

# Implementace algoritmu Merge-splitting Sort

(projekt do předmětu PDA)

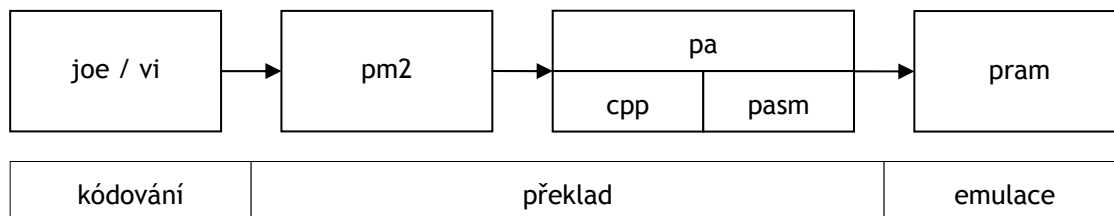
# Obsah

Obsah .....	1
Zadání projektu .....	2
Cíl projektu .....	2
Časový harmonogram řešení projektů .....	2
Odevzdání projektů: .....	2
Postup řešení projektu .....	3
Použitý systém .....	3
Implementace algoritmu .....	3
Experimentování .....	4
Bubble-sort .....	5
Quick-sort .....	5
Rozdíly mezi Quick a Bubble sort-em .....	6
Zdrojový kód programu .....	7
Použitá literatura .....	9

## Zadání projektu

### Cíl projektu

Cílem projektu je implementovat dva algoritmy pro paralelní zpracování dat. Algoritmy budou implementovány v jazyce *pm2* a spuštěny emulátorem *pram*. Cesta od napsání kódu k emulaci je ukázána na následujícím obrázku:



Kód programu lze přeložit do jazyka *PEL* (jazyk obdobný jazyku symbolických instrukcí) překladačem *pm2*. Dále by měl proběhnout překlad z jazyka *PEL* do binárního formátu. Dávka *pa* provádí tento překlad ve dvou krocích, a to nejprve spuštěním preprocesoru *cpp* na zadaný soubor a jeho následný překlad assemblerem *pasm*. Pokud překlad proběhne bez chyby, lze výsledný kód spustit emulátorem *pram*.

### Časový harmonogram řešení projektů

#### 1. fáze, do 18.11.2004

V první fázi řešení projektu se proto seznámte s jazyky *pm2*, případně *PEL* a instalujte (přeložte) potřebné programy. Tato fáze nemá výsledek ve formě zprávy či aplikace, ale má sloužit jako příprava na další úkoly. Důkladně proto ověřte chování překladačů a emulátoru v prostředí, které se rozhodnete používat.

#### 2. fáze, do 9.12.2004

##### Implementace algoritmu Merge-splitting Sort.

V prostředí jazyka *PM2* implementujte algoritmus Merge-splitting Sort a přeložte jej a následně ověřte jeho chování pomocí emulátoru *Pram*. Velikost vstupních dat i princip rozdělení mezi procesory si řešitel zvolí sám. Zvoleny postup ale musí být popsán v dokumentaci.

#### 3. fáze, do 23.12.2004

Implementace algoritmu pro paralelní výpočet, bude upřesněno.

### Odevzdání projektů:

V papírové podobě průběžně na přednáškách (pro 2. a 3. fázi v den termínu, tj. 9.12.2004 a 23.12.2004). Obsahem práce bude komentovaný výpis algoritmu v jazyce *PM2*, stručný rozbor algoritmu a jeho analýza a dále rozbor provedených experimentů.

NEW!. V termínech se odevzdávají projekty i v elektronické formě. To bude probíhat přes fakultní informační systém. Odevzdán bude jeden archiv souborů pojmenovaný loginem odevzdávajícího studenta s příponou zvoleného archivačního programu. Obsahem bude zdrojový kód v jazyce *pl2* a dokumentace, která bude odpovídat dokumentaci odevzdané v papírové formě.

4.11.2004, František Zbořil ml.

## Postup řešení projektu

### Použitý systém

Překladač *pm2* a emulátor *pram* jsem používal na operačním systému Linux, distribuce Gentoo Linux ([www.gentoo.org](http://www.gentoo.org)). Bylo několik problémů na které jsem při práci narazil:

- potřeba odstranění *-lterm* z makefile
- určení správné cesty k preprocesoru *cpp*

Protože neustálé překládání přes tři příkazy není uživatelsky příjemné, tak jsem si napsal skript *run* který udělá vše potřebné:

```
#!/bin/sh
clear
if [ -f "$1" ]
then
    rm -f "$1.pel" 2>&1 1>/dev/null
    ./pm2 "$1" "$1.pel"
    if [ -f "$1.pel" ]
    then
        rm -f "$1.bin" 2>&1 1>/dev/null
        ./pa "$1.pel" "$1.bin"
    if [ -f "$1.bin" ]
    then
        ./pram -t "$1.bin"
        rm -f "$1.bin" 2>&1 1>/dev/null
    else
        echo "Compilation of PEL failed. No BIN file has been created."
    fi
    rm -f "$1.pel" 2>&1 1>/dev/null
else
    echo "Compilation PM2 -> PEL failed. No PEL file has been created."
fi
else
    echo "Can not found the source file ($1) or no filename given"
fi
```

### Implementace algoritmu

Detailní popis algoritmu je popsán v dokumentu [1] a podle něho ho lze jednoduše implementovat, pokud teda umíme syntaxi jazyka *pm2*. Jelikož jsem nenašel žádný popis tohoto programovacího jazyka, musel jsem přijít na několik věcí metodou pokus-omyl. Postup můžu shrnout do několika bodů:

- procedura *init*, definice pole, výpis prvků
- procedura *finish* s výpisem
- kreslení čáry pomocí vypisování nul v cyklu jelikož nejde vypsat znaky, jen čísla
- čára jako procedura, bohužel pak nejde spustit program, protože výpis je možno provádět jenom a jenom v procedurách *init* a *finish*
- zjištění jak detekovat číslo procesoru, stačí použít proměnnou cyklu typu *par*
- řazení bubble-sort, otestování algoritmu
- implementace merge a split
- experimentování
- oprava chyb
- implementace lepšího počátečního řazení - quick sort algoritmus dle [2]
- další experimenty

## Experimentování

Jelikož na (funkčním) programu není nic zajímavého, rozhodl jsem se udělat několik experimentů, které ukáží vlastnosti daného řadícího algoritmu. Především mě zajímala doba potřebná k seřazení pole vztažena na 1 prvek v závislosti od počtu prvků na procesor a počtu procesorů.

K dosažení daného cíle jsem napsal testovací program v skriptovacím jazyce PHP. Program v jazyce *pm2* jsem upravil následovně:

```
const p = <?php echo $p; ?>;
const nPP = <?php echo $npp; ?>;

procedure init;
begin
  P := p;
  <?php $data = array();
  for($i=0;$i<$n;$i++)
    echo "    d[$i] := ", $data[]=rand(1,32000), ";\n";
?>
end init;
```

Výpočet teda sestával z:

- vygenerování pole náhodných čísel
- překladu
- emulace
- zjištění správnosti řešení
- zjištění doby výpočtu

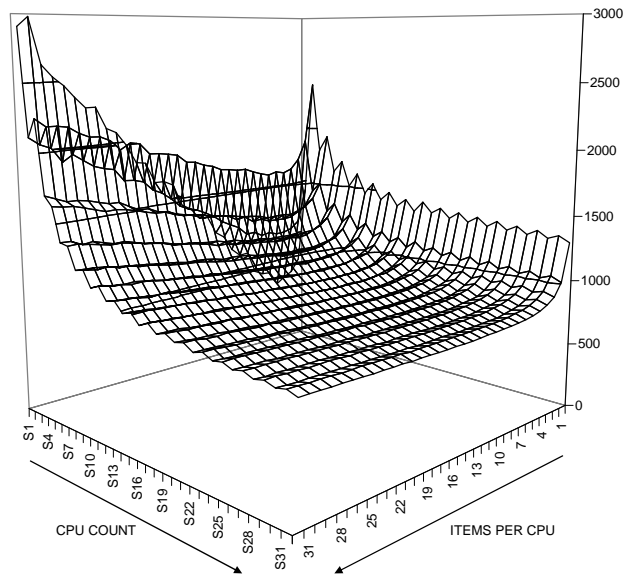
Protože věci dělám pořádně, zahrnul jsem do testu porovnání na správnost výpočtu. Tento test ale ukázal, že program neřadí vždy správně a při zvyšujícím se počtu prvků je pravděpodobnost chybovosti větší. Oprava spočívala v přidání příkazu *synchronize* za každý cyklus typu *par*. Ten příkaz je potřebný protože procesory pracují nezávisle na sobě ale se stejnými daty a délka trvání počátečního řazení závisí na obsahu pole. V praxi se to projevilo tak, že když se uvolnil procesor, tak byl přiřazen ihned operaci *merge-split* a to způsobilo chyby v řazení. Po doplnění synchronizačních bodů funguje již program bez chyby.

Detaily experimentu:

- dva různé počáteční řadící algoritmy ( *bubble-sort*, *quick-sort* )
- 1 až 32 procesorů
- 1 až 32 prvků pole na procesor
- výsledek je průměrem 5-ti pokusů

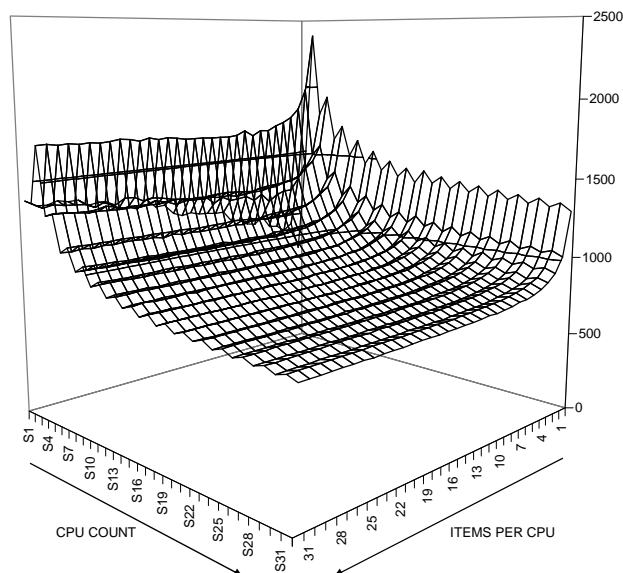
Výpočet varianty s *quicksort-em* spotřeboval 69 minut 41 sekund čistého času procesoru Intel Celeron Tualatin (jádro PIII, 256 kB L2 cache, takt 1568 MHz).

## Bubble-sort



Z grafu je hned jasné, že efektivita algoritmu závisí na počtu procesorů, resp. na tom, jestli je počet procesorů sudý nebo lichý. Pro situaci s jedním procesorem jsou výsledky odlišné od zbytku, co může být způsobeno nejspíš režii synchronizace.

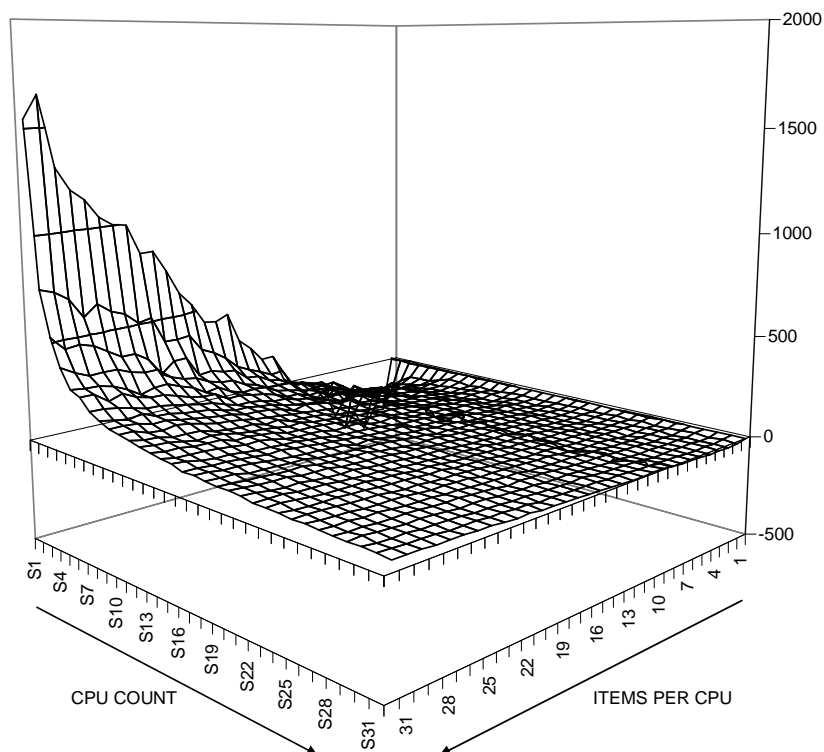
## Quick-sort



Při porovnání s předchozím grafem se dá zjistit, že algoritmus quick-sort má zcela jinou časovou náročnost než algoritmus bubble-sort. Závislost na sudém nebo lichém počtu procesorů ale zůstává, protože počet procesorů má vliv jenom na operaci merge-split, která byla stejná v obou případech.

## Rozdíly mezi Quick a Bubble sort-em

Pro ilustraci jsem sestrojil i graf rozdílu:



Z grafu se dá zjistit několik faktů, především je to rozdíl v časové náročnosti počátečních řadících algoritmů (kopeček u velkého počtu prvků na procesor ale málo procesorech, tj. levá část grafu). Dále je vidět, že pro malý počet prvků není výhodné používat algoritmus quicksort, jelikož jeho inteligence má velkou režii oproti algoritmům které řadí jednodušeji (propad v počátku grafu). Další zajímavost je, že zvyšující se počet procesorů smazává rozdíly mezi náročností počátečních řazení (vyhlazování grafu při zvyšujícím se počtu procesorů).

## Zdrojový kód programu

```
program mergeSplittingSort;

const p = 4;
const nPP = 3;
const n = p*nPP;

sharedvar d : array[0..n-1] of word;

procedure init;
begin
    P := p;
    d[ 0] := 12;
    d[ 1] := 9;
    d[ 2] := 10;
    d[ 3] := 11;
    d[ 4] := 7;
    d[ 5] := 4;
    d[ 6] := 3;
    d[ 7] := 6;
    d[ 8] := 2;
    d[ 9] := 1;
    d[10] := 8;
    d[11] := 5;
end init;

procedure finish;
var i : word;
begin
    for i := 0 to n-1 do
        writeln( d[i] );
    end;
end finish;

procedure swapData( x,y : word );
var temp : word;
begin
    temp := d[x];
    d[x] := d[y];
    d[y] := temp;
end swapData;

procedure split( start,stop : word ) : word;
var left: word;
    right: word;
    pivot: word;
begin
    pivot := d[start];
    left := start + 1;
    right := stop;
    while left <= right do
        while (left <= stop) and (d[left] < pivot) do
            left := left + 1;
        end;
        while (right > start) and (d[right] >= pivot) do
            right := right -1;
        end;
        if left < right then
            swapData(left,right);
        end;
    end;
    swapData(start,right);
    return right;
end split;

procedure quickSort( start,stop : word );
var splitpt : word;
begin
    if start < stop then
        splitpt := split(start,stop);
        quickSort( start, splitpt-1 );
        quickSort( splitpt+1, stop );
    end
end quickSort;
```

```

procedure sortOwn( pid : word );
begin
    if nPP > 1 then
        quickSort( pid * nPP
                    , ( pid + 1 ) * nPP - 1
                    );
    end;
end sortOwn;

procedure doMerge( baseCPU : word );
var tmp : array[0..2*nPP-1] of word;
    s1i : word;
    s1n : word;
    s2i : word;
    s2n : word;
    i : word;
begin
    s1i := (baseCPU + 0) * nPP;
    s2i := (baseCPU + 1) * nPP;
    s1n := nPP;
    s2n := nPP;
    for i := 0 to 2*nPP-1 do
        if s1n > 0 then
            if s2n > 0 then
                if d[s1i] < d[s2i] then
                    tmp[i] := d[s1i];
                    s1i := s1i + 1;
                    s1n := s1n - 1;
                else
                    tmp[i] := d[s2i];
                    s2i := s2i + 1;
                    s2n := s2n - 1;
                end;
            else
                tmp[i] := d[s1i];
                s1i := s1i + 1;
            end;
        else
            tmp[i] := d[s2i];
            s2i := s2i + 1;
        end;
    end;
    s1i := baseCPU * nPP;
    for i := 0 to (2*nPP-1) do
        d[s1i+i] := tmp[i];
    end;

end doMerge;

var i : word;
    k : word;
begin
    par i := 0 to p-1 do
        sortOwn(i);
    end;
    synchronize;
    for k := 1 to (p+1) div 2 do
        par i := 0 to p-2 do
            if ( i mod 2 ) = 0 then
                doMerge(i);
            end;
        end;
        synchronize;
        par i := 1 to p-2 do
            if ( i mod 2 ) = 1 then
                doMerge(i);
            end;
        end;
        synchronize;
    end;
end mergeSplittingSort.

```

## Použitá literatura

- [1] **Algoritmy řazení a poznámky**  
<http://www.fit.vutbr.cz/study/courses/PDA/private/www/pda02.doc>
  
- [2] **Pascal Quicksort**  
[http://sandbox.mc.edu/~bennet/cs404/doc/qsort\\_pas.html](http://sandbox.mc.edu/~bennet/cs404/doc/qsort_pas.html)
  
- [3] **Hämäläinen, P.: PRAM EMULATOR - User's Manual**  
<ftp://ftp.cs.joensuu.fi/pub/Software/pram/emulator.tar.Z> - userman.ps