FAKULTA MATEMATIKY, FYZIKY
A INFORMATIKY
UNIVERZITY KOMENSKÉHO
v Bratislave

D I P L O M O V Á P R Á C A

2005 Tomáš Mečíř

Fakulta matematiky, fyziky a informatiky Univerzity Komenského
Bratislava

Katedra informatiky

Diplomová práca

Vplyv konečnej šírky pásma na speed-up paralelných úloh

2005 *Tomáš Mečíř*

Ďakujem svojmu školiteľovi,
RNDr. Igorovi Odrobinovi, CSc.,
za cenné podnety, rady a
všestrannú pomoc a podporu
pri spracovávaní mojej
diplomovej práce

Prehlasujem, že som diplomovú
práca vypracoval samostatne
a s použitím uvedenej literatúry

V Senici, 25.4.2005 *Tomáš Mečíř*

Úvod

Paralelné spracovanie je zvyčajne vnímané ako prostriedok, pomocou ktorého je možné dosiahnuť lepší čas spracovania algoritmov. Mojím cieľom je ukázať, že toto tvrdenie je síce pravdivé, neplatí však pre všetky algoritmy. Prakticky som otestoval dva rôzne algoritmy, ktoré som vyberal tak, aby reprezentovali širšiu triedu algoritmov s istými vlastnosťami. Ide najmä o spôsob prístupu k dátam a možnosť rozdeliť algoritmus na menšie podúlohy.

Analýza algoritmov sa väčšinou zameriava na základný popis operácií a uspokojí sa s odhadom počtu operácií. Tento odhad avšak nemusí byť vždy postačujúci. Napríklad, väčšina triediacich algoritmov má zložitosť $O(n \log n)$.

Takéto odhady sú síce veľmi užitočné pri navrhovaní algoritmov, avšak neposkytujú dostatočnú informáciu o niektorých dôležitých aspektoch algoritmov, ktoré môžu mať v niektorých prípadoch zásadný vplyv na výkonnosť algoritmu na reálnych architektúrach. Preto je potrebné zohľadňovať celkovú efektivitu algoritmov.

Celková efektivita nemá vplyv len na paralelné spracovanie. Vzhľadom na vlastnosti existujúceho výpočtového vybavenia je možné tieto princípy aplikovať aj na algoritmy, ktoré pracujú na jednom procesore. V tejto práci sa pokúsim tiež zistiť, či je možné vhodným využitím vlastností cieľovej architektúry dosiahnuť urýchlenie aj týchto algoritmov, a ak áno, za akých podmienok je toto možné.

Dôležitým faktorom je v tomto prípade aj operačný systém a kompilátor. V tejto práci budem využívať operačný systém Linux s jadrom 2.6 a kompilátor Intel C++ 8.1 [ICC]. Všetky tieto nástroje sú voľne prístupné na internete.

Parametre výpočtového vybavenia

Pri zohľadňovaní celkovej efektivity je nevyhnutné poznať parametre použitého výpočtového vybavenia, keďže práve tieto spôsobujú, že sa algoritmy môžu správať inak, než ako by sa dalo očakávať podľa ich asymptotickej zložitosti. Ide najmä o nasledovné parametre:

Šírka prenosového pásma – udáva objem dát, ktorý je možné preniesť za časovú jednotku. Zvyčajne sa udáva v Mbit/s. S využitím tohto parametra je možné určiť dolné časové ohraničenie pre algoritmus, pokiaľ je známe množstvo dát, ktoré je potrebné preniesť. Prenosovým pásmom môže byť zbernica (v tom prípade sa prenáša z pamäti RAM) alebo sieťový kábel. Prenos z pamäti nastáva pri jednoprocesorových aj pri viacprocesorových algoritmoch, v prípade viacprocesorových algoritmov môže (ale nemusí) byť ich súčasťou aj sieťový prenos.

Latencia – oneskorenie, čas potrebný na prenos dát cez prenosové pásmo. Má vplyv najmä na algoritmy, ktoré nevyužívajú pamäť sekvenčne, a preto musia čakať pri každom čítaní z pamäte celý tento čas, nakoľko nedokážu využiť *prefetch* (viz nižšie).

Veľkosť cache – vyrovnávacia pamäť, do ktorej sa ukladajú často používané dáta. Vhodným využitím vlastností pamäti cache je možné docieliť výrazné zrýchlenie mnohých algoritmov. Podrobnejšie postupy sú popísané ďalej. Kľúčovým parametrom je veľkosť tejto pamäti. Navyše, táto pamäť býva často viacúrovňová, pričom nižšie úrovne cache sú menšie a rýchlejšie.

Cache-line – veľkosť jedného záznamu prenášaného medzi pamäťou a cache. Pri prenášaní dát sa vždy prenášajú bloky takejto veľkosti (alebo násobky). Pokiaľ algoritmus požaduje menšie množstvo dát, než je veľkosť cache-line, doplnia sa nasledujúce dáta až do veľkosti cache-line. Táto vlastnosť architektúry sa nazýva **prefetch**.

Pokiaľ má algoritmus bežať paralelne na viacerých počítačoch, je potrebné zohľadňovať aj architektúru sieťového vybavenia. Táto je do istej miery analogická s prípadom viacprocesorového systému.

Vplyv týchto parametrov na algoritmus je študovaný napríklad v [Parello02] a [Yang1-98].

OpenMP

OpenMP je technológia, ktorá umožňuje jednoduchú realizáciu viacvláknových programov, ktoré bežia na viacerých procesoroch v rámci jedného počítača. Dochádza teda k zdieľaniu pamäťového priestoru, každý procesor má ale svoju vlastnú cache pamäť. Ďalšie informácie sú k dispozícii v [OPENMP]. V tejto práci som využil implementáciu OpenMP, ktorá je súčasťou kompilátora Intel C++. [ICC]

MPI

MPI (*Message Passing Interface*) je technológia umožňujúca jednoduchú komunikáciu medzi procesmi bežiacimi na rôznych počítačoch. K dispozícii je viacero kompatibilných implementácii pre viaceré programovacie jazyky. V tomto prípade neexistuje spoločná pamäť, je preto potrebné prenášať dáta po sieti, čo môže mať za následok spomalenie algoritmu – v niektorých prípadoch môže viacprocesorový algoritmus bežať dokonca pomalšie než ekvivalentný algoritmus na jednom procesore. Konkrétny prípad demonštrujem neskôr. Merania popísané v tejto práci som uskutočňoval s využitím implementácie [LAMMPI].

Pri písaní programov som využíval niektoré postupy, ktoré sa v tejto oblasti využívajú. Tieto je možné nájsť napríklad v [Quammen01].

Použité architektúry

Merania som uskutočňoval na nasledovných architektúrach:

1. Dual Intel Xeon, 3.06 Ghz, 512 KB L2 cache, 2 MB L3 cache, 2 GB RAM

Na tejto architektúre som testoval výkon jednoprocesorových algoritmov, a tiež výkon na dvoch procesoroch s využitím OpenMP.

2. Cluster 4 počítačov, každý Intel Pentium 4, 2.4 Ghz, 512 KB L2 cache, 512 MB RAM

Počítače boli spojené gigabitovým ethernetom, cez switch s priepustnosťou 48 Gbit/sec.

Micro-benchmarky

Micro-benchmarky sú malé testovacie programy, ktoré umožňujú empiricky stanoviť parametre výpočtového vybavenia. Najčastejšie sa používajú na stanovenie reálnej šírky prenosového pásma, prípadne na empirické určenie veľkosti pamäti cache. Tiež sa dajú využiť na určenie niektorých ďalších vlastností vybavenia.

Na testovanie **pamäťového systému** som použil dva micro-benchmarky. Prvý meral rýchlosť čítania a zápisu do pamäti postupne – najskôr meral len rýchlosť čítania, potom len rýchlosť zápisu. Druhý micro-benchmark testoval reálnejšiu situáciu – súčasné čítanie a zápis (s využitím príkazov typu a[i] = a[j] + a[k], pričom premenné i, j a k boli uložené v registroch, aby neskresľovali výsledok).

Výsledky sú nasledovné:

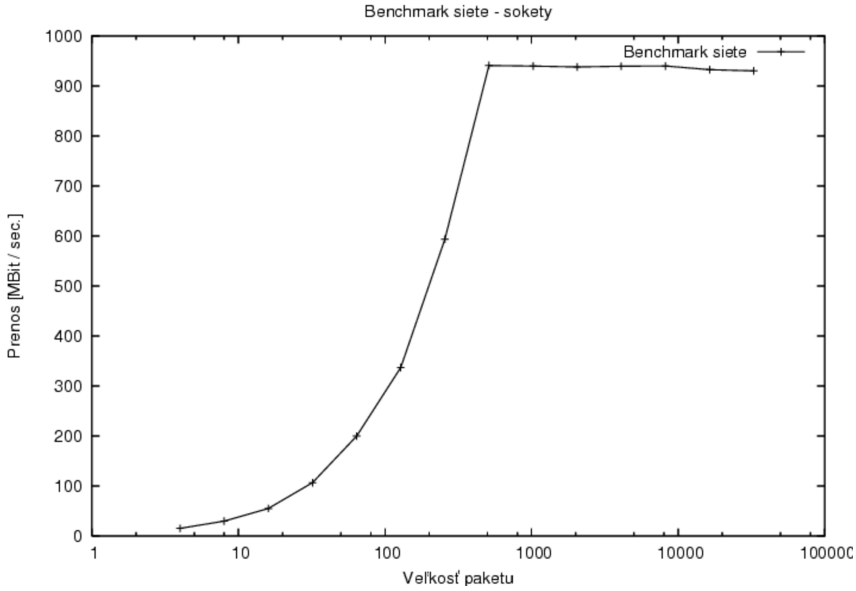
Architektúra 1 (Xeon): čítanie *1702 MB/sec.*, zápis *1160 MB/sec.*, čítanie+zápis *2870 MB/sec.*

Architektúra 2 (P4): čítanie *1852 MB/sec.*, zápis *1347 MB/sec.*, čítanie+zápis *2347 MB/sec.*

Testovanie **sieťového systému** som realizoval dvomi odlišnými prístupmi.

Prvý prístup bol s využitím štandardných TCP socketov. Nakoľko sieťový prenos je realizovaný pomocou sieťových paketov, je potrebné testovať prenos rôzne veľkých blokov údajov. Zároveň som realizoval vždy dva prenosy súčasne, medzi rôznymi dvojicami počítačov, čím som zistil, že

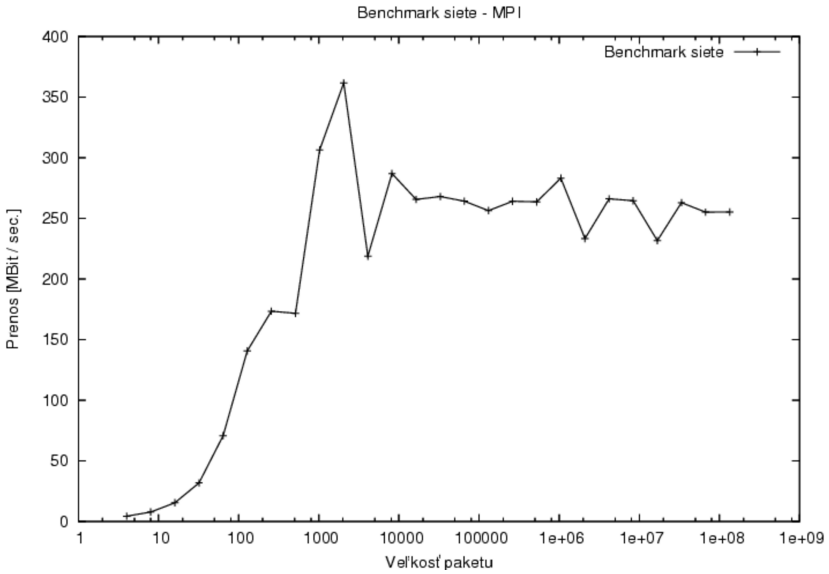
použitý sieťový switch má dostatočnú vnútornú priepustnosť, aby dokázal realizovať viacero nezávislých prenosov bez zníženia priepustnosti siete.



Z grafu je vidieť, že ak je veľkosť jedného prenášaného bloku aspoň 512 bajtov, je sieť schopná prenášať dáta maximálnou rýchlosťou – 1 Gbit / sec. Pre menšie veľkosti blokov už nedochádza k efektívnemu využitiu zdrojov sietre, nakoľko veľkosť rámca v sieťach Ethernet je konštantná a príliš malý sieťový paket nezaplní celý rámec.

Druhý prístup som realizoval pomocou knižníc MPI. V tomto prípade sa mi nepodarilo maximálne využiť priepustnosť siete. Menšia prenosová rýchlosť je pravdepodobne spôsobená tým, že pri prenose sú dáta kódované a následne dekódované, v dôsledku čoho majú prenášané dáta väčšiu veľkosť.

Namerané údaje sú zachytené v nasledujúcom grafe:



Z grafu je vidieť, že šírku prenosového pásma sa darí využívať asi na 30%, rýchlosť prenosu kolíše okolo 300 MBit / sec.

Celková efektivita

Ako som uviedol vyššie, v realite je rýchlosť vykonania algoritmu daná parametrami výpočtového vybavenia. Na formálny popis vplyvov týchto parametrov bolo vytvorených viacero modelov. Na ohodnotenie sa bežne sa používajú modely ako PRAM (Parallel Random Access Machine), ktorý štandardne obsahuje obmedzenie prístupu k dátam typu EREW, ERCW, CREW, CRCW.

Na ohodnotenie reálnej výkonnosti programov v praxi sa vyžaduje používanie modelov bližších technologickej realite. Jednými z najvýznamnejších sú dvojparametrový model (tzv. *α-β model*), zohľadňujúci dobu prístupu do pamäte, resp. komunikáciu medzi procesormi, dodatočným parametrom, ktorý popisuje šírku pásma. Uvediem tiež tzv. LogP model, ktorý ešte presnejšie odráža zložitosť prenosu dát medzi procesormi. Architektúra a vlastnosti súčasných počítačov viedli aj k zavedeniu tzv. BSP (*Bulk Synchronous Parallel*) modelu.

Dvojparametrový alfa-beta model

Tento model pre každý algoritmus zohľadňuje dva parametre – dobu výpočtu a dobu prenosu dát. Pri určovaní doby prenosu sa predpokladajú dve úrovne hierarchie pamäte – pomalá a rýchla. Na začiatku sú všetky dáta v pomalej pamäti, algoritmus sa snaží optimalizovať tak, aby čo najväčšia časť prístupov k dátam bola vykonaná v rýchlej pamäti.

BSP model

Model BSP (*Bulk Synchronous Parallel*) bol navrhnutý v roku 1990. [Valiant90] Predpokladá, že algoritmus pozostáva z dvoch fáz. Prvá je výpočtová fáza, počas ktorej procesory vykonávajú výpočty nad lokálnymi dátami, prípadne čítanú dáta z globálnej pamäte. Druhá fáza sa nazýva komunikačná. Počas tejto fázy procesory môžu zapisovať dáta do globálnej pamäte. Jednotlivé fázy sú na procesoroch separované tzv. bariérami, ktoré pre každý procesor zaručujú, že neprejde do ďalšej fázy, kým všetky procesory neskončili vykonávanie operácií súčasnej fázy.

LogP model

Iný nástroj na popis efektivity algoritmov ponúka LogP model [Culler93]. Tento model je prispôsobený konkrétnej triede počítačov. Na popis algoritmov používa štyri základné parametre:

L – *latencia*, horné ohraničenie doby potrebnej na doručenie jednej správy

o – *overhead*, doba, počas ktorej procesor spracúva jednu správu, a nemôže vykonávať žiadnu inú operáciu

g – *gap*, najmenší časový interval medzi dvomi po sebe nasledujúcimi správami. Recipročná hodnota 1/g preto zodpovedá šírke pásma pre jeden procesor.

P – *processors*, počet procesorov, prípadne pamäťových modulov

Granularita dát

Granularita dát je rozdelenie dát na menšie bloky, ktoré je následne možné spracovať samostatne, s tým, že následne je zvyčajne potrebné spojiť výsledky výpočtov na jednotlivých blokoch do konečného výsledku. Urýchlenie sa dosiahne vďaka vplyvu pamäti cache – celý blok sa uloží do tejto pamäti, následne je možné pristupovať k údajom rýchlejšie. Vhodnou voľbou granularity dát je možné dosiahnuť výrazné urýchlenie algoritmu. Empirický odhad tohto urýchlenia pre konkrétny algoritmus uvediem ďalej.

Tento postup však nie je možné uplatniť pre každý algoritmus, nakoľko niektoré algoritmy vyžadujú prístup ku všetkým údajom, alebo ich nie je možné rozdeliť na menšie podúlohy.

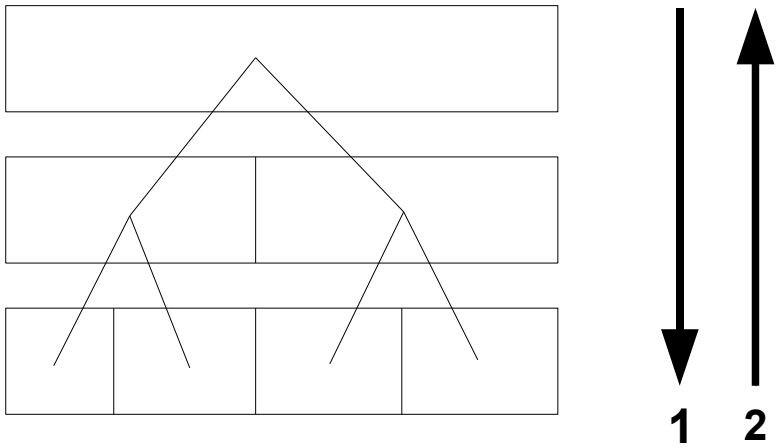
Analýza triedenia zlučovaním

Algoritmus triedenia zlučovaním (merge-sort) má zložitosť O(n log n), kde n je veľkosť vstupných dát. Vzhľadom na malý rozdiel medzi veľkosťou dát a časovou zložitosťou sa dá očakávať, že limitným faktorom pre tento algoritmus bude najmä šírka prenosového pásma.

Implementácia na jednom procesore

Štandardná implementácia merge-sortu vyzerá nasledovne:

- pole sa rozdelí na dve časti
- každá časť sa rekurzívne utriedi
- utriedené polovice poľa sa zlúčia do jedného poľa

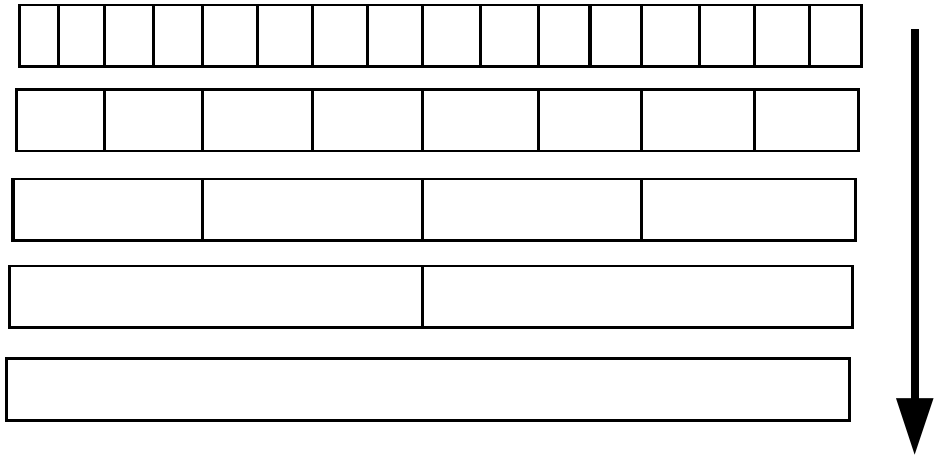


Tento prístup ale pre účely tejto práce nie je vhodný, keďže dochádza k veľkému počtu rekurzívnych volaní funkcií, ktoré trvajú pomerne dlho a skresľovali by výsledky. Preto použijem nasledovnú upravenú verziu algoritmu:

- postupne prechádzam poľom pre jednotlivé veľkosti častí poľa (2, 4, 8, 16, ...) takých, aké by boli použité pri stromovej verzii merge-sortu
- pri každom prechode zlučujem po dvojiciach jednotlivé časti poľa
- používam dve polia identickej dĺžky n, aby som mohol lepšie odhadnúť náročnosť prenosu.

Počas jedného kroku sa dáta prenášajú medzi týmito rovnako dlhými poľami. Po vykonaní i-teho prechodu získam pole pozostávajúce z utriedených úsekov dĺžky 2^i.

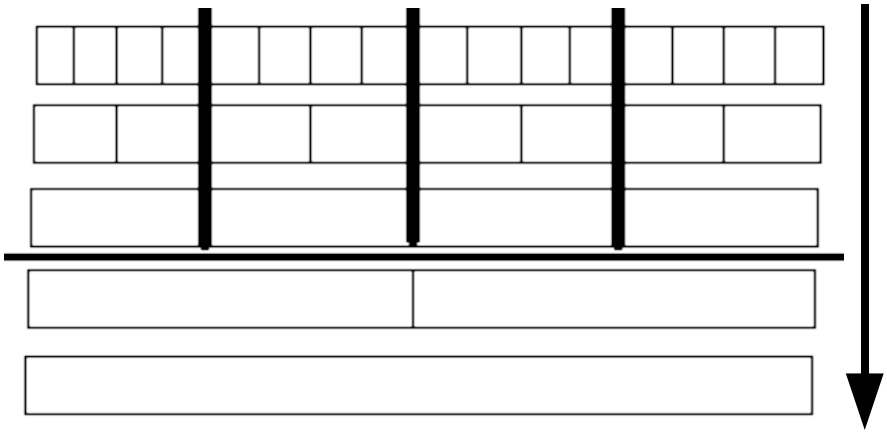
Za predpokladu, 6e n je mocnina 2, $2^{\log n} = n$. Na *log n* prechodov teda utriedim celé pole.



Výhodou tohto prístupu je nielen to, že sa vykonáva menej operácii, a preto sa efektívnejšie využije prenosové pásmo, ale tiež fakt, že pri tomto prístupe je možné lepšie optimalizovať algoritmus tak, aby efektívne využíval hardware počítača - najmä cache pamäť. Pri vhodnom rozdelení dát je možné dosiahnuť významné urýchlenie. Technika, ktorú pri tomto prístupe môžeme použiť, je nasledovná:

- najskôr rozdelím pole do blokov o veľkosti 2^k, kde k volím tak, aby $2^k < n$ (ak 2^k=n, dostanem pôvodný algoritmus)
- každý blok utriedim upraveným merge-sortom
- utriedené bloky následne zlúčim upraveným poľom, pričom začínam od veľkosti časti poľa rovnej veľkosti jedného bloku

Tento algoritmus nazvem *blokový merge-sort*.



Teraz skúsim odhadnúť, k akému urýchleniu dôjde na testovacích architektúrach, a následne výsledky experimentálne overím.

Zložitosť algoritmu je O (n log n), preto celkový čas behu musí byť *t n log n*, kde t je čas potrebný na vykonanie jednej operácie.

Čas potrebný na vykonanie jednej operácie je súčet času potrebného na čítanie záznamu z pamäti a času potrebného na zápis záznamu do pamäti. Tieto operácie sú uskutočňované po dvojiciach, ale každý záznam je čítaný v rámci jedného prechodu len raz - výsledný čas t n log n je teda korektný.

Čas potrebný na zápis jedného záznamu je vždy rovnaký, označme ho t_W.

Čas potrebný na čítanie jedného záznamu už ale nie je vždy rovnaký, nakoľko číslo môžeme čítať buď z hlavnej pamäti RAM, alebo z cache. Prvý čas môžeme označiť t_R, druhý t_C.

K čítaniu z cache dochádza len vo fáze 1 blokového merge-sortu, kedy sa triedia jednotlivé bloky. Konkrétne, ak veľkosť jedného bloku je 2^k, a veľkosť pamäti cache je 2^c, tak potom z celkového počtu log n prechodov poľa sa efekt cache uplatní pri min(c,k)-1 prechodoch. Prechodov nie je min(c,k), pretože pri prvom prechode ešte dáta v cache nie sú a musia sa do nej načítať z RAM. Nasledovné čítania v rámci daného bloku už pamäť cache využívajú.

Celkový čas potrebný na utriedenie poľa o veľkosti n možno preto odhadnúť podľa nasledovného vzťahu:

$$t = n.\log n \,.\, t_W + n.(\min(c,k)-1).t_C + n.(\log n - \min(c,k) + 1).t_R$$

Z teoretického hľadiska vyzerá tento vzťah korektne, praktické merania však ukazujú, že skutočnosť je zložitejšia. Ukazuje sa totiž, že operácie čítania a zápisu nemusia byť od seba oddelené - napríklad som zistil, že čas potrebný na zápis jedného čísla, a čas potrebný na zápis čísla A a následné prečítanie čísla B sú takmer rovnaké. Preto je pri odhadovaní času potrebné aplikovať iné metódy.

Zvolil som metódu, pri ktorej empiricky zmeriam rýchlosť čítania a následného zápisu prvkov poľa, pričom veľkosť poľa volím tak, aby som zmeral rýchlosť čítania z cache i z hlavnej pamäti. Zároveň zahrniem aj dodatočné operácie, ktoré umožnia presnejšie odhadnúť výsledky.

Meranie bez, respektíve s využitím cache som uskutočnil tak, že som čítal a zapisoval buď jedno veľké pole rozsahu väčšieho rozsahu než sa vojde do cache, alebo som veľakrát čítal a zapisoval pole dostatočne malé, aby sa po prvom prečítaní celé uložilo do cache a nasledovné cykly teda čítali údaje z cache pamäti, čím som zistil rýchlosť takéhoto čítania (aj s následným zápisom, keďže takto funguje merge-sort).

Nech rýchlosť čitania bez využitie cache je r_M a rýchlosť s využitím cache je r_C a nech obe sú udané v bajtoch. Na základe týchto nameraných rýchlostí potom môžem odhadnúť čas potrebný na triedenie podľa nasledovného vzťahu:

$$t = n \,.\, (\min(c, k) - 1) \,.\, 1/r_C + n \,.\, (\log n - \min(c, k) + 1) \,.\, r_M$$

Implementácia na viacerých procesoroch

Implementácia pomocou OpenMP je priamočiara. Použije sa rovnaký algoritmus ako pre jeden procesor, s tým, že v každej iterácii sa každý blok rozdelí na dve rovnako veľké časti (keďže výpočet prebieha na dvoch procesoroch), následne každý procesor utriedi svoju časť každého bloku. Procesory sa synchronizujú po každej iterácii. Procesory pristupujú k pamäti po spoločnej zbernici, každý však má vlastnú pamäť cache.

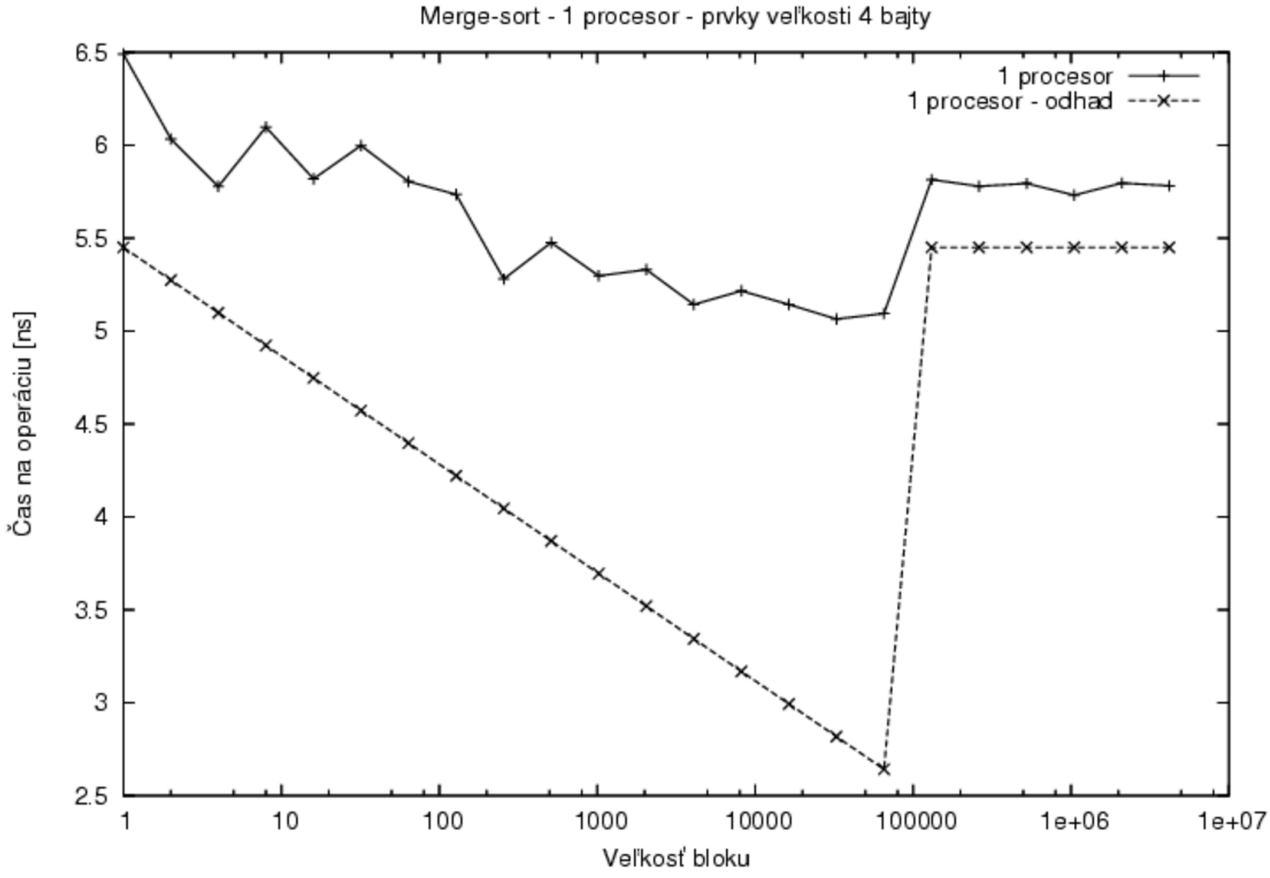
Implementácia pomocou MPI musí vyzerať inak, nakoľko je potrebné prenášať dáta medzi procesormi. Na začiatku sa dáta rozdelia do N blokov, kde N je počet procesorov. Každý procesor potom dostane svoj úsek, ktorý utriedi a pošle späť prvému procesoru. Prvý procesor spojí tieto úseky do výsledného utriedeného poľa.

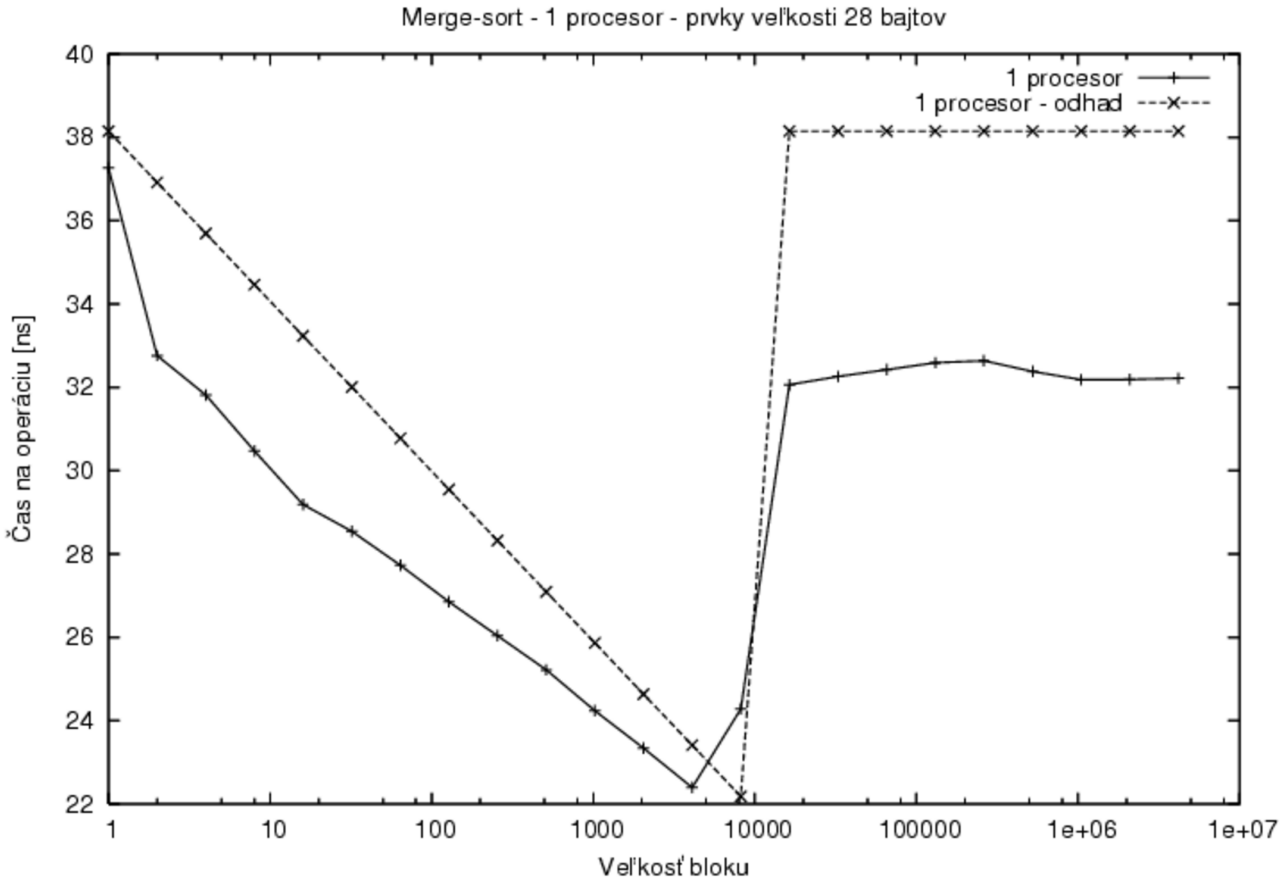
Kritickým miestom je v tomto prípade prenos údajov medzi procesormi. Pokiaľ trvá príliš dlho, nedosiahneme žiadne urýchlenie, môže dokonca dôjsť ku spomaleniu.

Najkratšiu dobu prenosu údajov zistím jednoducho, stačí, keď veľkosť dát v bajtoch vydelím priepustnosťou siete. Skutočná doba prenosu môže byť nižšia, ako bolo uvedené v časti o mikro-benchmarkoch.

Výsledky meraní

Na základe týchto údajov môžem spraviť odhady a porovnať ich s reálne získanými dátami.

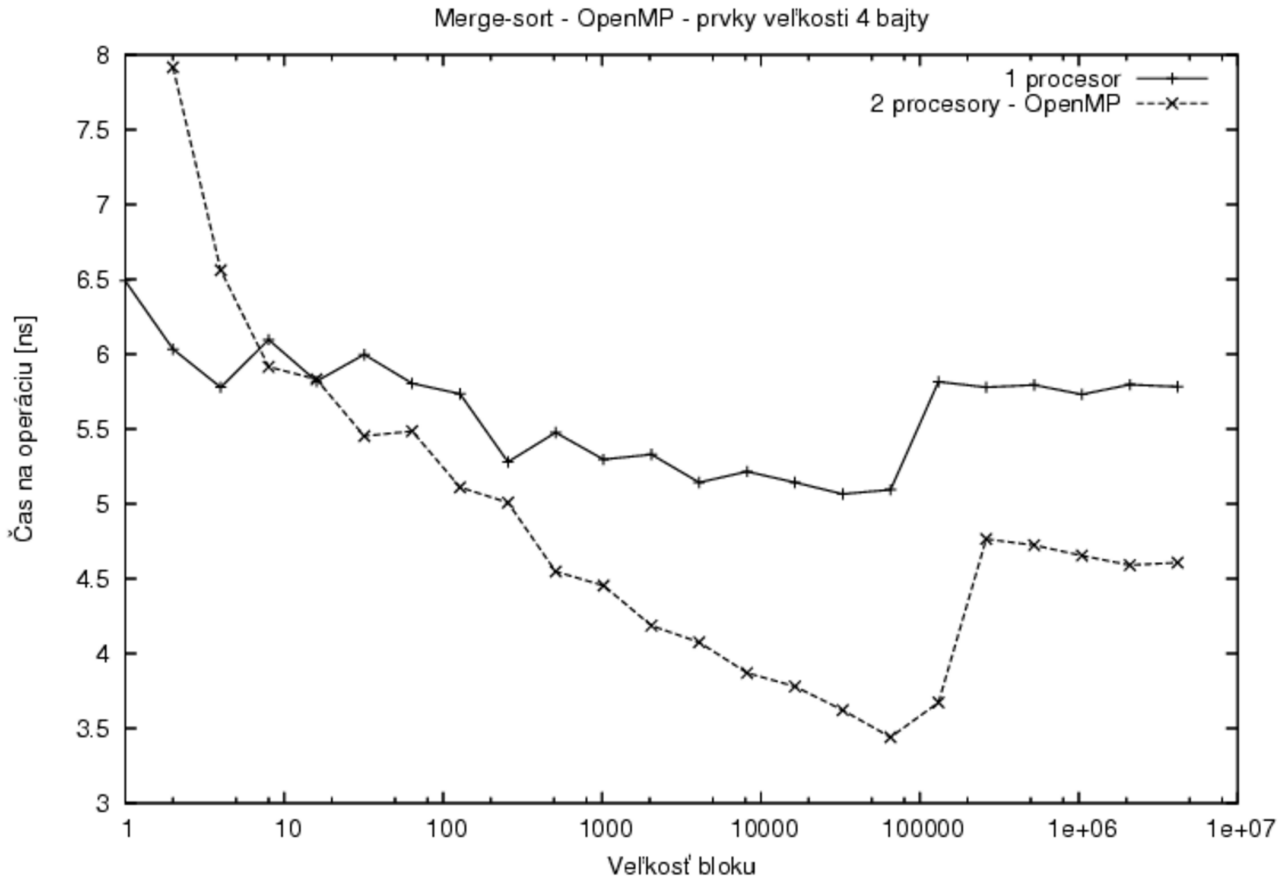


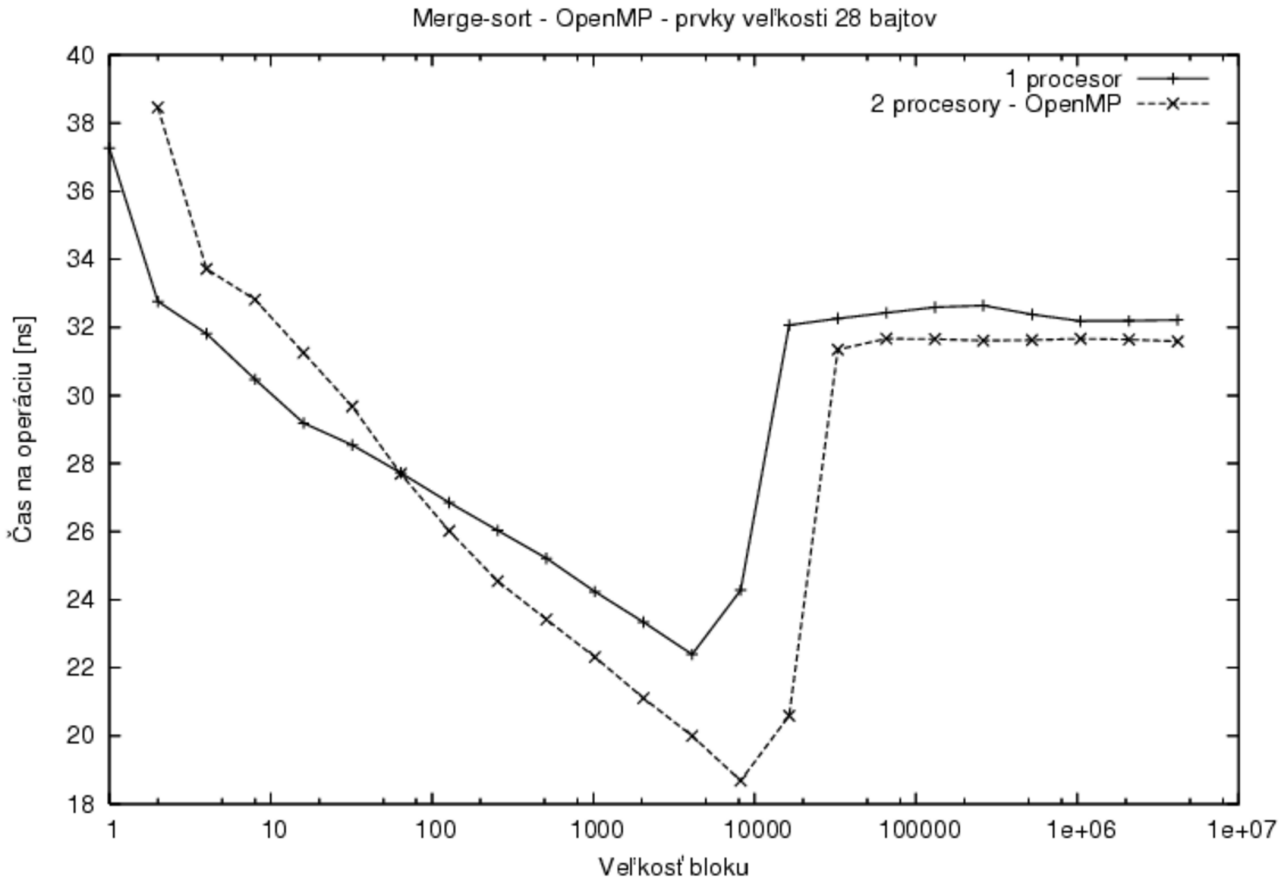


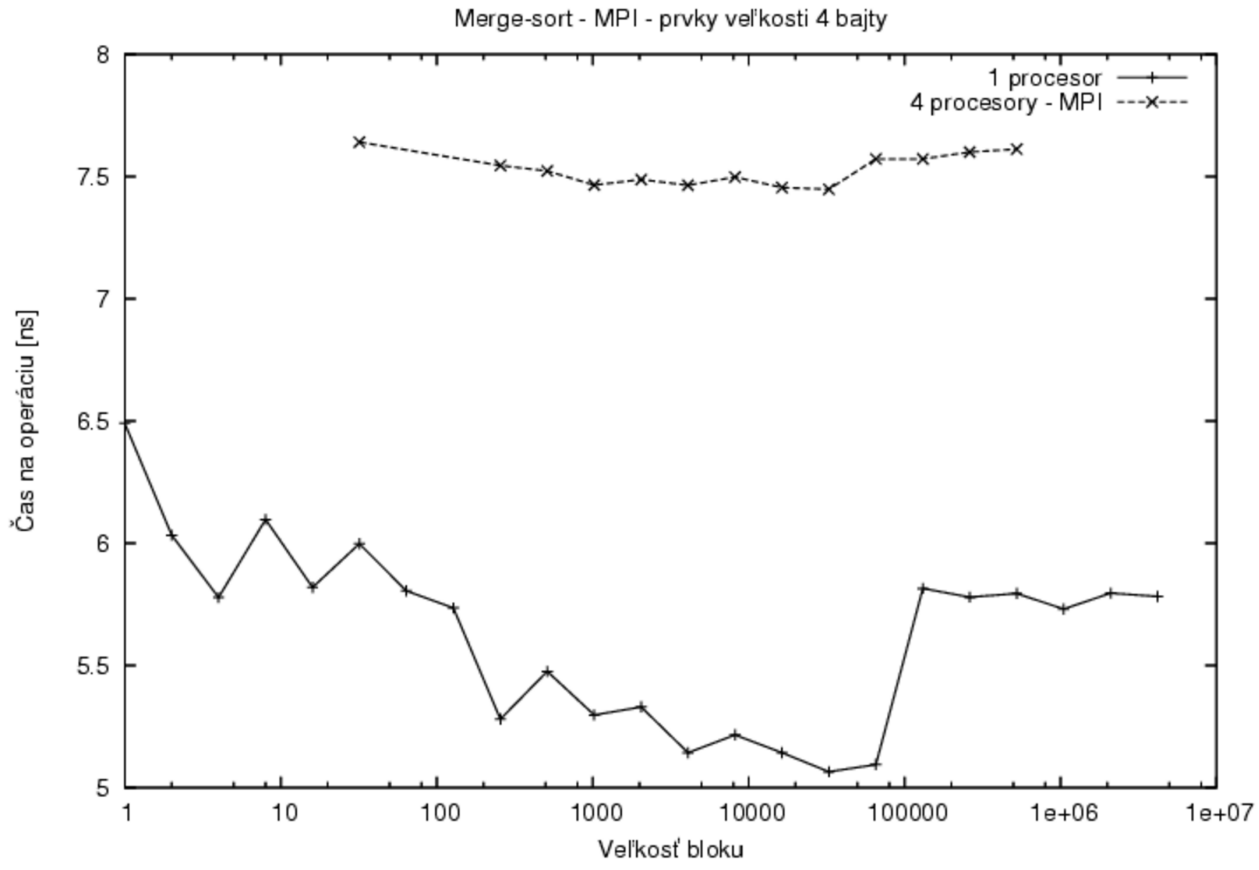
Ako je možné vidieť z grafov, v prípade presunu veľkých blokov sa odhad pre jeden procesor blíži k reálnym výsledkom, v prípade menších prvkov poľa (4 bajty) je odhadované urýchlenie príliš optimistické. Dôvodom je fakt, že reálny algoritmus okrem samotného pristupovania k pamäti vykonáva aj ďalšie operácie (porovnávanie, ...). V prípade väčších blokov sa v každom kroku prenáša väčšie množstvo dát, dominantný je preto vplyv limitnej šírky zbernice, a preto výsledky zodpovedajú odhadom.

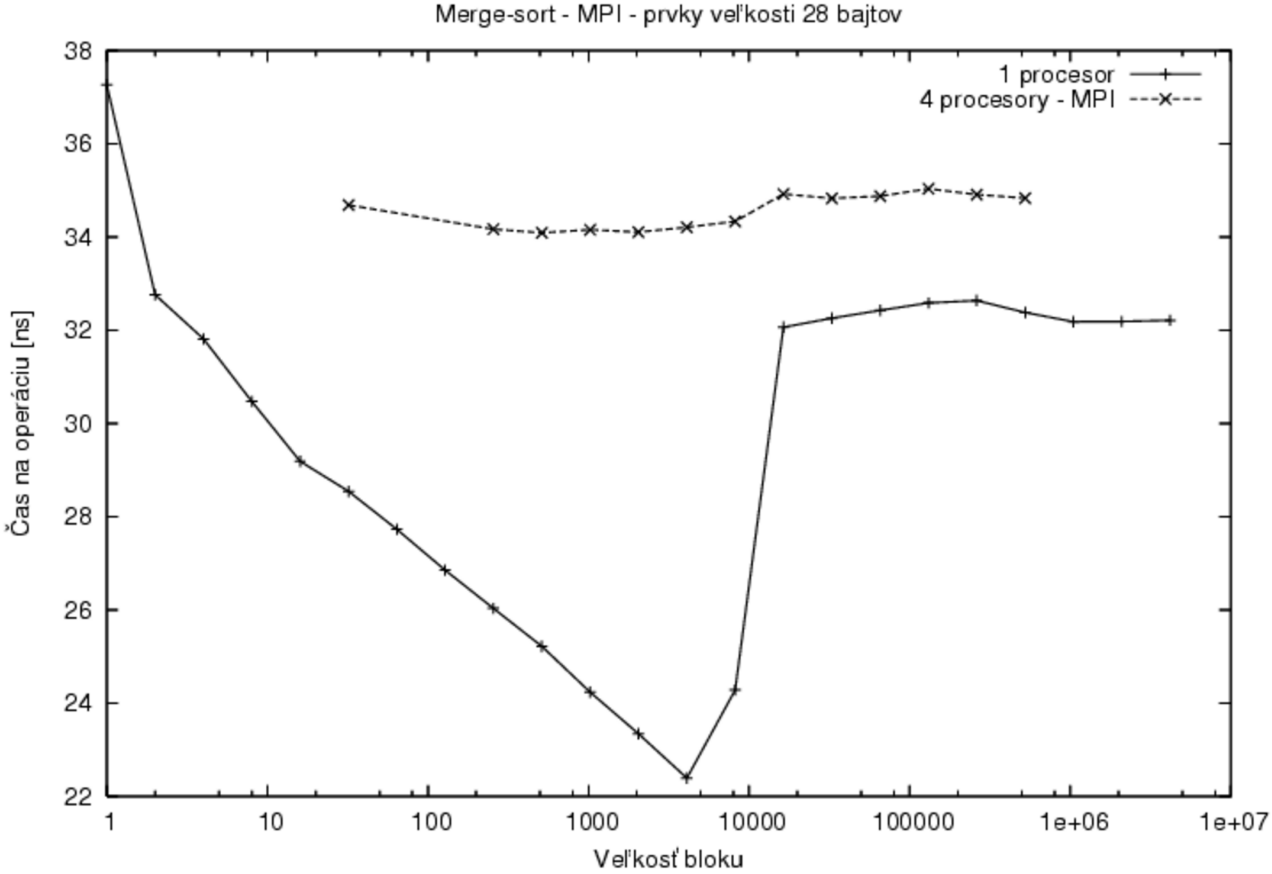
Ukazuje sa, že vhodným využitím efektu cache pamäti je možné urýchliť algoritmus merge-sort (ale asi aj iné algoritmy so zložitosťou n log n) tak, že čas spracovania bude asi 66% pôvodnej hodnoty.

Takéto urýchlenie je možné tiež v prípade architektúry OpenMP, kde nie je potrebné kopírovať dáta medzi procesormi. Pri použití architektúry MPI ale spomalenie spôsobené prenosom dát je natoľko výrazné, že použitie viac procesorov nemá želaný efekt. Oba tieto fakty je možné pozorovať na nasledovných grafoch.


Merge-sort - OpenMP - prvky veľkosti 4 bajty
1 procesor
2 procesory - OpenMP
Čas na operáciu [ns]
Veľkosť bloku
8
7.5
7
6.5
6
5.5
5
4.5
4
3.5
3
1
10
100
1000
10000
100000
1e+06
1e+07


Merge-sort - OpenMP - prvky veľkosti 28 bajtov
1 procesor
2 procesory - OpenMP
Čas na operáciu [ns]
Veľkosť bloku
40
38
36
34
32
30
28
26
24
22
20
18
1
10
100
1000
10000
100000
1e+06
1e+07


Merge-sort - MPI - prvky veľkosti 4 bajty
1 procesor
4 procesory - MPI
Čas na operáciu [ns]
Veľkosť bloku
8
7.5
7
6.5
6
5.5
5
1
10
100
1000
10000
100000
1e+06
1e+07


Merge-sort - MPI - prvky veľkosti 28 bajtov
1 procesor
4 procesory - MPI
Čas na operáciu [ns]
Veľkosť bloku
38
36
34
32
30
28
26
24
22
1
10
100
1000
10000
100000
1e+06
1e+07

Analýza Gaussovej eliminácie

Gaussova eliminácia je algoritmus s časovou zložitosťou O(n^3), pre maticu veľkosti *n x n*. Teda zložitosť vzhľadom na veľkosť vstupu je $O(n^{3/2})$.

Použitý algoritmus pre jeden procesor je popísaný v [Yang2-98].

Algoritmus pozostáva z dvoch fáz – dopredná eliminácia a spätná substitúcia.

Dopredná eliminácia:

for k = 1 **to** n-1
  for i = k+1 **to** n
    T(i,k);
  endfor
endfor

T(i,k) je výkonná časť algoritmu, vyzerá nasledovne:

$a_{i,k} = a_{i,k} / a_{k,k;}$

for j = k+1 **to** n+1
  $a_{i,j} = a_{i,j} - a_{i,k} * a_{k,j};$
endfor

Keďže po eliminácii je dolná trojuholníková matica nulová, algoritmus využíva toto miesto na ukladanie multiplikátorov ($a_{i,k} = a_{i,k} / a_{k,k}$).

Spätná substitúcia:

for i = n **to** 1
  S(i,x);
endfor

Substitučný krok S(i,x) vyzerá nasledovne:

  for j = i+1 **to** n
    $x_i = x_i - a_{i,j} * x_j;$
  endfor
  $x_i = x_i / a_{i,i};$

Pozn.: Vektor x_i je uložený na pozícii $a_{i,n+1}$.

Implementácia na jednom procesore

Na jednom procesore je možné použiť vyššie uvedený algoritmus bezo zmien.

V prípade tohto algoritmu už nie je možné využiť vplyv pamäti cache, nakoľko úlohu nie je možné rozdeliť na menšie podúlohy – celá eliminácia sa musí robiť naraz. Preto sa efekt cache pamäti prejaví len v prípade, ak je veľkosť matice menšia ako veľkosť pamäti cache. V takom prípade by malo dôjsť k takmer dvojnásobnému urýchleniu (nakoľko zápis je aj v tomto prípade realizovaný do pamäti RAM, nielen do cache).

Implementácia na viacerých procesoroch

Na viacerých procesoroch je potrebné algoritmus upraviť tak, aby si jednotlivé procesory navzájom neprepisovali údaje, aby nedochádzalo k zbytočnému opakovaniu častí výpočtu, a aby bola záťaž jednotlivých procesorov približne rovnaká.

Algoritmus upravím tak, že každý riadok matice bude pridelení niektorému procesoru využitím tzv. mapovacej funkcie – táto pre číslo riadku vráti číslo procesora, ktorému je tento riadok pridelený (TODO, odkaz na literatúru kde sa táto technika spomína).

Samotný algoritmus, ktorý pochádza z [Yang2-98], bude potom vyzerať nasledovne:

Dopredná eliminácia:

```
me=mynode();
for i = 1 to n
  if proc_map(i)==me then initialize row i;
endfor
if proc_map(1)==me then
  broadcast row 1;
else
  receive row 1;
for k = 1 to n-1
  for i = k+1 to n
    if proc_map(i)==me then T(i,k);
  endfor
  if proc_map(k+1)==me then
    broadcast row k+1;
  else
    receive row k+1;
endfor
```

Spätná substitúcia:

```
for i = n to 1
  if id==0 then
    receive row i
    S(i,x);
  else
    send row i to processor 0
endfor
```

Spätnú substitúciu vykonáva teda prvý procesor, ostatné procesory v tomto kroku len posielajú svoje dáta.

Odhady

Na jednom procesore sa dá očakávať, že doba potrebná na vykonanie jednej operácie bude pre rôzne veľkosti matice rovnaká, za predpokladu, že veľkosť matice je väčšia než veľkosť pamäti cache.

Počet operácií, ktoré vykoná algoritmus, je možné jednoducho určiť z algoritmu. Vzťah vyzerá nasledovne:

$$\frac{n(n-1)}{2} + \sum_{i=0}^{n-1} (i(i+1))$$

Zároveň sa ukazuje, že, podobne ako v prípade algoritmu merge-sort, jediný spôsob, ako vykonať rozumný odhad doby spracovania, spočíva vo využití micro-benchmarkov. Výsledný odhad doby spracovania potom získam jednoducho tak, že vynásobím počet operácií časom potrebným na vykonanie jednej operácie. Jedna operácia zahŕňa 3 čítania a jeden zápis vo fáze doprednej eliminácie, 2 čítania a jeden zápis vo fáze spätnej substitúcie. Nakoľko zložitosť spätnej substitúcie je len $O(n^2)$, môžeme pri odhadovaní predpokladať, že čas na vykonanie jednej operácie je v oboch prípadoch rovnaký.

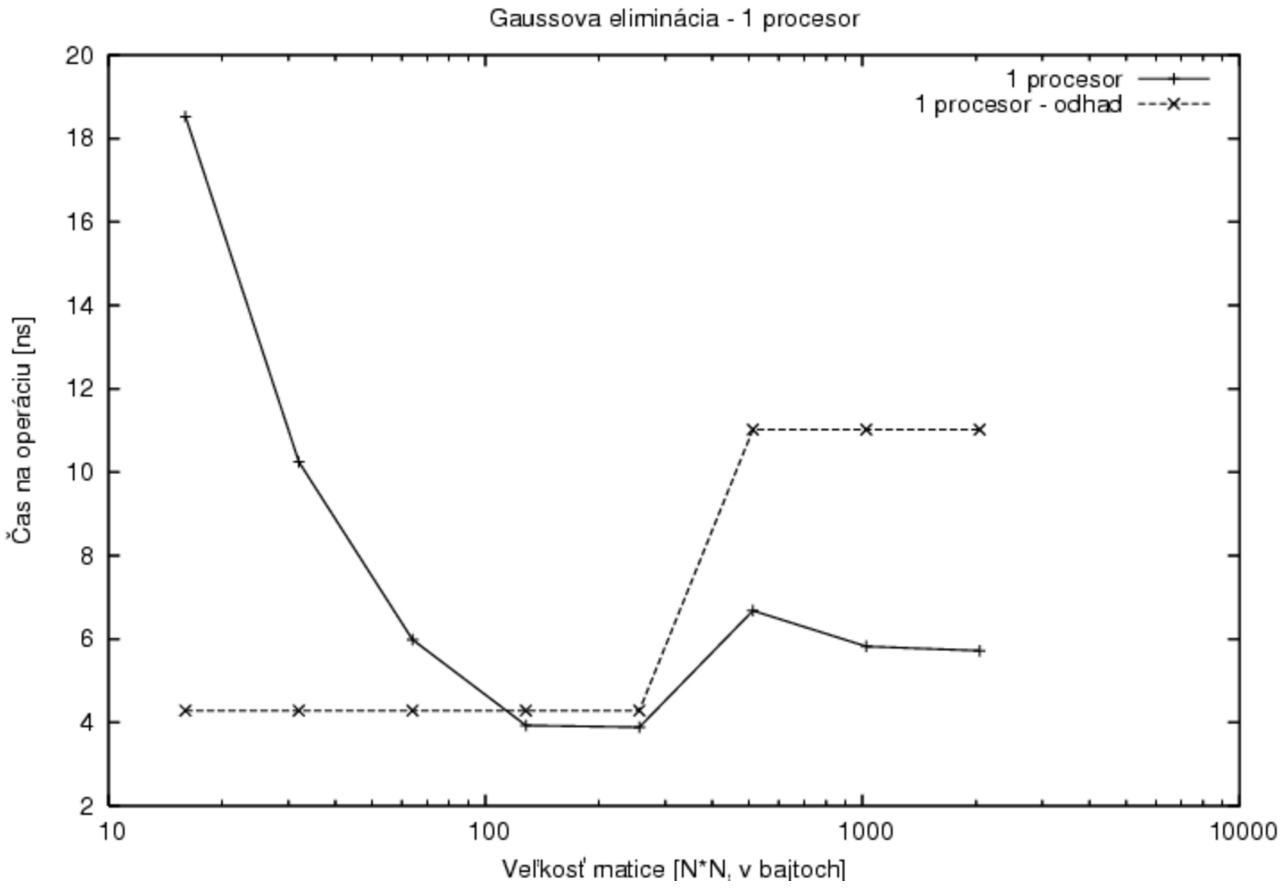
Tieto vzťahy sa dajú jednoducho odvodiť z algoritmu analyzovaním počtu čítaní a zápisov, napríklad operácia $x_i = x_i / a_{i,i}$; vyžaduje dve čítania a jeden zápis.

Tento odhad je možné upresniť s využitím faktu, že použitá architektúra dokáže čiastočne paralelizovať čítanie kombinované so zápisom, ako vyplýva s micro-výsledkov benchmarkov.

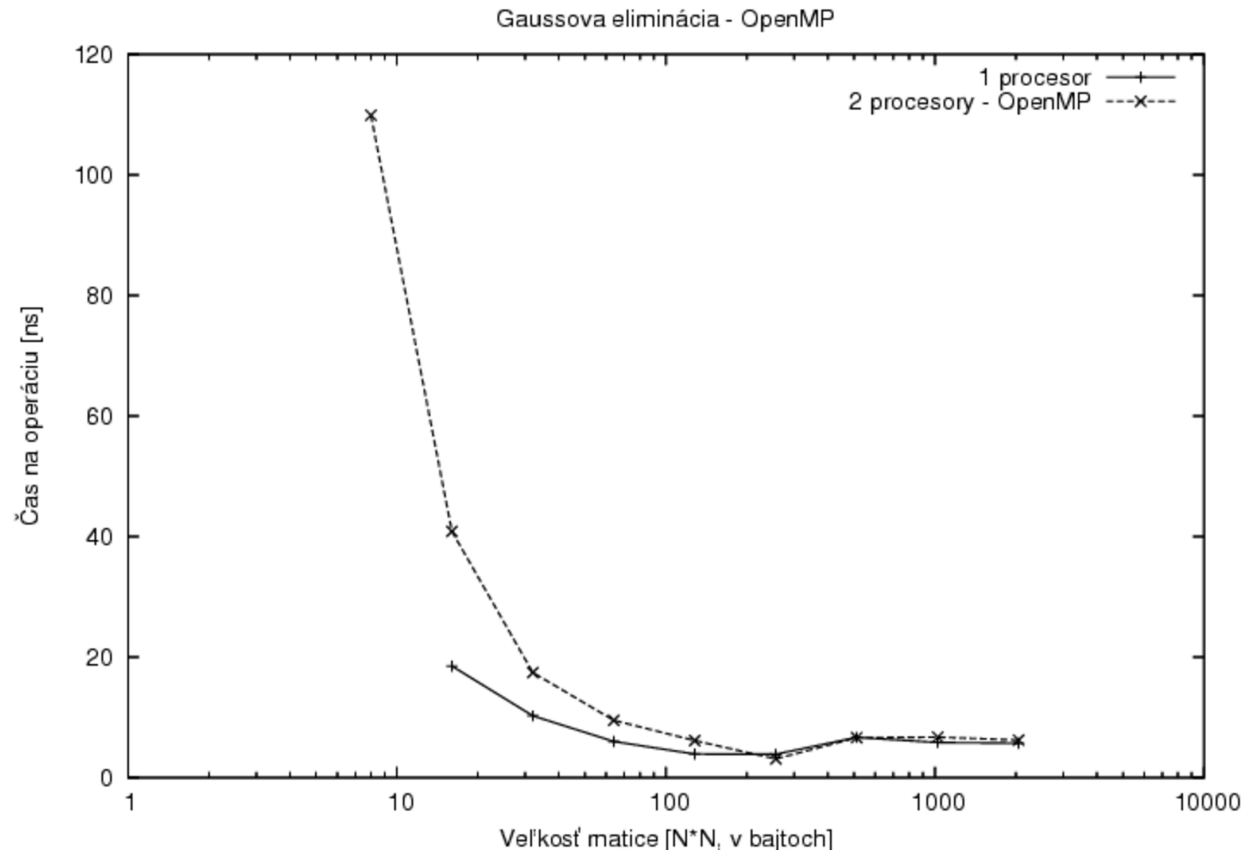
Verzia **OpenMP** vyzerá podobne ako verzia pre MPI, hlavným rozdielom je ale fakt, že nie je potrebné počas výpočtu prenášať dáta medzi procesormi, nakoľko všetky majú prístup k spoločnej pamäti. Preto sa dá očakávať, že prenos po spoločnej zbernici bude v tomto prípade limitným faktorom, a teda že použitie dvoch procesorov nebude mať na čas spracovania žiadny výraznejší vplyv – mala by zostať rovnaká ako v prípade spracovania na jednom procesore.

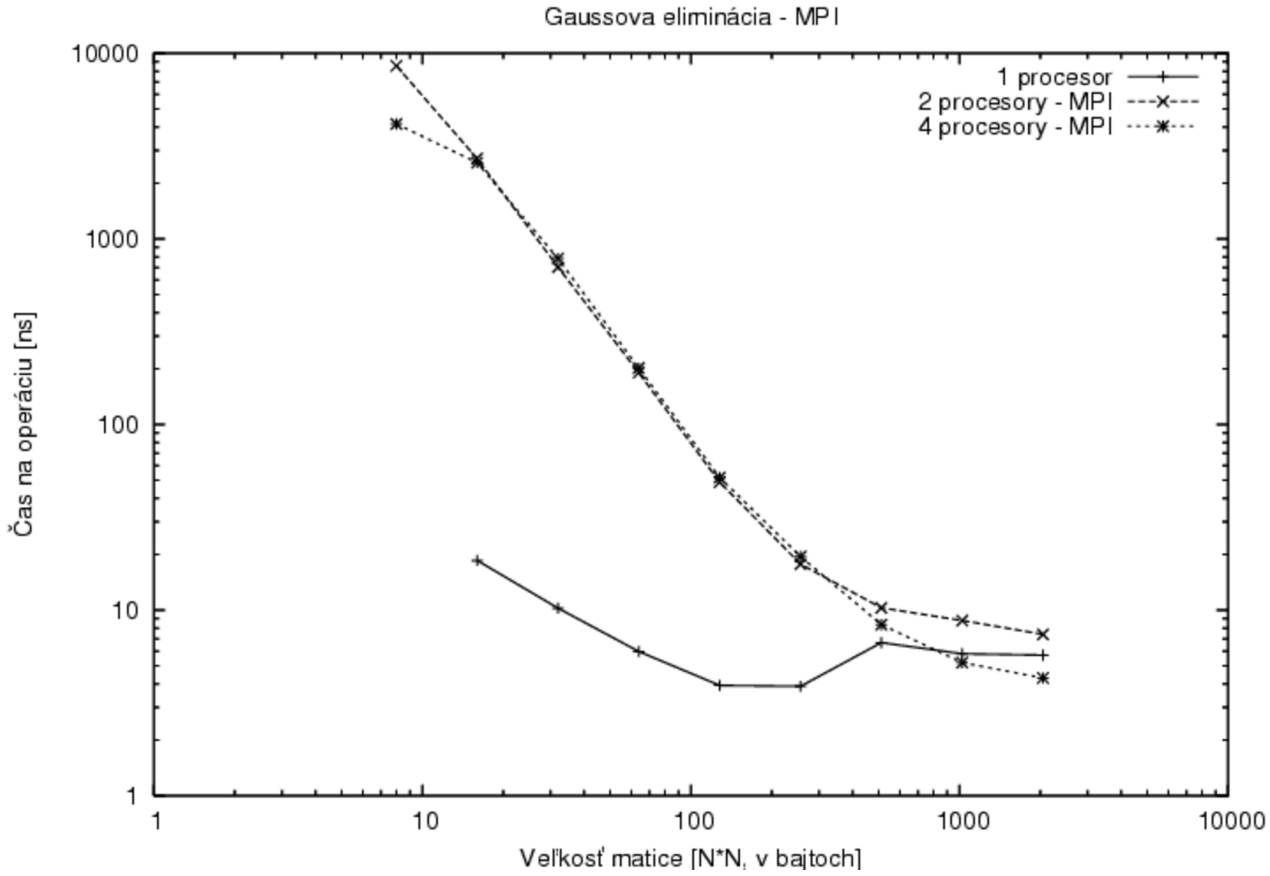
Verzia **MPI** vyžaduje prenos dát medzi procesormi po sieťovom rozhraní. Dáta sa prenášajú po riadkoch. Pre každý riadok *i* platí, že procesor, ktorému je riadok *i* pridelený, posiela tento riadok všetkým ostatným procesorom – ide o všesmerové vysielanie (broadcast). Samotné spracovanie sa pre každý riadok uskutočňuje vždy len na jednom procesore. Dá sa preto očakávať, že pre dostatočne veľké matice bude MPI verzia rýchlejšia než jednoprocesorová verzia – to samozrejme závisí hlavne od rýchlosti a latencii pamäte a siete.

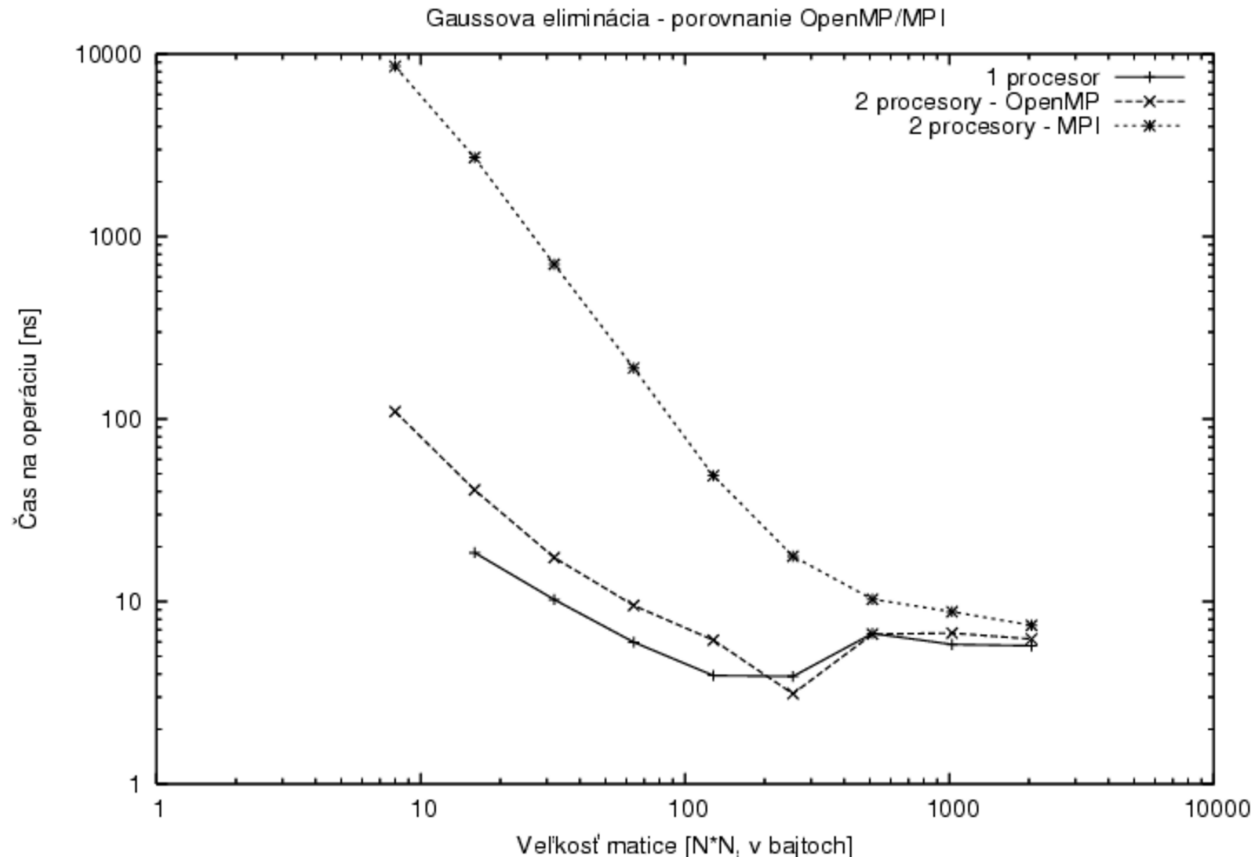
Výsledky

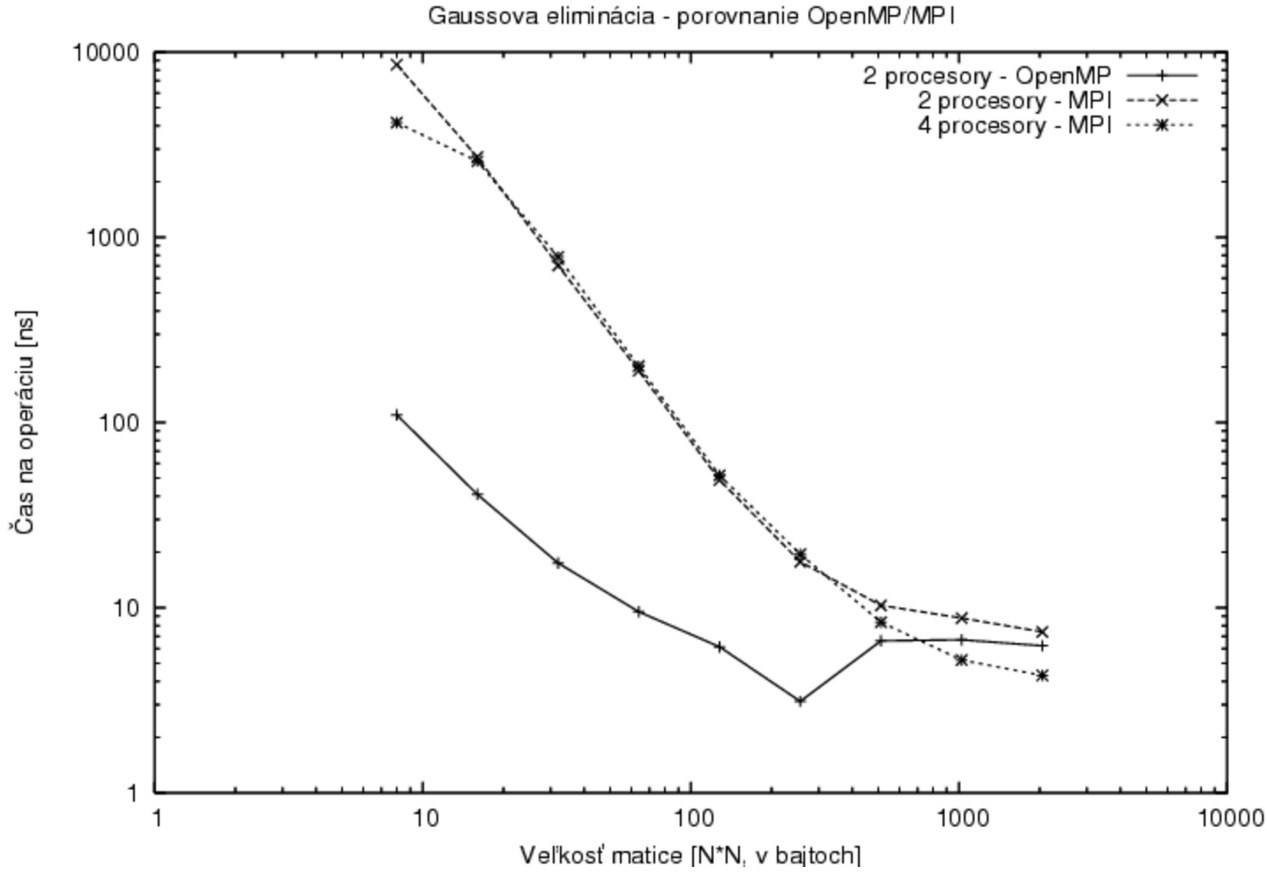


Z grafu je vidieť, že odhad sa najlepšie zhoduje s realitou v prípade, kedy je veľkosť matice porovnateľná s veľkosťou cache pamäti. Pre veľmi malé matice je čas na operáciu nižší v dôsledku toho, že vplyv iných častí programu (skoky a pod.) je príliš výrazný. Pre veľké matice sa ukazuje, že výpočet prebieha rýchlejšie, než sa očakáva podľa benchmarkov. To je spôsobené pravdepodobne tým, že algoritmus je schopný aspoň čiastočne využiť efekt cache pamäti, hoci tento efekt nie je natoľko výrazný ako v prípade algoritmu triedenia zlučovaním.


Gaussova eliminácia - OpenMP
1 procesor
2 procesory - OpenMP
Čas na operáciu [ns]
Veľkosť matice [N*N, v bajtoch]
0
20
40
60
80
100
120
1
10
100
1000
10000


Gaussova eliminácia - MPI
1 procesor
2 procesory - MPI
4 procesory - MPI
Čas na operáciu [ns]
Veľkosť matice [N*N, v bajtoch]
1
10
100
1000
10000


Gaussova eliminácia - porovnanie OpenMP/MPI
1 procesor
2 procesory - OpenMP
2 procesory - MPI
Čas na operáciu [ns]
Veľkosť matice [N*N, v bajtoch]
10000
1000
100
10
1
1
10
100
1000
10000


Gaussova eliminácia - porovnanie OpenMP/MPI
2 procesory - OpenMP
2 procesory - MPI
4 procesory - MPI
Čas na operáciu [ns]
Veľkosť matice [N*N, v bajtoch]
10000
1000
100
10
1
1
10
100
1000
10000

Implikácie práce pre implementáciu algoritmov

Architektúra súčasných počítačov sa vyznačuje vysokou zložitosťou a jej návrh je historicky určený na jednej strane inžinierskymi a fyzikálnymi obmedzeniami, na druhej strane procesom optimalizácie na triedu najrozšírenejších používaných programov a aplikácií.

Výber konkrétnej implementácie preto závisí na viacerých faktoroch.

Na základe testovaných algoritmov je možné dospieť k nasledovným záverom:

- Pokiaľ je možné vhodným spôsobom využiť granularitu dát tak, aby algoritmus využil výhody rýchlej pamäti cache, je možné dosiahnuť až takmer dvojnásobné urýchlenie algoritmu na jednom procesore.
- Pokiaľ algoritmus vyžaduje v každom kroku prístup k veľkej časti údajov, možnosti využitia pamäti cache sú veľmi obmedzené. V takomto prípade zvyčajne nie je možné týmto spôsobom algoritmus urýchliť.
- Pokiaľ výpočtová náročnosť algoritmu dostatočne presiahne komunikačnú zložitosť, je možné algoritmus urýchliť s využitím paralelného spracovania. Z algoritmov študovaných v tejto práci, triedenie zlučovaním má časovú zložitosť $O(n \log n)$ a komunikačnú zložitosť $O(n)$, líšia sa teda koeficientom $\log n$. Gaussova eliminácia má časovú zložitosť $O(n^3)$ a komunikačnú zložitosť $O(n^2)$, teda sa líšia koeficientom $n^{3/2}$. Prvý z algoritmov kvôli tomu nie je možné urýchliť využitím paralelizmu, kým v prípade druhého z nich to možné je, avšak len pri dostatočne veľkom objeme dát (pre študovanú architektúru musí byť veľkosť matice aspoň 1024x1024, pre iné architektúry, resp. počty procesorov, môže byť táto hranica pri inej ve+lkosti vstupu).
- Pokiaľ výpočtová náročnosť algoritmu výrazne presiahne komunikačnú zložitosť, algoritmus je zvyčajne možné paralelizovať bez problémov s komunikačnou zlozitosťou. Takéto algoritmy pre účely tejto práce nie sú zaujímavé. Niektoré problémy z tejto triedy sú popísané napríklad v [Yang1-98].

Zoznam použitej literatúry

ICC: Intel C++ 8.1, *http://www.intel.com/software/products/compilers/*

Parello02: Parello,D., Temam,O., Verdun,J.-M.: On Increasing Architecture Awareness in Program Optimizations to Bridge the Gap between Peak and Sustained Processor Performance, 2002

Yang1-98: Yang,T.: Model-based Programming Methods, 1998

OPENMP: Domovská stránka OpenMP, *http://www.openmp.org/*

LAMMPI: Domovská stránka pre LAM-MPI, *http://www.lam-mpi.org*

Quammen01: Quammen,C.: Introduction to Programming Shared-Memory and Distributed-Memory Parallel Computers, 2001

Valiant90: Valiant,L.: A Bridging Model for Parallel Computation, 1990

Culler93: David Culler a kol.: LogP: Towards a Realistic Model of Parallel Computation, 1993

Yang2-98: Yang,T.: Gaussian Elimination for Solving Linear Systems, 1998

Obsah