Paralelizmus a Lokalita

Väzby na architektúru počítačov

(podľa cs252 a cs267 at Berkeley
a fialového draka)

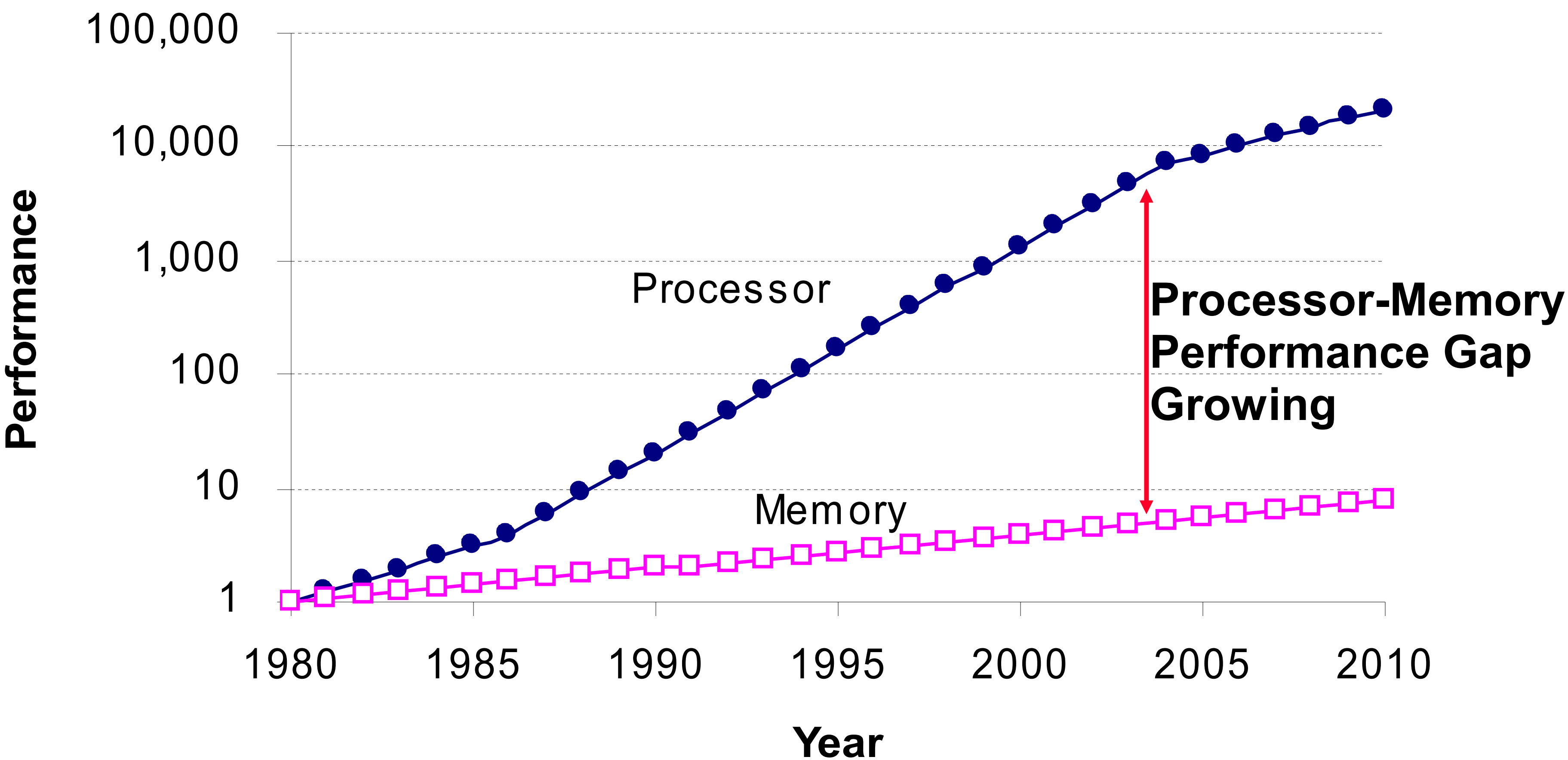
Ján Šturc

Please silence your phones and close your laptops!

OBSAH

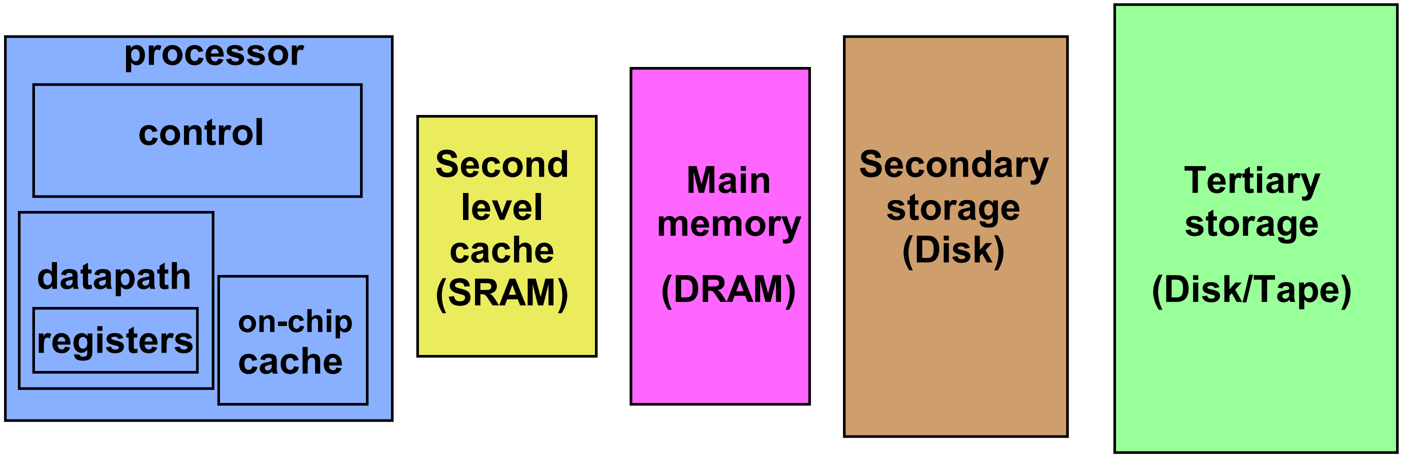
- Hierarchia pamätí
- Paralelizmus v rámci jedného procesoru
- SIMD
- Cache, optimalizácie využitia
- Modely paralelizmu
- Afinné transformácie
- Násobenie matíc – vnorené cykly
- Nástroje
  - Fourier-Motzkin elimination
  - GCD test
  - Integer linear programming
  - Farkas' lemma

Prečo hierarchia pamätí?



Hierarchia a parametre pamätí

- Most programs have a high degree of locality in their accesses
  - **spatial locality:** accessing things nearby previous accesses
  - **temporal locality:** reusing an item that was previously accessed
- Memory hierarchy tries to exploit locality



Speed	1ns	10ns	100ns	10ms	10sec
Size	10 KB	MB	GB	100 GB	TB a viac

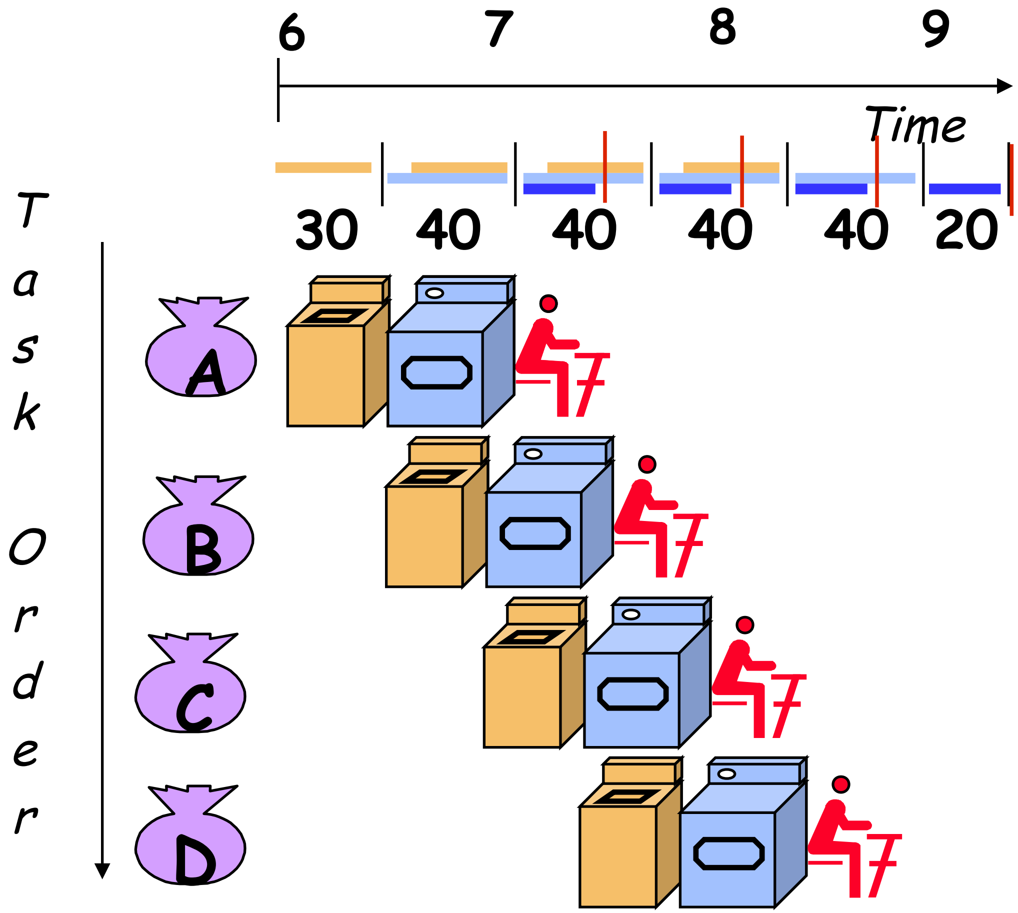
Paralelizmus v rámci jedného procesoru

- Hidden from software (sort of)
- Pipelining
- SIMD units

Prúdové spracovanie – pipelining

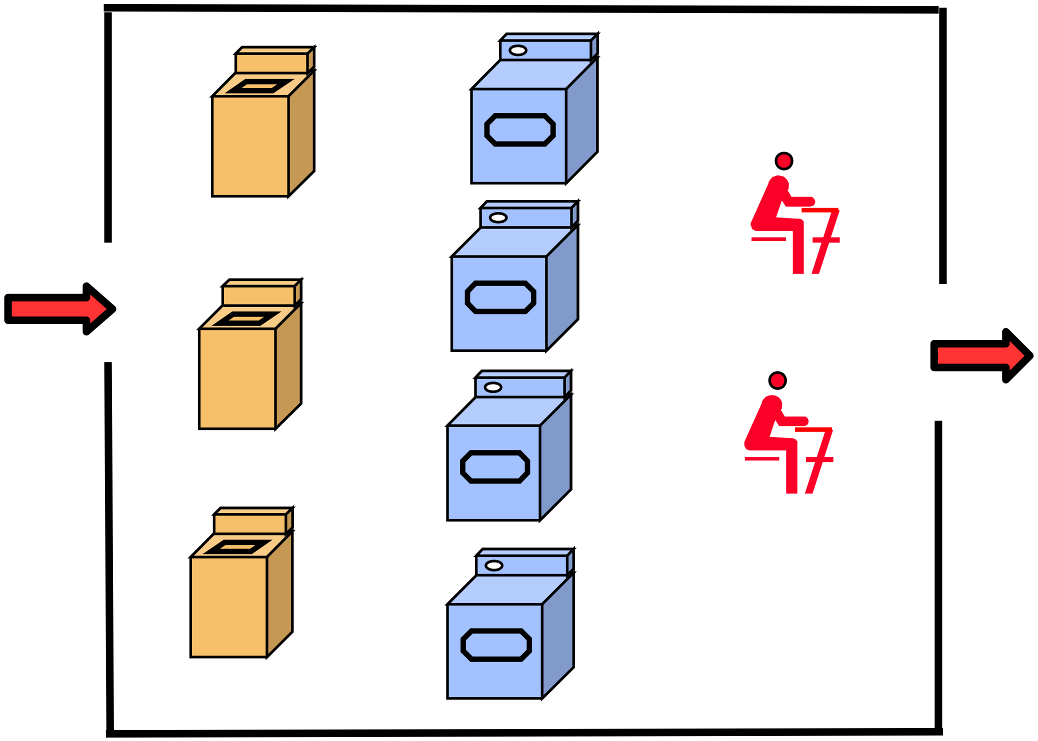
Dave Patterson's Laundry example: 4 people doing laundry (americká práčovňa)

Latency: wash (30 min) + dry (40 min) + fold (20 min) = 90 min



- In this example:
- Sequential execution takes 4*90min = 6 hours Pipelined execution takes 30+4*40+20 = 3.5 hours
- Bandwidth BW= loads/hour
- Period T = time between two consecutive loads
- BW = 4/6 l/h without pipelining
- BW = 4/3.5 l/h with pipelining
- Pipelining improves bandwidth but not latency (90 min)
- Bandwidth limited by slowest pipeline stage
- Potential speedup = Number pipes

Vylepšenie - „big laundry“



Latency is still 90 min. but the period is 10 min. Moreover 80 min. after the start all the devices are fully utilized. The BW = 6. This module is somehow perfect. Further speed-up can be achieved by replication of these modules.

Latency: L = Time that one client spend in

Period: T = Time between two consecutive clients in/out.

Bandwith: BW = number of loads per hour. BW = 1hour/T

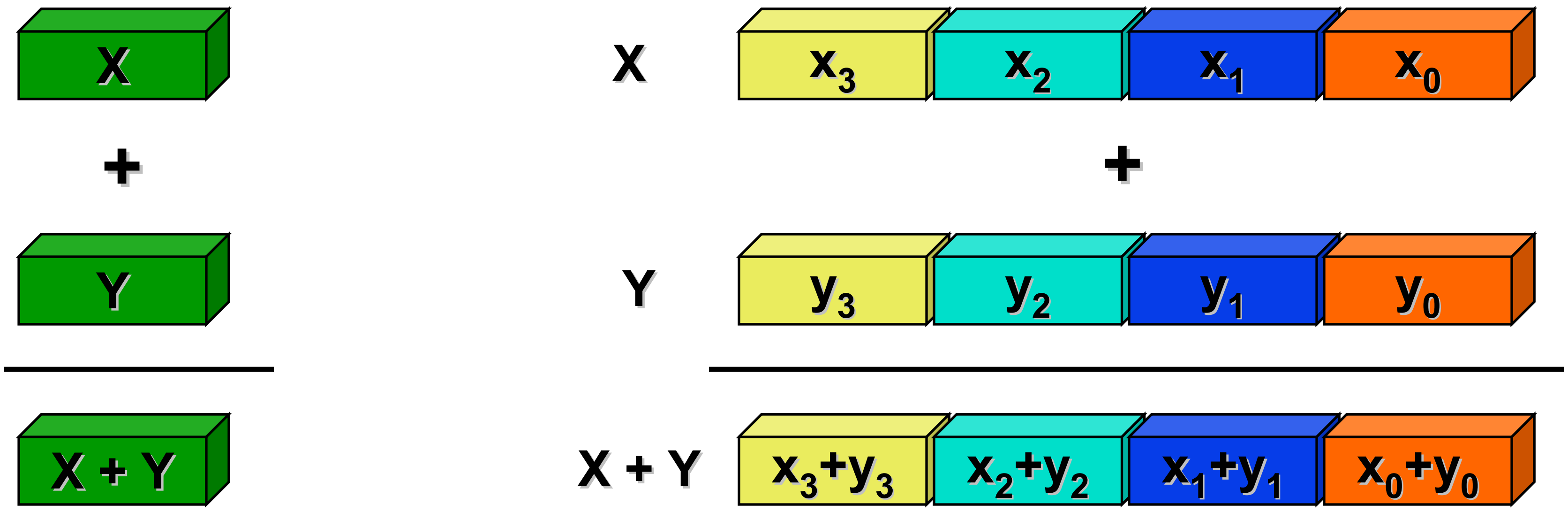
Úplná úloha

- Fabrika alebo veľká stavba
  - permanentne umiestené stroje (zariadenia počítača)
  - miesto pre dočasne umiestené stroje (podprogramy v inštrukčnej cache)
  - vnútorné sklady (cache)
    - » materiál (dáta)
    - » medzi produkty (medzi výsledky)
  - externé sklady (vyššie úrovne pamäte)
    - » materiál (dáta)
    - » medziprodukty (medzi výsledky)
    - » stroje (podprogramy)
- Cieľ optimalizovať logistiku pre minimálnu periódu alebo maximálny throughput.
  - Optimalizačné úlohy lineárne a nelineárne programovanie obvykle celočíselné.
  - Je to NP-complete až undecidable. Heuristiky.

SIMD – single instruction multiple data

zdroj Intel Corporation

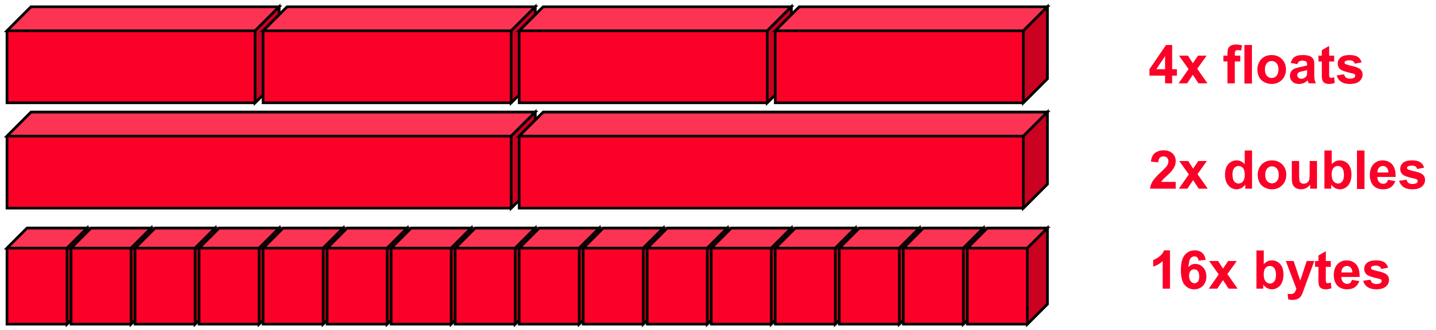
- Scalar processing
  - traditional mode
  - one operation produces one result
- SIMD processing
  - with SSE / SSE2
  - one operation produces multiple results



Všeobecnejšie riešenie je aritmetika s deleným prenosom (Grečný, Šturc 1967, Projekt RPP16, ÚTK SAV)

SSE / SSE2 SIMD on Intel

- SSE2 data types: anything that fits into 16 bytes, e.g.,



- Instructions perform add, multiply etc. on all the data in this 16-byte register in parallel
- Challenges:
  - Need to be contiguous in memory and aligned
  - Some instructions to move data around from one part of register to another

Basic Cache Optimizations

- Reducing hit time

1. Giving Reads Priority over Writes
   - E.g., Read complete before earlier writes in write buffer
2. Avoiding Address Translation during Cache Indexing

- Reducing Miss Penalty

3. Multilevel Caches

- Reducing Miss Rate

4. Larger Block size (Compulsory misses)
5. Larger Cache size (Capacity misses)
6. Higher Associativity (Conflict misses)

Advanced Cache Optimizations

- Reducing hit time
1. Small and simple caches
2. Way prediction
3. Trace caches

- Increasing cache bandwidth
1. Pipelined caches
2. Multibanked caches
3. Nonblocking caches

- Reducing Miss Penalty
1. Critical word first
2. Merging write buffers

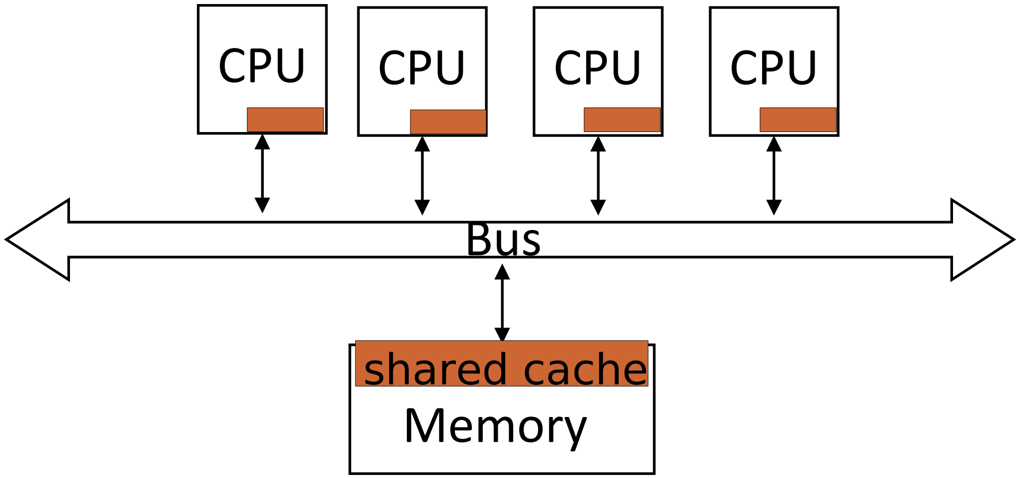
- Reducing Miss Rate
1. Compiler optimizations

- Reducing miss penalty or miss rate via parallelism
1. Hardware prefetching
2. Compiler prefetching

Modely paralelizmu

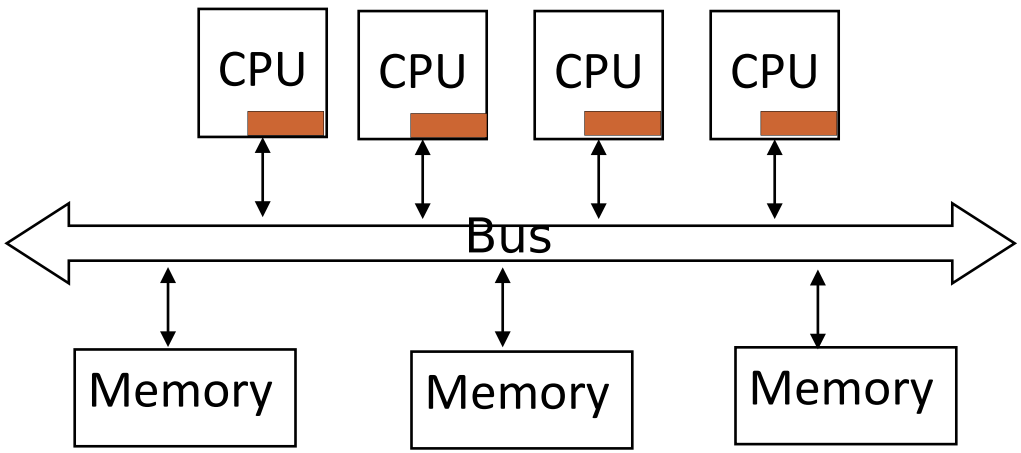
- Shared memory (Theory PRAM)
  - Jedna spoločná zdielaná pamäť
  - Viac zdielaných pamätí (distributed shared memory)
- Distributed computing. Nezávislé procesory s hierarchiou pamätí prepojené sieťou
  - Cellular architecture
  - Message passing
- SIMD Paralelný výpočet s centrálnym riadením a centrálnou synchronizáciou.

Shared memory



- Processors all connected to a large shared memory.
  - Typically called Symmetric Multiprocessors (SMPs)
  - SGI, Sun, HP, Intel, IBM SMPs (nodes of Millennium, SP)
  - Multicore chips, except that all caches are shared
- Difficulty scaling to large numbers of processors
  - <= 32 processors typical
- Advantage: uniform memory access (UMA)
- Cost: much cheaper to access data in cache than main memory.

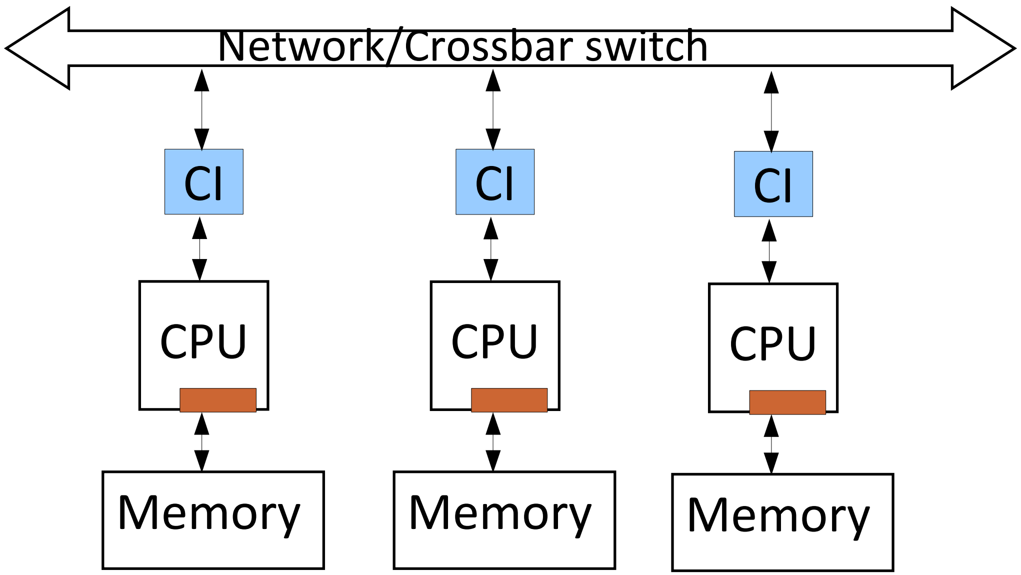
Distributed shared memory



Cache lines (pages) must be large to amortize overhead. Locality is critical to performance.

- Memory is logically shared, but physically distributed
  – Any processor can access any address in memory
  – Cache lines (or pages) are passed around machine
- SGI Origin is canonical example (+ research machines)
  – Scales to 512 (SGI Altix (Columbia) at NASA/Ames)
- Limitation is *cache coherency protocols* – how to keep cached copies of the same address consistent

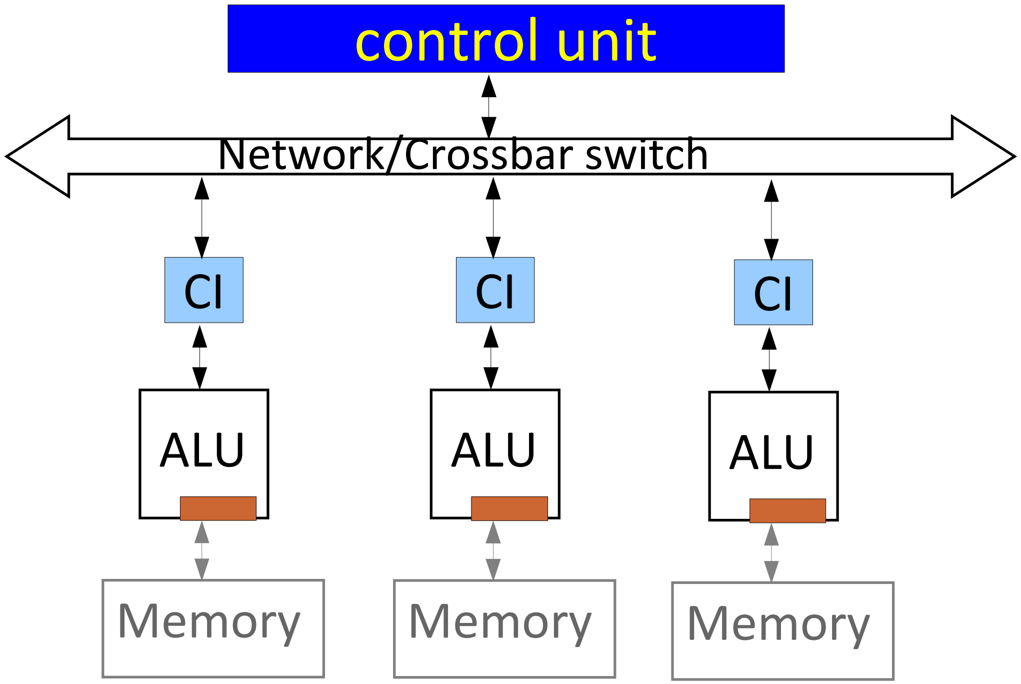
Cellular architecture



CI is a communication interface.
A distributed paralell machine.
We did not want to specify details of the interconnection and the communication interface, now.

- Each processor has its own memory and cache but cannot directly access another processor's memory.
- Each cell has a network interface (CI) for all communication and synchronization.

Globálne riadenie - SIMD



- Procesory sú obvykle jednoduche. Obvykle len aritmeticko logické jednotky. Všetky vykonávajú tú istú operáciu na svojich dátach synchrónne.
- Často sieť je mriežka (systolické systémy)

Amdahlov zákon (Amdahl's law)

- If f is the fraction of code parallelized, and if the parallelized version run on a p processor machine with no communication or paralellization overhead, the speedup is:

$$k = \frac{1}{1 - f + f/p}$$

- V ideálnom prípade $f = 1$ a $k = p$. Reálne hodnoty sú však kvôli tomu, že sa dá paralelizovať len časť kódu, a nákladom na komunikáciu oveľa menšie.
- Horšie je, že ak $f < 1$, k konverguje k $1/(1 - f)$ pre $p \to \infty$.

Loop-level parallelism – an example

```
for (i = 0; i < n; i++)
    { Z[i] = X[i] – Y[i];  Z[i] = Z[i]*Z[i]; }
```

- The loop is parallelizable because each iteration access a different set of data.
- Assume, we execute the loop on a computer with M processors.
- A processor identifier p is an integer from <0, M – 1>.

```
  b = ⌈(n/M)⌉;
for (i = b*p; i < min(b*(p+1), n); i++)
          /* processor specific constants. */
    { Z[i] = X[i] – Y[i];  Z[i] = Z[i]*Z[i]; }
```

- Single program multiple data

Data locality – example

for (i = 0; i < n; i++) Z[i] = X[i] – Y[i];
for (i = 0; i < n; i++) Z[i] = Z[i]*Z[i];

- Takáto podoba programu je zrejme nevýhodná. Keď už raz Z[i] je v registri alebo cache, treba ho využiť.
- Návrat k pôvodnému programu

for (i = 0; i < n; i++) { Z[i] = X[i] – Y[i]; Z[i] *= Z[i]; }

- Vlastne by sme mali urobiť podrobné porovnanie zložitosti priamočiareho prekladu oboch programov do „strojového kódu“.

Aspoň takto

```
i = 0;
Loop1:
*(z+i) = *(x+i) - *(y+i);
        i++ ;
        if i < n goto Loop1;
i = 0;
Loop2: *(z+i) *= *(z+i);
        i++;
        if i < n goto Loop2;
```

15 *n + 2

```
i = 0;
Loop:
        t = *(x+i) - *(y+i);
          *(z+i) = t*t;
          i++;
          if i < n goto Loop;
```

11*n + 1

„Afinné transformácie“ definícia

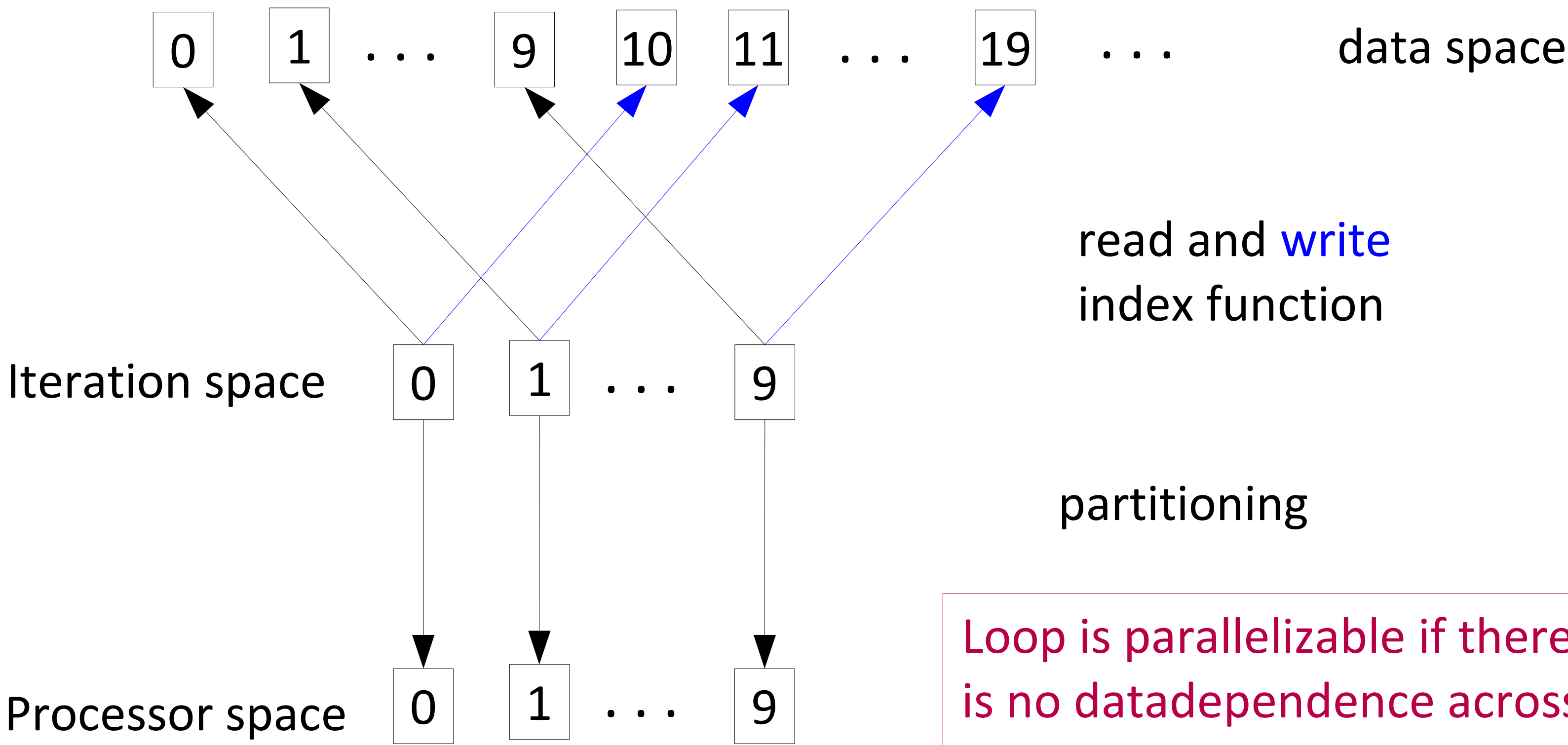
- Operate on arrays with affine acces (e.g. Fortran). No pointers and pointer arithmetic. They exploit three spaces:

0. The iteration space is the set of combination of values taken on by the loop indices.
1. The data space is the set of array element accessed.
2. The processor space is the set of processor in the system. Normally they are enumerated by integers or vectors of integers (to distnguish among them).
3. The data dependence (conflict) between two data access is, if:
   i. At least one of them is a write.
   ii. They access the same data element.

Príklad

Program: float Z[100];

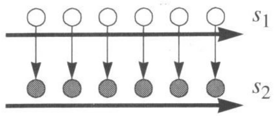
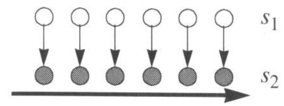
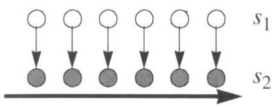
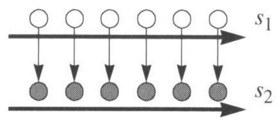
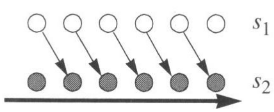
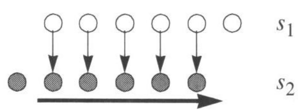
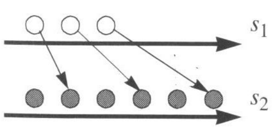
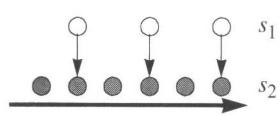
for (i = 0; i < 10; i++) Z[i+10] = Z[i];

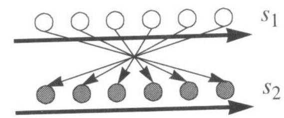
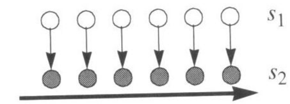
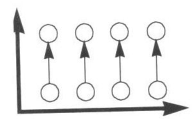
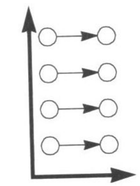
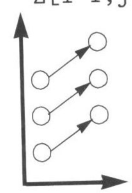
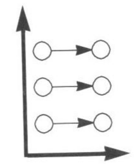


Loop is parallelizable if there is no datadependence across iteration space.

Afinné transformácie

dragon book

<table>
<tr><th>Source Code</th><th>Partition</th><th>Transformed Code</th></tr>
<tr><td>for (i=1; i<=N; i++)
  Y[i] = Z[i]; /*s1*/
for (j=1; j<=N; j++)
  X[j] = Y[j]; /*s2*/

</td><td>Fusion
$s_1 : p = i$
$s_2 : p = j$</td><td>for (p=1; p<=N; p++){
  Y[p] = Z[p];
  X[p] = Y[p];
}

</td></tr>
<tr><td>for (p=1; p<=N; p++){
  Y[p] = Z[p];
  X[p] = Y[p];
}

</td><td>Fission
$s_1 : i = p$
$s_2 : j = p$</td><td>for (i=1; i<=N; i++)
  Y[i] = Z[i]; /*s1*/
for (j=1; j<=N; j++)
  X[j] = Y[j]; /*s2*/

</td></tr>
<tr><td>for (i=1; i<=N; i++) {
  Y[i] = Z[i];   /*s1*/
  X[i] = Y[i-1]; /*s2*/
}

</td><td>Re-indexing
$s_1 : p = i$
$s_2 : p = i - 1$</td><td>if (N>=1) X[1]=Y[0];
for (p=1; p<=N-1; p++){
  Y[p]=Z[p];
  X[p+1]=Y[p];
}
if (N>=1) Y[N]=Z[N];

</td></tr>
<tr><td>for (i=1; i<=N; i++)
  Y[2*i] = Z[2*i]; /*s1*/
for (j=1; j<=2N; j++)
  X[j]=Y[j];   /*s2*/

</td><td>Scaling
$s_1 : p = 2 * i$
$(s_2 : p = j)$</td><td>for (p=1; p<=2*N; p++){
  if (p mod 2 == 0)
    Y[p] = Z[p];
  X[p] = Y[p];
}

</td></tr>
</table>

<table>
<tr><th>Source Code</th><th>Partition</th><th>Transformed Code</th></tr>
<tr><td>for (i=0; i>=N; i++)
  Y[N-i] = Z[i];   /*s1*/
for (j=0; j<=N; j++)
  X[j] = Y[j];   /*s2*/

</td><td>Reversal
$s_1 : p = N - i$
$(s_2 : p = j)$</td><td>for (p=0; p<=N; p++){
  Y[p] = Z[N-p];
  X[p] = Y[p];
}

</td></tr>
<tr><td>for (i=1; i<=N; i++)
  for (j=0; j<=M; j++)
    Z[i,j] =
      Z[i-1,j];
</td><td>Permutation
$\begin{bmatrix} p \\ q \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}$</td><td>for (p=0; p<=M; p++)
  for (q=1; q<=N; i++)
    Z[q,p] = Z[q-1,p]
</td></tr>
<tr><td>for (i=1; i<=N+M-1; i++)
  for (j=max(1,i+N);
       j<=min(i,M); j++)
    Z[i,j] =
      Z[i-1,j-1];
</td><td>Skewing
$\begin{bmatrix} p \\ q \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}$</td><td>for (p=1; p<=N; p++)
  for (q=1; q<=M; q++)
    Z[p,q-p] =
      Z[p-1,q-p-1]
</td></tr>
</table>

Násobenie matíc

for (i =0; i < n; i++)

for (j =0; j < n; j++)

{ Z[i,j] = 0.0 /* Z[i,j] sa uloží do registra. */

for (k =0; k < n; k++) Z[i,j] = Z[i,j] + X[i,k] * Y[k,j]; }

- Cache misses for a serial algorithm.
  - Assume capacity of the cache is c array elements (c|n). And caching by rows.
  - Pre prvú iteráciu (i = 0) potrebujeme priniesť celú maticu X prináša sa sekvenčne t.j. n^2/c cahe misses a celú maticu Y prináša sa na preskáčku t.j. n^2 cahe misses.
  - Ak je cache dosť veľká prvky Y v nej prežijú (všetky alebo časť), ak nie budeme ich prinášať znovu pri každej iterácii. To dá pre celý výpočet $n^2/c + n^3$ cache misses.

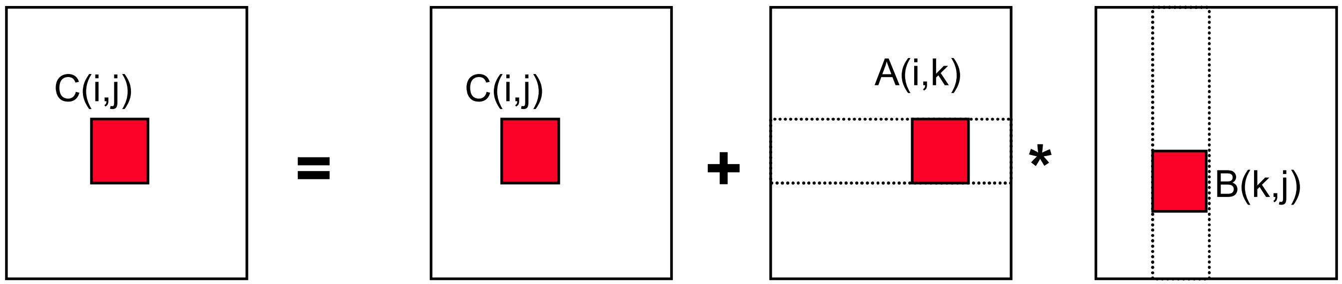
Paralelné násobenie matíc

- Pri paralelnej práci p procesorov každý procesor vypočíta n^2/p orvkov matice Z vykoná n^3/p operácii násobení a sčítaní potrebuje načítať n^2/p prvkov X a n^2 prvkov Y.
- celkový počet cahe misses je tak $(1+p)n^2/c$.
- Pre $p \to n$ konverguje počet operácii na jednom procesore (čas spotrebovaný na výpočet k n^2),
- ale cena komunikácie k $(1+n)n^2/c$. Stále $O(n^3)$.
- Zlá lokalita. Riešenie rozdelenie na bloky (tiles) veľkosti b.
- Matica $n \times n$ je videná ako $(n/b) \times (n/b)$ matica $b \times b$ matíc.

Idea násobenia po blokoch

Consider A,B,C to be n-by-n matrix viewed as N-by-N matrices of b-by-b subblocks where b=n / N is called the block size

```
for i = 1 to N
        for j = 1 to N
     {read block C(i,j) into fast memory}
     for k = 1 to N
          {read block A(i,k) into fast memory}
          {read block B(k,j) into fast memory}
           C(i,j) = C(i,j) + A(i,k) * B(k,j) {do a matrix multiply on blocks}
     {write block C(i,j) back to slow memory}
```

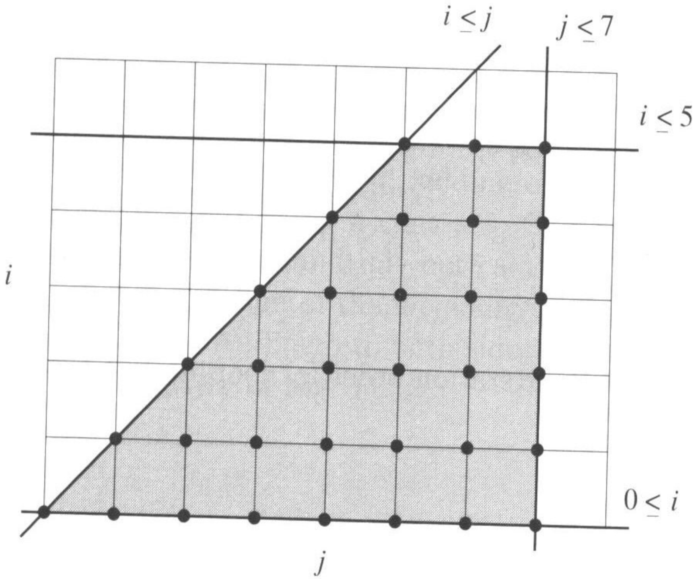


Program

```
for (ii = 0; ii < n; ii +=b)
    for (jj = 0; ii < n; jj +=b)
        for (kk = 0; ii < n; kk +=b)
            for (i = ii, i < ii + b, i++)
                for (j = jj, j < jj + b, j++)
                    for (k = kk, k < kk + b, k++)
                        Z[i,j] = Z[i,j] + X[i,k] * Y[k,j];
```

- Spracovanie matice $b \times b$ spôsobí $2b^2/c$ cache misses a požaduje b^3 sčítaní a násobení.
- Vonkajšie cykly bežia $(n/b)^3$ krát.
- To je celkove $2n^3/bc$ cache misses.
- Prínos je, že tento postup môžeme znovu aplikovať pre každú úroveň pamäťovej hierachie.

Iteration space for nested loops




```
for (i = 0; i <= 5; i++)
      for (j = i, j <= 7; j++)
            Z[j,i] = 0;
```

$$\begin{pmatrix} 1 & 0 \\ -1 & 0 \\ -1 & 1 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 5 \\ 0 \\ 7 \end{pmatrix} \ge \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

- Vo všeobecnosti, iteračný priestor d vložených cyklov je d-rozmerný mnohosten (polyhedron). Matematicky sa popisuje množinou nerovností lepšie maticovou nerovnosťou: $\{\mathbf{i} \in Z^d: \mathbf{Bi} + \mathbf{b} \ge \mathbf{0}\}$
- Adresáciu reprezentuje výraz **Fi** + **f**, kde **F** je d$\times$d matica.
- Ak dve iterácie **i** a **i'** pristupujú k tomu isténu prvku poľa, potom **F**(**i**-**i'**) = **0**. Vo všeobecnosti **Fi** + **f** = **Fi'** + **f'**.

Formalizácia prístupu k poliam

- Prístup k prvku pola pre vložené cykly hĺbky d je štvorica $\mathcal{F}$ = (**F**, **f**, **B**, **b**), kde **Bi** + **b** ≥ **0** je systém nerovností pre ohraničenia a **Fi** + **f** je adresa prvku.
- Dva prístupy k poľu $\mathcal{F}$ = (**F**, **f**, **B**, **b**) a $\mathcal{F}'$ = (**F'**, **f'**, **B'**, **b'**) sú v konflikte (dátová závislosť), ak platí:
  1. Aspoň jeden z nich je zápis.
  2. Existujú i$\in Z^d$ a i'$\in Z^{d'}$ také že:
     a) **Bi** ≥ **0**

     b) **B'i'** ≥ **0**

     c) **Fi** + **f** = **F'i'** + **f'**.

Euklidov algoritmus (gcd)

- Pretože, najväčší spoločný deliteľ dvoch celých čísieľ, je najväčší spoločný deliteľ ich absolutných hodnôt. Môžeme bez ujmy na všeobecnosti predpokladať, že všetky čísla sú kladné (nulu môžeme vynechať): gcd(0, 0) nedefinované, gcd(0, x) = x, gcd(x, y) = gcd(y, x).

```
integer function gcd(integer x, integer y)
    { if (x > y ) swap(x,y);
        if (y == 0) {error; break}
        return ((x == 0) ? y : gcd(y%x, x)) }
```

- Pre viac argumentov:

$$gcd(x_1, x_2, x_3, \ldots x_n) = gcd(gcd(x_1, x_2), x_3, \ldots x_n).$$

Fourier-Motzkinova eliminácia

- Projekcia n dimenzionálneho mnohostenu na n-1 dimenzionálny mnohosten.
- Daný je systém nerovníc S (polyhedron) s premennými $x_1, \dots , x_{m-1}, x_m, x_{m+1}, \dots , x_n$.
- Vytvoriť systém nerovníc S' (polyhedron) s premnnými $x_1, \dots , x_{m-1}, x_{m+1}, \dots , x_n$, ktorý neobsahuje premennú x_m a je priemetom S do podpriestoru zvyšných premenných
- Nech C je množina všetkých nerovností obsahujúcich premnnú x_m.
- Každá z nerovností v C sa dá upraviť na tvar $L \leq cx_m$ alebo na tvar $dx_m \leq U$, kde c a d sú kladné konštanty.
- Nech D = Ø.

Fourier-Motzkinova eliminácia algoritmus

- Pre každdú dvojicu podmienok:

  $L \leq cx_m$

  $dx_m \leq U$

- pridáme do D nerovnicu $dL \leq cU \quad | /gcd(c, d)$.
- $S' = S – C \cup D$.
- Splniteľnosť (prázdnosť): Pretože platí, že S' je priemet S. Musia oba mnohosteny byť prázdne alebo neprázdne súčasne.
- Ak čo i len jedna nerovnica je nesplniteľná je S aj S' prázdne.
- Trivialne nerovnice (nerovnice neobsahujúce premenné). Napr: $u \leq x_m \leq v$ (u a v konštanty) generuje $u \leq v$.

Výpočet ohraničení pre dané poradie premenných

Vstup: Kovexný mnohosten S s premennými $x_1, x_2, \dots, x_n$ (S je množina nerovníc obsahujúcich uvedené premenné).

Výstup: Dolné a horné ohraničenia L_i a U_i také, že každé ohraničenie obsahuje nanajvýš premenne x_j pre $j < i$.

Algoritmus:

$S_n = S$;

for (i = n; i > 1; i--)

{ L_i = set of all lower bounds on x_i in S_i;

U_i = set of all upper bounds on x_i in S_i;

Compute S_{i-1} by elimination variable x_i using Fourier-Motzkin;

}

Odstránenie zbytočných ohraničení

$S' = \emptyset$;

for ($i = 1$; $i \leq n$; i++)

{ remove bounds in L_i and U_i implied by S';

add the remaining constraints of L_i and U_i on x_i to S';

}

GCD test

Veta: Lineárna diofantická rovnica $a_1x_1 + a_2x_2 + ... + a_nx_n = c$ má riešenie práve vtedy, keď $gcd(a_1, a_2, ... , a_n)$ delí c.

- Riešenie systému lineárnych diofantických rovníc: „Gausova eliminácia“ riadená GCD testom.
- Po každej eliminácii otestujeme, či vzniknutá rovnica splňuje GCD test.

Príklad:

$$x - 2y + z = 0$$
$$3x + 2y + z = 5$$

- Obe rovnice splňujú gcd test. Po eliminácii x z druhej rovnice vznikne $8y - 2z = 5$. Nesplňuje gcd test, teda riešenie neexistuje.

Riešenie nerovníc – celočíselné lineárne programovanie

Vstup: Konvexný mnohosten S_n v premenných $x_1, x_2, \dots, x_n$

Výstup: True, ak S_n je neprázdny (obsahuje aspoň jeden bod s celočíselnými súradnicami. Inak False.

Postup:

1) Postupne eliminuj premenné v poradí $x_n, x_{n-1}, \dots, x_1$. Nech S_i je mnohosten po odstránení i+1. premennej.
2) **if** (S_0 is empty) **return**(false); /* nesplniteľné ohraničenia No solution*/
3) **for** (i = 1; i ≤ n; ii++)
   { **if** (S_i is empty) **break**;
   pick c_i an integer in the middle of range for x_i in S_i;
   modify S_i by replacing x_i by c_i; }
4) **if** (i== n +1) return(true); **if** (i== 1) **return**(false);
5) Nech l_i a u_i sú dolná a horná hranica pre x_i v S_i.
6) Rekurzívne aplikuj algoritmus na $S_n \cup \{x_i \le \lfloor l_i \rfloor\}$ a $S_n \cup \{x_i \ge \lceil u_i \rceil\}$. Ak niektorá z týchto aplikácii vráti true, vráť true. Inak vráť false.

Farkasova lema

Veta: Nech **A** je m×n matica reálnych čísieľ a **c** je reálny nenulovy n rozmerný vektor. Potom zo systémov $\mathbf{Ax} \geq \mathbf{0}$, $\mathbf{c}^T\mathbf{x} < \mathbf{0}$ (primal) a $\mathbf{A}^T\mathbf{y} = \mathbf{c}$, $\mathbf{y} \geq \mathbf{0}$ (dual), práve jeden má reálne riešenie.

- Je to známa veta lineárneho programovania.
- Duálny systém sa dá riešiť Fourier-Motzkinovou elimináciou vyeliminovaním premenných **y**.
- Ak duálny systém má riešenie potom pre všetky splňujuce $\mathbf{Ax} \geq \mathbf{0}$, platí $\mathbf{c}^T\mathbf{x} \geq \mathbf{0}$.