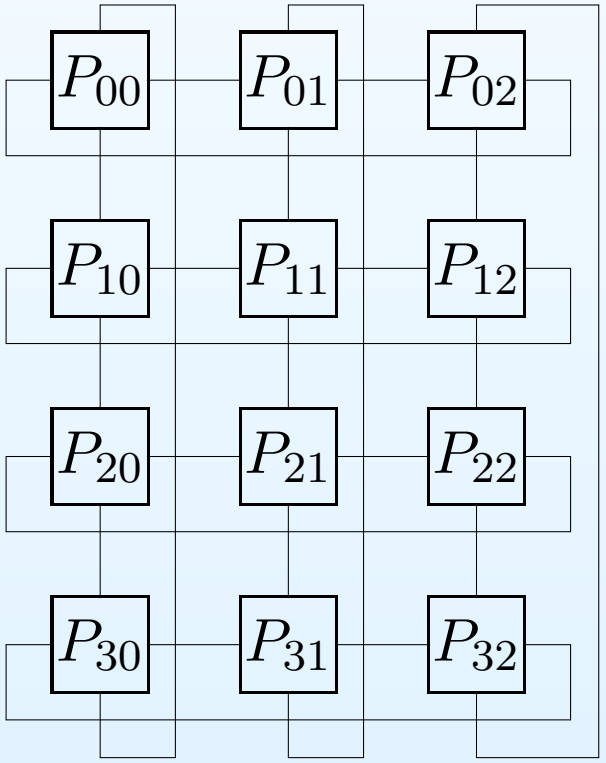
- Procesorul P_i este legat de P_j, unde
  $j = (i-1) \bmod p$ (P_j este vecinul din stânga)
  sau $j = (i+1) \bmod p$ (P_j este vecinul din dreapta)
- Grad: $\Delta = 2$
- Numărul de arce: p
- Diametrul: $D = \lfloor p/2 \rfloor$



Grila şi torul

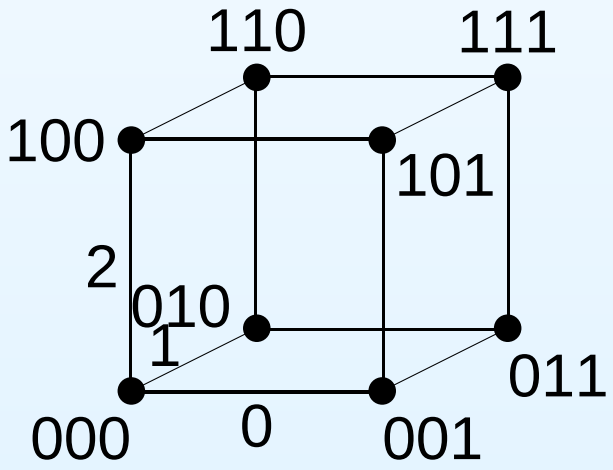
- Pentru un tor pătrat $\sqrt{p} \times \sqrt{p}$
- Grad: $\Delta = 4$
- Numărul de arce: $2p$
- Diametrul: $D = 2\lfloor p/2 \rfloor$

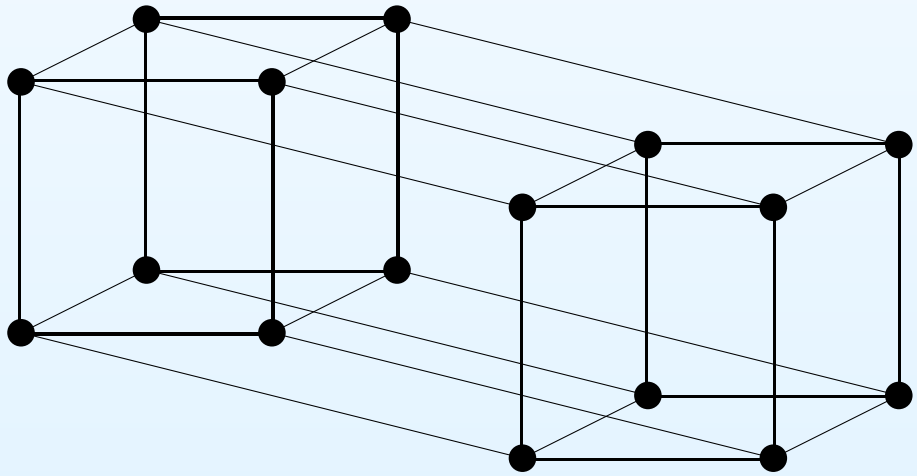




Hipercubul

- Arhitectură foarte avantajoasă pentru unii algoritmi
- Notăm $\mathcal{H}_d$ hipercubul de dimensiune d, care are $p = 2^d$ procesoare
- Fiecare procesor este legat de d vecini; P_i este conectat de P_j dacă reprezentările binare ale numerelor i şi j diferă printr-un singur bit





Hipercubul

- Grad: $\Delta = d = \log p$
- Numărul de arce: $2pd = 2p \log p$
- Diametrul: $D = p$
- Distanţa dintre două noduri este distanţa Hamming dintre adresele lor
- Definiţie recursivă: $\mathcal{H}_d$ poate fi obţinut luând două copii ale lui $\mathcal{H}_{d-1}$ şi adăugându-le arcele care unesc nodurile aflate în aceeaşi poziţie în cele două copii; la adresele nodurilor se va adăuga un bit în poziţia cea mai semnificativă: 0 pentru prima copie a lui $\mathcal{H}_{d-1}$, 1 pentru a doua

Recapitulare

Topologie	Δ	D	N
Inel	2	$O(p)$	$O(p)$
Grilă	4	$O(\sqrt{p})$	$O(p)$
Tor	4	$O(\sqrt{p})$	$O(p)$
Tor 3D	6	$O(\sqrt[3]{p})$	$O(p)$
Hipercub	$\log p$	$O(\log p)$	$O(p \log p)$

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)