Implementace paralelních algoritmů pomocí knihovny OpenMP

Implementation of parallel algorithms by using OpenMP

Bc. Ondrej Podolinský

Diplomová práce
2008

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav aplikované informatiky
akademický rok: 2007/2008

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Ondrej PODOLINSKÝ**
Studijní program:	**N 3902 Inženýrská informatika**
Studijní obor:	**Informační technologie**
Téma práce:	**Implementace paralelních algoritmů pomocí knihovny OpenMP.**

Zásady pro vypracování:

1. **Vytvořte literární rešerši popisující:**
   - vlastnosti základních typů paralelních systémů (Flynnova taxonomie)
   - vlastnost PRAM modelů a jejich vzájemné vztahy, vztah k SMP a možnosti simulací na jiných než SMP systémech
   - základní metody paralelizace sekvenčních úloh
   - problémy a překážky vznikající při paralelizaci úloh na výpočetních systémech
   - vlastnosti a možnosti jazyk OpenMP při paralelizaci výpočetních úloh a jiných algoritmů na SMP systémech.
2. **V praktické části dokumentu popište:**
   - základní paralelizační techniky, včetně analýzy časové a paměťové složitosti a implementace příkladů pomocí jazykce C/C++ a knihovny OpenMP
   - paralelizaci základních výpočetních a datových algoritmů, včetně analýzy časové a paměťové složitosti a implementace v jazyce C/C++ s pomocí knihovny OpenMP.
3. **Proveďte měření reálné doby běhu paralelizovaných algoritmů pro 1– 8 procesorů a výsledky popište a zpracujte ve formě přehledných tabulek. Komentujte dosažené výsledky.**

Rozsah práce:

Rozsah příloh:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

1. **Brian W. Kernighan, Dennis M. Ritchie, Programovací jazyk C, Computer Press, 2006, ISBN 80-251-0897-X.**
2. **Stephen Prata, Mistrovství v C++, 3. vydání, Computer Press, 2007, ISBN 978-80-251-1749-1.**
3. **Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., Menon, R., Parallel Programming in OpenMP, Morgan Kaufmann Publishersm, 2001, ISBN 1-55860-671-8.**
4. **Prof. P. Tvrdík, Paralelní systémy a algoritmy, Nakladatelství ČVUT, 2006, ISBN 80-01-03565-4.**
5. **J. Keller, Ch. W. Keßler, J. L. Träff, Practical PRAM Programming, John Wiley & Sons, Inc., 2001, ISBN 0-471-35351-5.**
6. **Jesse Liberty, Naučte se C++ za 21 dní, Computer Press, 2002, ISBN 80-7226-774-4.**
7. **MSDN:OpenMP in Visual C++ [online]. Dostupný z WWW: http://msdn2.microsoft.com/en-us/library/tt15eb9t.aspx**
8. **OpenMP [online]. Dostupný z WWW: http://www.openmp.org/blog/**
9. **OpenMP [online]. Dostupný z WWW: https://computing.llnl.gov/tutorials/openMP/**

Vedoucí diplomové práce: **Ing. Michal Bližňák**
Ústav aplikované informatiky

Datum zadání diplomové práce: **20. února 2008**

Termín odevzdání diplomové práce: **19. května 2008**

Ve Zlíně dne 20. února 2008

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
L.S.

prof. Ing. Vladimír Vašek, CSc.
děkan

doc. Ing. Ivan Zelinka, Ph.D.
ředitel ústavu

ABSTRAKT

Obsahem této práce je seznámení se s principy paralelního programování, jeho základních rozdělení a implementace paralelních metod do stávajících programů v programovacím jazyce C resp. C++.

Klíčová slova: openmp, c, c++, vícevláknové programování, paralelní algoritmy

ABSTRACT

The aim of this thesis is to acknowledge with principles of parallel programming, its basic divisions and implementations of parallel methods into already existing programs in C or C++ programming language.

Keywords: openmp, c, c++, multithreads programming, parallel algorithms

Poděkování:

Děkuji tímto svému vedoucímu práce Ing. Michalu Bližňákovi, Ph.D., který měl se mnou občas více trpělivosti, než by bylo zdrávo a vždy měl nějakou vtipnou poznámku nebo smajlík po ruce, když jsem s ním něco řešil.

Rovněž musím, a to především, poděkovat svojí mamince, která na mě nenásilně tlačila a kopala do učení celý život, protože hlavně díky ní teď mohu psát toto poděkování. Děkuji bratrovi, který mi byl skvělým duchovním rádcem, děkuji babičce, že mě finančně celé studia podporovala a také děkuji všem spolužákům a učitelům, kteří mne formovali celá má studia na Univerzitě Tomáše Bati ve Zlíně.

Motto:

Člověk je nedokonalý, ale dokonale si s tím poradí. (Leszek Kumor)

Prohlašuji, že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků, je-li to uvolněno na základě licenční smlouvy, budu uveden jako spoluautor.

V Uh. Hradišti, 1.9.2008

…….

Bc. Ondrej Podolinský

OBSAH

ÚVOD

V současnosti má již skoro každý nový procesor v osobním PC či notebooku více než jedno jádro, ať už se jedná o procesory Intel či AMD. To sebou nese dvě hlavní výhody vyplývající z počtu jader většího než 1 – každé jádro v procesoru může zpracovávat jednu aplikaci, čímž se zvyšuje počet obsloužených aplikací v jeden okamžik na počet, který je roven počtu jader.

Druhým způsobem je vytvářet aplikace jako vícevláknové (výpočetní část programu je rozložena mezi více jader), čímž je zabezpečeno rovnoměrnější vytížení všech jader po většinu doby běhu aplikace, což má za následek někdy větší, někdy menší zvýšení výkonnosti aplikace.

I. TEORETICKÁ ČÁST

1 CO JSOU TO PARALELNÍ VÝPOČTY

Běžnou a zavedenou praxí mezi drtivou většinou programátorů, je psaní programů sekvenčně tzn. každý program je tvořen určitou posloupností příkazů, které se provádějí v určeném pořadí – v danou chvíli může daný procesor resp. jádro zpracovat pouze jednu instrukci, na kterou navazuje přímo či nepřímo celý zbytek programu.

Paralelní výpočet je forma výpočtu, kdy dochází k mnohonásobnému provádění určité operace současně a to tím způsobem, že se dílčí problém rozdělí na několik menších, které jsou pak následně řešeny současně, probíhají tedy paralelně.

Existuje několik technik paralelizace: paralelismus na bitové úrovni (bit-level parallelism), paralelismus na programové úrovni (instruction-level parallelism), datový paralelismus (data paralelism) a funkční paralelismus (task parallelism).

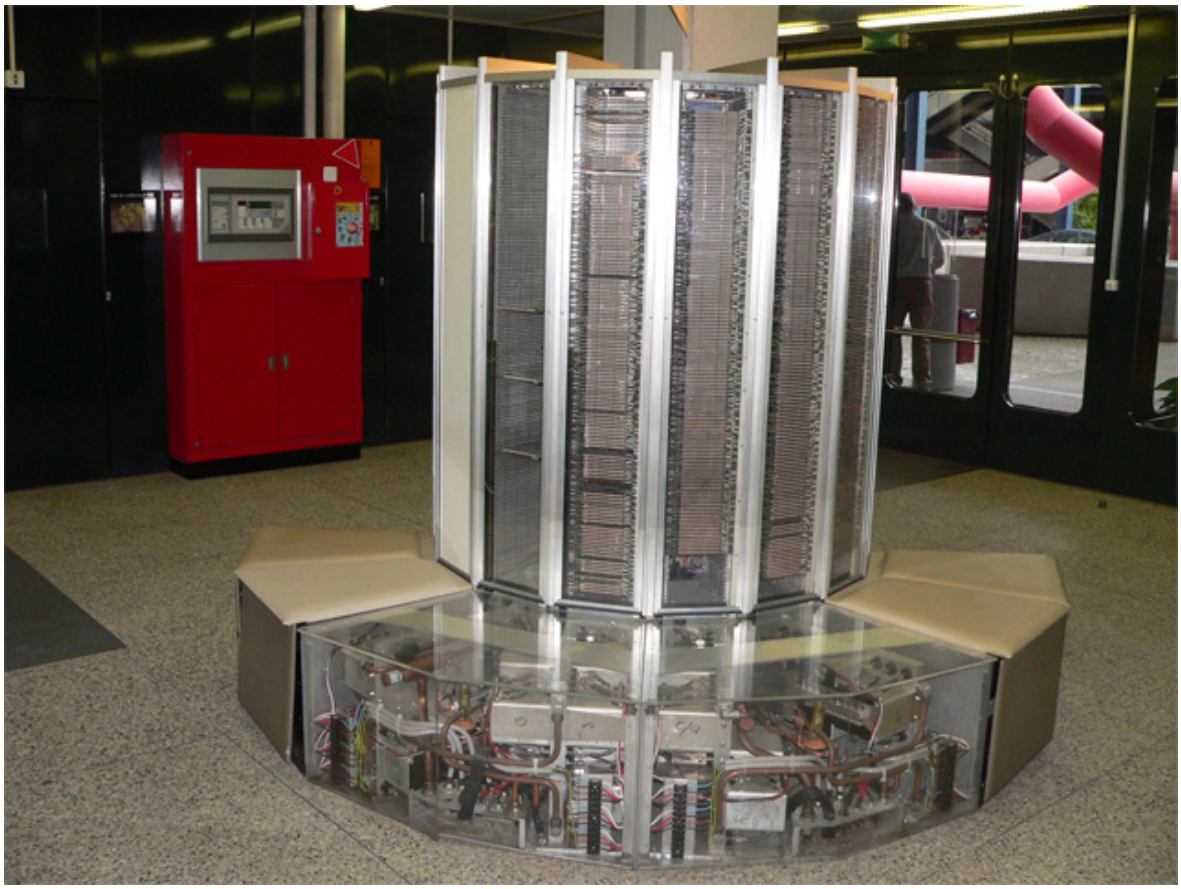
Napsat paralelní program je obtížnější než standardní program, tzn. sekvenční, což je způsobeno nutností synchronizace a komunikace mezi jednotlivými částmi úloh. Tyto překážky ve výsledku značně snižují výkonnost paralelního programu oproti jeho sekvenčnímu ekvivalentu.

2 HISTORIE PARALELNÍCH POČÍTAČŮ

První paralelní počítače vznikly na přelomu padesátých a šedesátých let 20. století v laboratořích IBM, který je do dnešních dnů významným hráčem v oblasti informačních technologií. Obsahovaly na tu dobu neskutečné 4 procesory, které sloužily pro matematické výpočty.

Zde je nutné podotknout, že vývoj superpočítačů té doby byl tlačen dopředu armádou a potažmo tedy napjatou atmosférou studené války.

Koncem 60. let byl zvýšen počet procesorů na osm, avšak cesta k prvnímu paralelnímu superpočítači, který měl pak díky svému úspěchu několik dalších následovníků, byl počítač Cray.



Obrázek 1 Vektorový paralelní superpočítač Cray 1 [11]

Cray patřil do rodiny vektorových počítačů (viz. Kapitola 3.4.4), které byli na poli paralelních počítačů dominantní v 70. a 80. letech. V této době se začal prosazovat i nový druh paralelních počítačů, který obsahoval až stovky procesorů, pro které se začal používat termín masivně paralelní počítače.

Tato skupina počítačů je hojně používaná do dnešní doby - například mnohaprocesorový masivně paralelní počítač Blue Gene patří v dnešní době (léto 2008) k nejvýkonnějším počítačům na světě.

Obrázek 2 Masivně paralelní počítač Blue Gene [11]

Od devadesátých let 20. Století došlo k rozmachu paralelizačních technik a také k rozvoji různých principů paralelizace, co se týče sdílení dat mezi jednotlivými výpočetními procesory či uzly, proto bude následovat rozdělení podle jednotlivých skupin počítačů. [11]

3 DRUHY PARALELNÍCH POČÍTAČŮ

Paralelní počítače mohou být klasifikovány podle úrovně, na které hardware podporuje paralelismus. Tato klasifikace je obdobou vzdálenosti mezi základními výpočetními uzly. Jednotlivé druhy se vzájemně nevylučují a mohou se i prolínat. Například svazky počítačů, tzv. clustery ze symetrických multiprocesorů jsou poměrně časté.

3.1 Vícejádrové procesory

Vícejádrovým procesorem je označován procesor, který obsahuje více výpočetních jednotek (jader). Je to pro řadového uživatele zdaleka nejběžnější a asi také jediný typ paralelního počítače, se kterým se může setkat – jeden procesor obsahuje několik výpočetních jednotek (2,4), které se chovají jako samostatné procesory a mohou zpracovávat instrukce odděleně. [11]

3.2 Symetrické víceprocesorové systémy

Jsou to systémy o více totožných procesorech, které jsou přes společnou sběrnici připojeny ke sdílené paměti. Sběrnicové zapojení zabraňuje ve větší škálovatelnosti, proto se používá u těchto systémů do 32 procesorů. [11]

3.3 Distribuované výpočetní systémy

Těmito systémy se myslí výpočetní jednotky propojené mezi sebou při výpočtech do sítě. Tyto systémy jsou vysoce škálovatelné. [11]

3.3.1 Clusterové výpočetní systémy

V dnešní době tvoří největší procento výkonných paralelních počítačů stroje, založené na clusterovém výpočetním systému. Cluster je skupina počítačů, úzce svázaných pomocí technologií vycházejících z lokálních sítí. I přes to, že jednotlivé počítače mohou být samostatné pracovní stanice a jsou vybaveny standardními operačními systémy, mohou být clustery považovány v některých pohledech za jeden počítač. I přes to, že počítače zapojené do clusteru nemusí být symetrické a mít potažmo stejný výpočetní potenciál, je to kvůli mnohem snazšímu vyvážení zátěže na výpočty doporučeno. [11]

Nejběžnějším typem clusteru je Beowulf cluster, což je systém založený na propojení běžně prodávaných PC pomocí lokální sítě a protokolu TCP/IP. [11]

Obrázek 3 Cluster Beowulf [11]

3.3.2 Masivně paralelní výpočetní systémy

Pod tímto systémem si může čtenář představit jeden velký počítač, který obsahuje mnoho stejných procesorů. Masivně paralelní výpočetní systémy mají mnoho stejných znaků jako clustery, avšak velmi často obsahují obrovské množství procesorů – např. IBM Blue Gene 212992 procesorů. Každý procesor má vlastní paměť, kopii operačního systému a aplikace. Jednotlivé subsystémy mezi sebou komunikují díky vysokorychlostnímu vnitřnímu zapojení. Příkladem tohoto systému může být např. Masivně paralelní počítač Blue Gene. [11]

3.3.3 Výpočetní systémy GRID

GRID jsou nejvíce distribuovanou formou systémů pro paralelní výpočty. Pro výpočty využívá počítače komunikující přes síť internet. Jelikož je internet komunikační médium s relativně vysokou latencí spojení a nízkými přenosovými rychlostmi, hodí se tyto systémy pouze na určitý druh výpočtů. Známé projekty, které byly za pomocí systému GRID realizovány, jsou SETI@Home zabývající se dekódováním signálů přijatých z vesmíru a hledáním mimozemské inteligence a Folding@Home.

Tyto systémy většinou potřebují pro svůj běh určitý program tzv. middleware, který se nahraje na počítač určený pro dané výpočty. Tato aplikace si průběžně stahuje i odesílá všechna relevantní data a spouští se při nečinnosti PC, například se může spustit při přechodu počítače do úsporného režimu typu spořič obrazovky. [11]

3.4 Specializované paralelní počítače

Krom výše uvedených ještě existují speciální typy paralelních počítačů, které však mají omezené možnosti využití. [11]

3.4.1 FPGA

Překonfigurovatelné výpočetní systémy využívají programovatelná hradlová pole (FPGA), což je zjednodušeně řečeno počítačový čip, který je schopný se modifikovat dle zadaného požadavku.

FPGA bývá programováno v hardwarově popisném jazyku jako je VHDL či Verilog. [11]

3.4.2 GPU

Grafický čip na akceleraci a vykreslování grafiky má v dnešní době takový výkon, že dostal označení GPU, neboli Graphics Processsing Unit. Tyto procesory jsou navrženy a optimalizovány pro práci s počítačovou grafikou, takže jejich využití v paralelních systémech se týká hlavně operací s maticemi v lineární algebře. Aplikace využívající GPU jsou využívány v oblasti medicíny a techniky (Matlab). [11]

3.4.3 Aplikací specifikované integrované obvody

Některé tyto obvody, zkráceně ASIC, byly navrženy výhradně pro paralelní zpracování operací. Jelikož je ASIC navržen speciálně pro danou aplikaci, je možné jej navrhnout maximálně optimalizovaný pro danou paralelizaci.

Procesory ASIC jsou vytvářeny pomocí rentgenové litografie, pro kterou musí být vytvořena speciální maska. Tato má poměrně vysokou cenu v závislosti na počtu tranzistorů (cena je i několik desítek milionů korun). Vzhledem k Mooreovu zákonu, který říká, že každých osmnáct měsíců se zdvojnásobí počet tranzistorů v procesoru, není příliš efektivní vyrábět paralelní systémy tímto způsobem – vysoké počáteční náklady a krátká morální životnost hovoří proti použití této technologie při využití v paralelních systémech. [11]

3.4.4 Vektorové procesory

Vektorový procesor je CPU nebo počítačový systém, který může na velké sadě dat vykonat stejnou operaci. Tato sada dat mohou být lineární pole čísel nebo vektorů, např. je systém použitelný pro násobení 64 elementových vektorů, každý se 64 bitovými čísly.

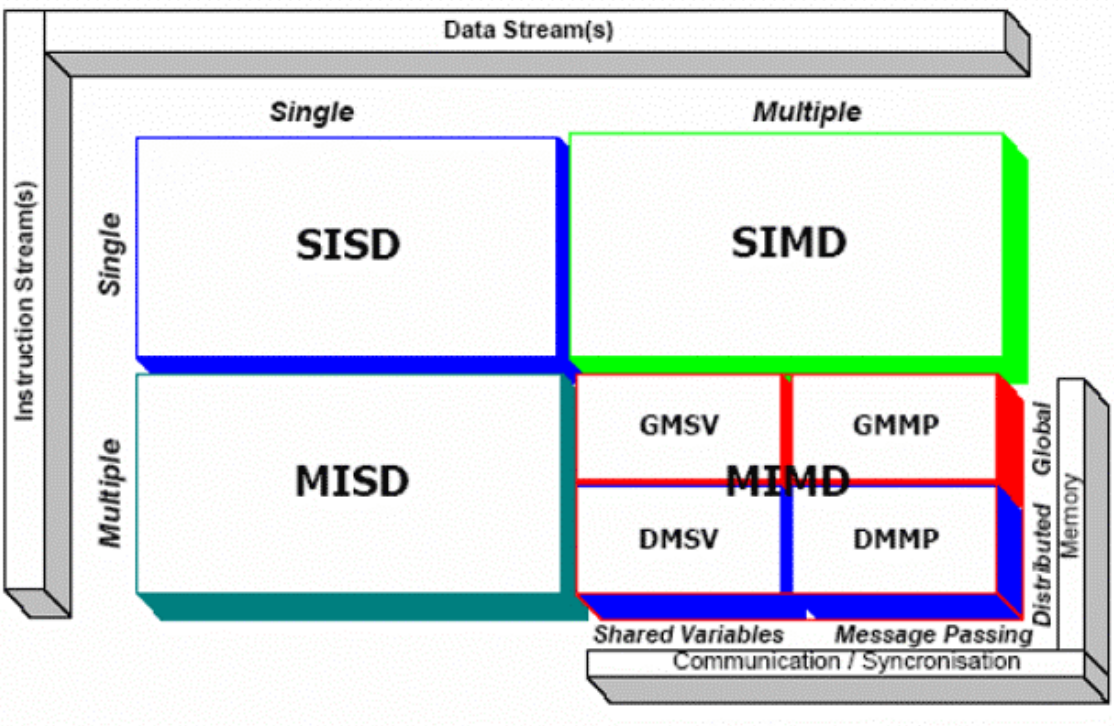
V dnešní době je podpora vektorových operací součástí instrukčních sad procesorů. Konkrétně se jedná o instrukce AltiVec a SSE. [11]

4 TAXONOMIE PARALELNÍCH ARCHITEKTUR

Paralelní systémy lze rozdělit z několika různých hledisek.

4.1 Flynn-Johnsonova taxonomie

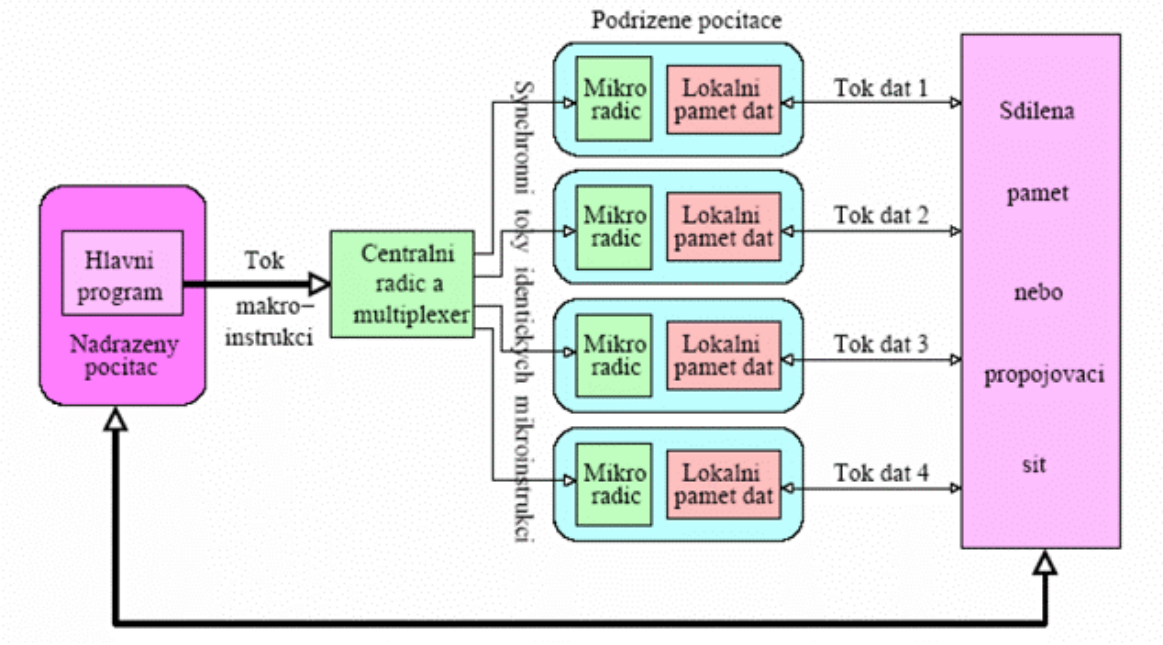
Taxonomie z hlediska toků instrukcí a dat je jedno z nejčastějších rozdělení paralelních systémů, i když v dnešní době ztrácí toto rozdělení vzhledem k výhradnímu zastoupení MIMD význam. Práce se bude dále zabývat pouze architekturou SIMD a MIMD. Architektura SISD je klasický Von Neumannův model se sekvenčním zpracováním dat a u počítačů MISD probíhá více instrukcí na jediném proudu dat a je nerealistický pro paralelní výpočty a proto nemá cenu je více popisovat. [4] [10]



Obrázek 4 Flynn-Johnsonova taxonomie [10]

4.1.1 SIMD – Single Instruction Multiple Data

Architektura SIMD počítačů je oproti MIMD značně jednoduchá a mezi paralelními počítači se nepoužívá. Je využitelná téměř výhradně tam, kde se zpracovávají množiny navzájem nezávislých výpočtů nebo při výměně dat mezi sousedy při operaci posun apod. I přes to, že se tento systém komerčně nepoužívá, slouží jeho základní verze PRAM pro popis a návrh paralelních algoritmů. [4]



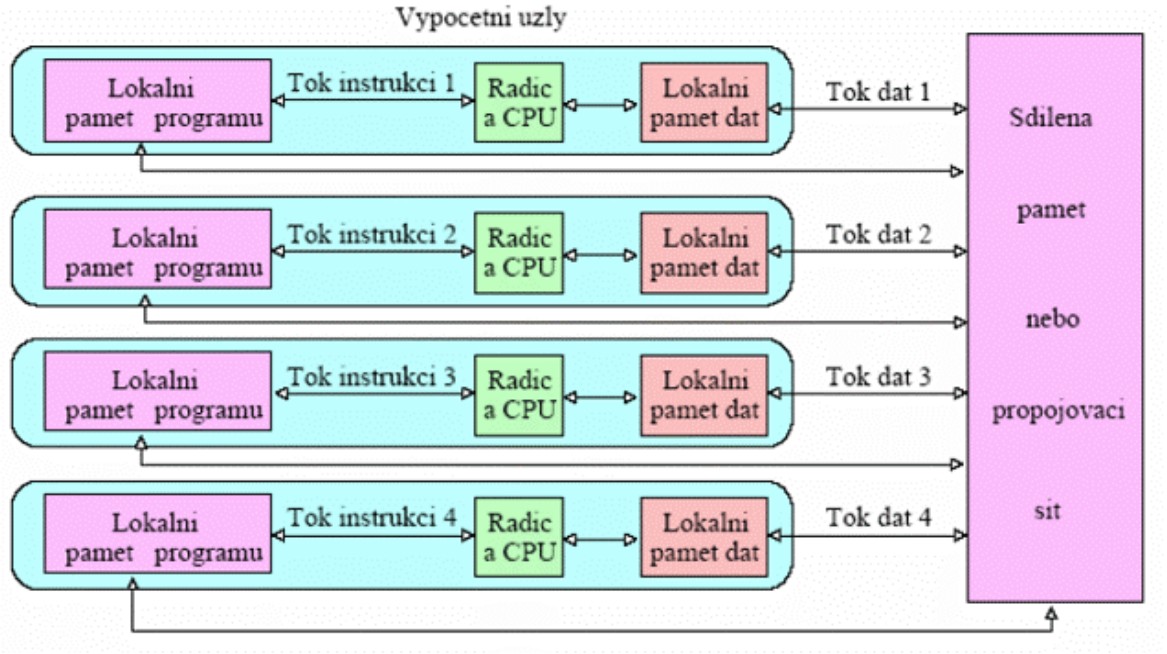
Obrázek 5 Schéma architektury paralelního počítače SIMD [4]

Vlastnosti tohoto systému: [4]

- Systém je složen z identických podřízených výpočetních uzlů s mikrořadičem a vlastní pamětí.
- Nadřazený počítač posílá do centrálního řadiče makroinstrukce, jež jsou přeloženy na mikroinstrukce předávané jednotlivým uzlům.
- Výpočty na všech procesorech jsou prováděny **synchronně** pod řízením centrálních hodin. Za jeden krok se považuje provedení jedné makroinstrukce.
- Počáteční data jsou načtena do lokální datové paměti výpočetního uzlu před započetím výpočtu, kde s nimi pracuje výpočetní uzel po celou dobu.
- I přes to, že posílá nadřazený počítač všem výpočetním uzlům stejnou makroinstrukci, některé jsou aktivní a některé jsou pasivní, bez čehož se nedá při programování algoritmů obejít z důvodu podmíněných skoků, příkazů atd. Který uzel je aktivní a který pasivní je zakódováno pomocí bitové masky v makroinstrukci.
- Procesory spolu komunikují buďto pomocí dat předávaných přes sdílenou paměť, nebo posíláním zpráv společnou propojovací sítí.

4.1.2 MIMD – Multiple Instruction Multiple Data

Význam počítačů s MIMD architektury stále roste a v dnešní době se již není téměř možné setkat s počítačem architektury SIMD. V kategorii MIMD počítačů začínají vzhledem k výhodnému poměru cena/výkon převládat clustery neboli svazky počítačů, které jsou výkonově stále více srovnatelné s dražšími masivně paralelními počítači. [4]



Obrázek 6 Schéma architektury paralelního počítače MIMD [4]

- Jednotlivé uzlové počítače v MIMD provádějí samostatné lokální výpočty a musí si být rovněž schopny samostatně načítat a ukládat data a program, jelikož nadřazený počítač nemusí existovat. Pokud existuje, je jeho úlohou pouze zavedení programů a dat do pamětí uzlových počítačů, na spouštění výpočtů a na sběr výsledných dat.
- Uzly mohou pracovat asynchronně či synchronně, jak vyžaduje daný problém.
- Je nutné, aby mezi sebou mohly komunikovat jednotlivé uzly. Buďto jenom dva nebo více. Proto je potřeba mít v systému synchronizační mechanismy, které jsou obvykle implementovány sice na SW úrovni, avšak na co nejnižší úrovni směrem ke komunikačnímu hardwaru.
- Vlastní komunikace mezi uzly je opět realizována buď přes sdílenou paměť nebo propojovací síť. [4]

4.1.3 Srovnání architektur SIMD a MIMD

SIMD uzly jsou většinou mnohem jednodušší než u architektury MIMD, což má za následek, že jsou mnohem levnější a menší.

Jelikož pracují MIMD systémy se sdílenou pamětí a nejsou řízeny posloupností mikroinstrukcí jako v případě SIMD výpočetních uzlů, pracují tyto systémy asynchronně. Je nutné provádět synchronizaci jednotlivých MIMD uzlů a to pomocí explicitních mechanismů pro koordinaci a synchronizaci paralelních procesů. Takto synchronizované počítače se nazývají synchronizované MIMD (SMIMD). SMIMD je v podstatě MIMD systém, který díky malé režii při synchronizaci umožňuje i přes asynchronnost výpočtu efektivně provádět operace nad paralelními datovými strukturami. Na rozdíl od SIMD neprobíhá synchronizace po provedení každé instrukce, ale vždy až po provedení určitého počtu instrukcí.

Významný rozdíl mezi SIMD a MIMD výpočtem je v případě provádění podmíněných příkazů, např.

if *podmínka* ***then*** *posloupnost instrukcí A* ***else*** *posloupnost instrukcí B*,

kde posloupnost instrukcí A a B jsou dvě libovolné posloupnosti instrukcí. Na SIMD počítači je zpracován tak, že v závislosti na stavu lokálních proměnných a/nebo adres procesorů, se množina procesorů P rozdělí na P_1, ve kterých podmínka platí a P_2=P-P_1, ve kterých neplatí. V první fázi procesory patřící do P_1 provádějí *posloupnost instrukcí A,* zatímco procesory patřící do P_2 jsou nečinné. V druhé fázi jsou naopak procesory v P_1 nečinné, zatímco procesory v P_2 provádějí *posloupnost instrukcí B*. Čím více instrukcí je v podmíněném příkazu **then/else** větvích, tím více výpočetního potenciálu paralelního SIMD počítače se plýtvá. Na druhou stranu, v MIMD počítači mohou obě skupiny procesorů provádět instrukční posloupnosti obou větví současně. SIMD počítače jsou vhodné pro dobře strukturované problémy s pravidelnými řídícími strukturami, kdy stejné nebo podobné operace je třeba vykonat nad různými částmi dat. MIMD počítač je výhodnější, jestliže struktura problému je nepravidelná a jednotlivé procesy, na které se výpočet rozdělí, jsou rozdílné. [4]

5 ASYMPTOTICKÉ FUNKCE

Pro vyhodnocování výkonu efektivnosti paralelních algoritmů se nejčastěji používá asymptotických porovnání funkcí, a to sekvenční a paralelní. Nejčastěji se porovnává horní mez, která označuje složitost nejlepšího známého algoritmu a spodní mez, která označuje nejmenší možnou dosažitelnou složitost, kterou nemůže žádný algoritmus překonat. Důležitou součástí těchto porovnání je uvedení popisu funkce včetně skrytých konstant, které nemůžeme přesně vyjádřit.

Důležitým a často zmiňovaným výrazem u vyjadřování asymptotických funkcí je řád funkce. Tento nevyjadřuje nic jiného než asymptotickou časovou složitost algoritmu. Touto složitostí není myšlen přesný čas, po který funkce běží, ale pouze čas řádový, proto právě výše zmíněný řád funkce.

Značení pro asymptotické porovnávání funkcí, které se používají i v jiné literatuře zabývající se složitostí algoritmů, používá následující výrazy: O, Ω, Θ, o, □. [4]

Definice: [4]

Nechť N^+ označuje množinu přirozených čísel a R^+ množinu kladných reálných čísel. Nechť $f, g : N^+ \rightarrow R^+$ jsou dvě funkce.

- O funkci *f(n)* řekneme, že je **řádu nejvýše** *g(n)*, psáno *f(n)=O(g(n))*, jestliže podíl *f(n)/g(n)* nepřesáhne od jistého *n* určitou kladnou konstantu, čili

$$\exists c \in R^+ \qquad \exists n_0 \in N^+ \qquad \forall n \geq n_0 : \qquad (f(n) \leq c \cdot g(n)) .$$

- O funkci *f(n)* řekneme, že je **řádu nejméně** *g(n)*, psáno *f(n)= Ω(g(n))*, jestliže podíl f(n)/g(n) neklesne od jistého *n* pod určitou kladnou konstantu, čili

$$\exists c \in R^+ \qquad \exists n_0 \in N^+ \qquad \forall n \geq n_0 : \qquad (f(n) \geq c \cdot g(n)) .$$

- O funkci *f(n)* řekneme, že je **stejného řádu** jako *g(n)*, psáno *f(n)= Θ(g(n))*, jestliže jsou stejné až na multiplikativní konstantu, čili

f(n)=O(g(n)) a *f(n)= Ω(g(n))*

- O funkci *f(n)* řekneme, že je **striktně nižšího řádu** než *g(n)*, psáno *f(n)= o(g(n))*, jestliže podíl f(n)/g(n) je od jistého *n* menší než jakákoliv konstanta, čili

$$\exists c \in R^+ \qquad \exists n_0 \in N^+ \qquad \forall n \geq n_0 : \qquad (f(n) < c \cdot g(n)) .$$

- O funkci *f(n)* řekneme, že je **striktně vyššího řádu** než *g(n)*, psáno *f(n)*= □*(g(n))*, jestliže podíl f(n)/g(n) přeroste od jistého *n* jakoukoli konstantu, čili

$$\exists c \in R^{+} \qquad \exists n_0 \in N^{+} \qquad \forall n \geq n_0 : \qquad (f(n) > c \cdot g(n)) .$$

Následující analogie mezi asymptotickými vztahy a reálnými čísly prezentovanými písmeny *a* a *b* slouží pro jednodušší pochopení těchto vztahů: [4]

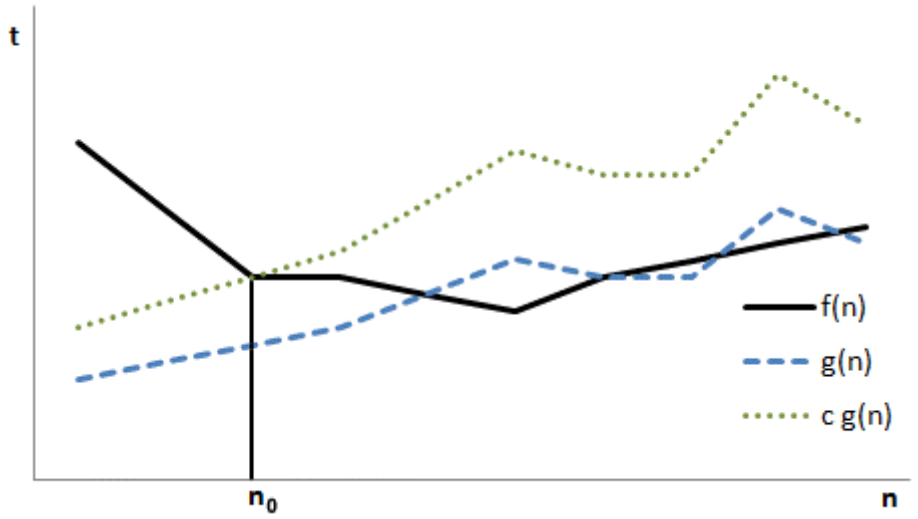
$$f(n) = O(g(n)) \qquad a \leq b$$

$$f(n) = \Omega(g(n)) \qquad a \geq b$$

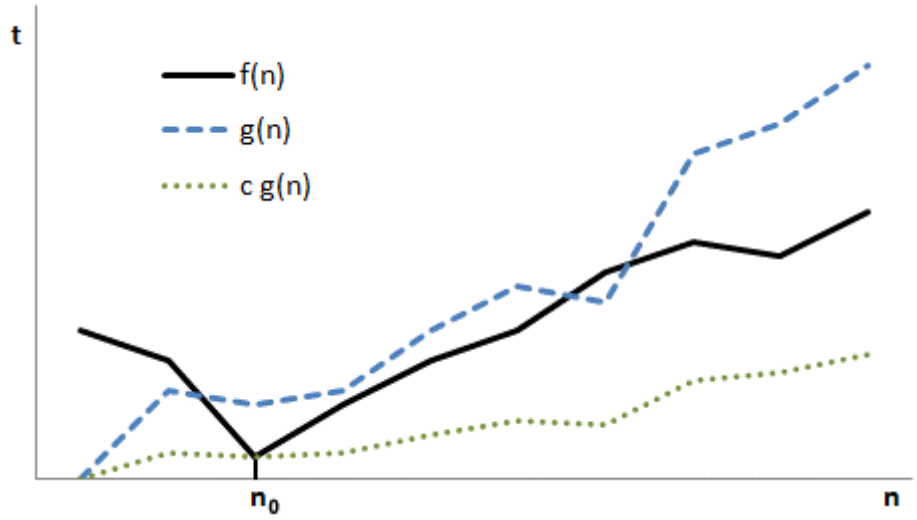
$$f(n) = \Theta(g(n)) \quad \approx \quad a = b$$

$$f(n) = o(g(n)) \qquad a < b$$

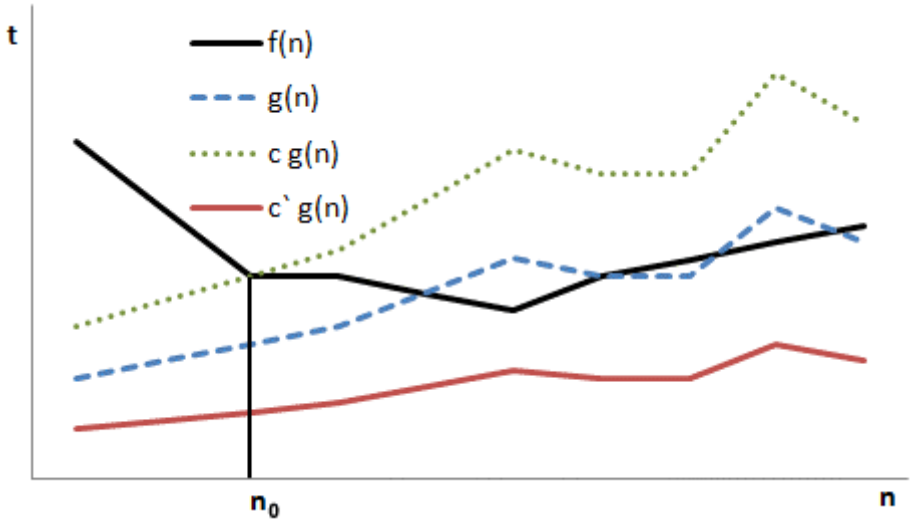
$$f(n) = \square(g(n)) \qquad a > b$$



Obrázek 7 Asymptotický vztah f(n) = O (g(n)) – funkce *f* je řádu nejvýše *g* [10]



Obrázek 8 Asymptotický vztah f(n) = Ω (g(n)) – funkce *f* je řádu nejméně jako *g* [10]



Obrázek 9 Asymptotický vztah f(n) = Θ (g(n)) – funkce *f* je stejného řádu jako *g* [10]

5.1 Zákony asymptotiky a standardní pojmy a značení

Tranzitivita:	$f(n) = O(g(n)) \wedge g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$. Podobně pro Ω, Θ, o, □.
Refexivita:	$f(n) = O(f(n))$. Podobně pro pro Ω, Θ.

Symetrie:	$f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$.
Transpoziční symetrie:	$f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$, $f(n) = o(g(n)) \Leftrightarrow g(n) = \varpi(f(n))$.
Inkluze:	$f(n) = o(g(n)) \Leftrightarrow g(n) = O(f(n))$, $f(n) = \varpi(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$.

Tabulka 1 Zákony asymptotiky [4]

Základní typy asymptotických funkcí jsou následující: [4]

- **Neznámá konstanta** se zapisuje jako *O(1)*.
- Funkce *f(n)* je funkcí **polynomiální**, pokud platí $f(n) = \Theta(n^{O(1)})$.
- Funkce *f(n)* je funkcí **subpolynomiální,** pokud platí $f(n) = o(n^{O(1)})$.
- Funkce *f(n)* je funkcí **polygaritmická**, pokud platí $f(n) = \Theta(log_2{}^{O(1)}(n))$.

6 ANALÝZA SEKVENČNÍCH ALGORITMŮ

Měřítkem výkonnost jednotlivých sekvenčních algoritmů je doba jejich běhu. Ta závisí na počtu instrukcí, výpočetních kroků či operací, které jsou daným algoritmem provedeny. [4]

6.1 Základní pojmy při analýze sekvenčních algoritmů

- K - řešený problém
- A - algoritmus řešící daný problém
- $ST_A^K(n)$ - **doba běhu** sekvenčního algoritmu
- $SL_A^K(n)$ - **spodní mez** algoritmu. Je to nejhorší možná doba běhu **nejrychlejšího možného** sekvenčního algoritmu. Tato mez nemůže být překročena.
- $SU_A^K(n)$ - **horní mez** algoritmu. Je to nejhorší možná doba běhu **nejrychlejšího známého** sekvenčního algoritmu.

Agoritmus A je **optimální** sekvenční algoritmus pro řešení problému K, pokud:

$$ST_A^K(n) = \Theta(SL_A^K(n)) = \Theta(SU_A^K(n)).$$

Algoritmus A je **nejlepší známý** sekvenční algoritmus pro řešení problému K, pokud:

$$ST_A^K(n) = \Theta(SL_A^K(n)) = \varpi(SU_A^K(n)).$$ [4]

7 ANALÝZA PARALELNÍCH ALGORITMŮ

Měření výkonnosti paralelních algoritmů vychází z analýzy sekvenčního algoritmu *A* řešícího daný problém K, která je rozšířena o další rozměr, a to **počet procesorů** *p.* [4]

7.1 Základní pojmy při analýze paralelních algoritmů

- *p* - **počet procesorů** je přímo závislý na počtu zpracovávaných dat a také závislý na daném algoritmu, který je použit. Se vzrůstajícím počtem procesorů se zvyšují časové nároky na režii algoritmu (viz. kroky v době běžu), proto nelze paušalizovat všechny operace jakožto vhodné pro zpracování paralelním algoritmem, i když existuje pro daný sekvenční algoritmus jeho paralelní ekvivalent.

- $T_A^K(n,p)$ - **doba běhu** sekvenčního algoritmu. Je to doba od začátku paralelního výpočtu až do chvíle, kdy skončí poslední procesor se svým výpočtem. Na rozdíl od sekvenčního výpočtu je nutné k této době připočíst ještě výpočetní a směrovací kroky, které jsou nutné pro operace s daty mezi procesory.

- $L_A^K(n,p)$ - **spodní mez** algoritmu. Je to nejkratší možná doba, za kterou může *p* procesorů vyřešit daný problém: $L_A^K(n,p) = \frac{SL_A^K(n)}{p}$.

- **Časově optimální paralelní algoritmus** je vždy vztažen k nejlepšímu známému sekvenčnímu algoritmu: $T_A^K(n,p) = \Theta\left(\frac{SU_A^K(n)}{p}\right)$.

- $C_A^K(n,p)$ - **cena paralelního algoritmu** je definována jako součin procesorů a doby běhu algoritmu: $C_A^K(n,p) = p \cdot T_A^K(n,p)$.

  Paralelní algoritmus je **cenově optimální**, pokud je stejného či nižšího řádu jak *SU:* $C_A^K(n,p) = O(SU_A^K(n))$ *popř.* $C_A^K(n,p) = \Theta(SU_A^K(n))$.

- $W_A^K(n,p)$ - **práce** paralelního algoritmu je podobná jako cena, avšak na rozdíl od ní je přesnější. Odpovídá totiž přesně počtu paralelně provedených operací. $W_A^K(n,p) = N_1 + N_2 + ... + Ni$, kde N je počet vytížených procesorů v jednotlivých krocích $i = \{1,2,...t_0\}$.

Pracovně optimální je algoritmus, jestliže je jeho řád nejvýše řádu horní meze časové složitosti řešeného problému: $W_A^K(n,p) = O(SU(n))$.

V praxi často není možné nečinné procesory uvolnit pro jiné výpočty, proto se častěji používá v paralelních algoritmech pojem cena paralelního systému než jeho práce.

- $S_A^K(n,p)$ - **zrychlení** paralelního algoritmu udává, o kolik je paralelní verze algoritmu rychlejší než sekvenční: $S_A^K(n,p) = \frac{SU_A^K(n)}{T_A^K(n,p)}$.

  Zrychlení je vždy menší nebo rovno počtu procesorů. Je-li $S_A^K(n,p) = p$ nebo $S_A^K(n,p) = \Theta(p)$, jedná se o **lineární zrychlení.** Teoreticky by se nemělo stát, že $S_A^K(n,p) \geq p$, jelikož zrychlení je přímo závislé na horní mezi a bylo by to v rozporu s její definicí, avšak ve skutečnosti tento stav nastat může a nazývá se superlineární zrychlení.

  Superlineární zrychlení nastává, pokud je zrychlení vyšší, než počet procesorů řešící původně sekvenční algoritmus. Tento stav by teoreticky neměl nastat, ale může k němu dojít, pokud HW paralelního počítače nabízí možnosti nedostupné pro počítač sekvenční nebo pokud je algoritmus typu Divide-and-Conquer (Rozděl a panuj).

- $E_A^K(n,p)$ - **efektivnost** paralelního algoritmu je dána poměrem sekvenčních a paralelních operací složitosti: $E_A^K(n,p) = \frac{SU_A^K(n)}{C_A^K(n,p)}$. Jedná se ve své podstatě o **zrychlení na jeden procesor:**

$$E_A^K(n,p) = \frac{SU_A^K(n)}{C_A^K(n,p)} = \frac{S_A^K(n,p) \cdot T_A^K(n,p)}{p \cdot T_A^K(n,p)} = \frac{S_A^K(n,p)}{p} \leq 1.$$

- $H_A^K(n,p)$ - **paralelní režie** udává, o kolik více operací provede paralelní algoritmus oproti sekvenčnímu: $H_A^K(n,p) = C_A^K(n,p) - SU_A^K(n)$. [4]

7.2 Zdroje neefektivnosti paralelních algoritmů

- **Nedostatek užitečné práce**, která by vytížila všechny použité procesory
- **Komunikační náklady** jsou vyšší než výpočetní složitost daného problému – čas, který by zabraly některé lokální výpočty je nižší než tyto výpočty posílat jednotlivým procesorům a přijímat od nich výsledky.
- **Synchronizace** jednotlivých procesorů je delší než efektivní čas, který stráví zpracováním dané skupiny dat.
- **Špatná distribuce práce**, tzn. nerovnoměrné zatížení jednotlivých procesorů daným algoritmem – některé procesory jsou nevytíženy díky špatnému přerozdělení dat mnohem dříve než jiné. [4]

7.2.1 Odstranění neefektivnosti

- **Technologické:**
  - Rychlejší komunikační HW.
  - Zmenšení komunikační režie na SW úrovni.
  - Překrývání komunikačních a výpočetních operací.
- **Aritmetické:**
  - Volba vhodného počtu procesorů vzhledem ke složitosti řešeného problému.
  - Dobré statické mapování algoritmu na paralelní architekturu.
  - Rovnoměrné statické rozdělení výpočetní zátěže.
  - Vhodné předřazování komunikačních operací před místa v programu, kde jsou vyměňovaná data potřeba. [4]

8 ŠKÁLOVATELNOST PARALELNÍCH ALGORITMŮ

Škálovatelností paralelního algoritmu se rozumí udržení co největší efektivity navzdory změně počtu využitých procesorů nebo množství zpracovávaných dat. [4] [10]

8.1 Brentův simulační princip

Věta: Uvažujme problém K se vstupními daty o velikosti n, řešitelný v t paralelních krocích na n procesorech při zanedbání komunikační režie. Nechť m_i je počet operací v kroku i. Pak $W_A^K(n,p) = \sum_{i=1}^{t} m_i$ a stačí použít $p = \max_{i=1}^{t} m_i$ procesorů $P_1, P_2, \ldots P_{p-1}$, čímž dostaneme $C_A^K(n,p) = p \cdot t = \max_{i=1}^{t} m_i \cdot t$. Uvažujme $p`$-prostorový počítač $M$ s $p` < p$ týmiž procesory. Jestliže lze u M též ignorovat komunikační režii mezi operacemi při řešení K na M, lze tentýž výpočet na M provést v $T_A^K(n,p`)$ paralelních krocích, kde

$$T_A^K(n,p`) = \frac{W_A^K(n,p)}{p`} + t .$$

Význam tohoto algoritmu spočívá v tom, že je možné libovolně snižovat zbytečně vysoký počet procesorů, které algoritmus využívá, přičemž doba výpočtu poroste nejvýše úměrně při zachování řádově stejné efektivnosti a celkové práci. [4], [10]

8.2 Izoefektivnost paralelních algoritmů

Pokud by byl nějaký paralelní algoritmus neefektivní, lze jeho efektivitu zvýšit pouze v případě, že je možné počet procesorů p volit v závislosti na počtu prvků n tak, aby byla **většina procesorů** vytížena **většinu času.**

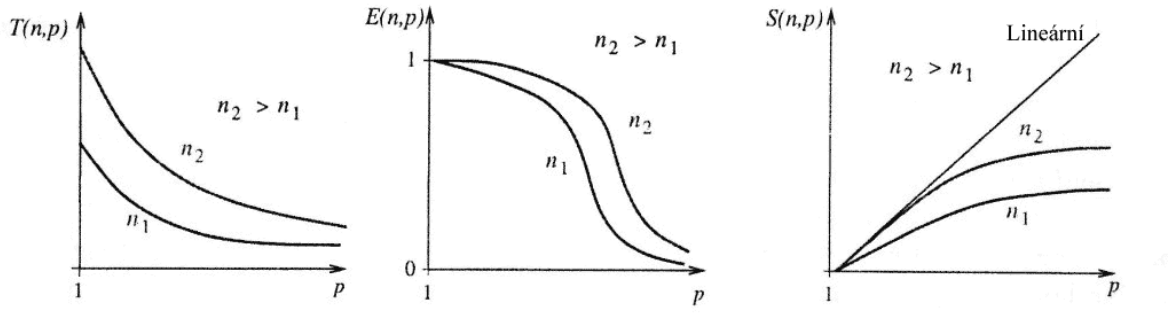
Jak je z předchozího odstavce a snad i celého předchozího textu patrné, ne vždy vede velké množství procesorů k rychlejšímu výsledku, na což je nutno pamatovat.

U všech algoritmů není možné zvýšit efektivnost snižováním počtu použitých procesorů. Výjimku tvoří algoritmy typu Rozděl-a-panuj, kde lze snižovat počet použitých procesorů bez případného snižování resp. stagnujícího stavu efektivnosti téměř vždy.

Stanovení horní a dolní meze rozumného počtu procesorů:

- **Dolní mez:** nejnižší počet procesorů může být procesor jeden. V tomto případě se bude jednat o sekvenční výpočet, který má sice nejhorší čas zpracování algoritmu, avšak maximální efektivitu.
- **Horní mez:** tuto není jednoduché vhodně zvolit. Sice se často volí $p = SU_A^K(n)$, avšak tato granularita je často zbytečně vysoká a proto se často používá *p=o(n)*. [4], [10]

8.2.1 Obvyklé vztahy závislostí paral. času, efektivnosti a zrychlení na *p*



Obrázek 10 Obvyklé vztahy závislosti paralelního času, efektivnosti a zrychlení na počtu procesorů *p* [4]

Pro udržení dobré efektivity resp. času je nutno při vzrůstajícím počtu procesorů zvyšovat i počet dat určených ke zpracování a rovněž i naopak. Tyto parametry *n* a *p* jsou vzájemně propojeny v přímé úměře.

Horní a dolní mez počtu procesorů určují **izoefektivní funkce** ψ_1, ψ_2. [4], [10]

8.2.2 Izoefektivní funkce

Definice:

Nechť je dána konstanta $0 < E_0 < 1$. Pak:

- ψ_1 je **asymptoticky minimální funkce** taková, že:

$$\forall n_p = \Omega(\psi_1(p)) : E_A^K(n_p, p) \geq E_0,$$

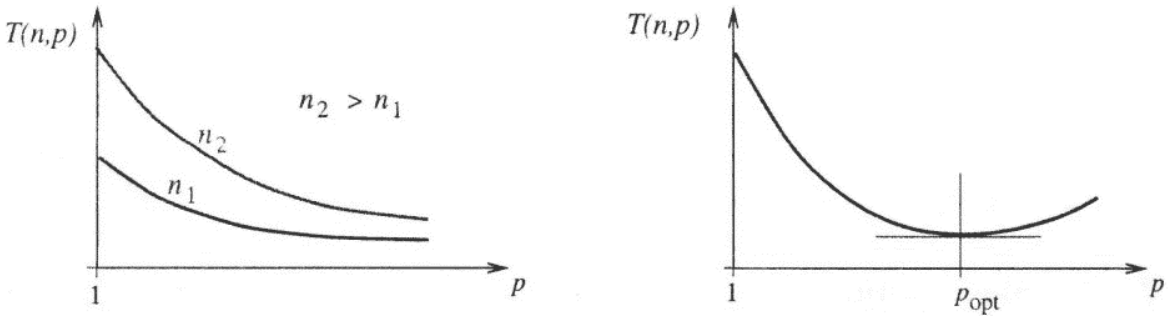
  čili $\psi_1(p)$ udává asymptoticky nejmenší instanci problému, která je na daném počtu procesorů p řešitelná s konstantní efektivností.

- ψ_2 je **asymptoticky maximální funkce** taková, že:

$$\forall p_n = O(\psi_2(p)) : E_A^K(n, p_n) \geq E_0,$$

čili $\psi_2(n)$ udává asymptoticky největší počet procesorů, který ještě poskytuje řešení dané instance problému o velikosti n s konstantní efektivností. [4], [10]

8.2.3 Absolutně minimální čas



Obrázek 11 Závislost času paralelního algoritmu na počtu procesorů [4]

Jak je z obrázku patrné, čas s rostoucím počtem procesorů klesá, avšak pouze do jisté hranice, za níž začíná časová náročnost paralelního algoritmu opět stoupat a to převážně díky veliké režii na správu a komunikaci tolika procesorů.

Minimální čas v bodě p_{opt} lze řešit rovnicí: $\left.\frac{\partial T_A^K(n,p)}{\partial p}\right|_{p=p_t} = 0$ a tím pádem $T_{\min} = T_A^K(n,p)$. [4], [10]

8.3 Amdahlův zákon

Velmi často zmiňovaný zákon mluvící o tom, že pro dané n má smysl zvyšovat p pouze do $\psi_2(n)$. Poté dochází k saturaci a další procesy nepřispívají k řešení.

Definice:

Každý algoritmus obsahuje **vrozeně sekvenční** a **vrozeně paralelní** části. Předpokládejme, že sekvenčně trvá výpočet čas $f_s+f_p=1$, kde f_s resp. f_p je relativní podíl sekvenční resp. paralelní části. S použitím p procesorů je paralelní čas $T_A^K(n,p) = f_s + \frac{f_p}{p}$. Zrychlení je tudíž:

$$S_A^K(n,p) \le \frac{1}{f_s + \frac{1-f_s}{p}}.$$

Pokud se do výše uvedeného vzorce dosadí nekonečno za počet procesorů, tzn. algoritmus bude řešen nekonečně velkým množstvím procesorů, bude S záviset pouze na převrácené hodnotě f_s. Pokud by tedy bylo například f_s 10%, bude $S_A^K(n,\infty) \leq 10$. [4], [10]

8.4 Gustafsonův zákon

Tento zákon platí pro masivně paralelní algoritmy, které je možno škálovat tak, že se vzrůstajícím počtem procesorů p se zvětšuje i čas f_p při zachování konstantního f_s. Při lineárním zvětšování f_p spolu s p se dá hovořit o lineárním zrychlování.

Definice:

Předpokládejme jednotkový čas $T_A^K(n,p) = f_s + f_p = 1$. Je-li takový algoritmus prováděn sekvenčně, bude trvat $T_A^K(n,1) = f_s + p \cdot f_p$, jeden procesor simuluje práci p procesorů. Pak je zrychlení:

$$S_A^K(n,p) = f_s + p \cdot f_p = f_s + p \cdot (1 - f_s) = p + f_s(1-p) = p\left(1 - f_s + \frac{f_s}{p}\right).$$

Stejně jako u Amdahlova zákona i tady lze při úpravě vzorce použít nekonečný počet procesorů. V tom případě vznikne: $S_A^K(n,\infty) = p(1-f_s) = \Theta(p)$.

Tento zákon ošetřuje speciální případ lineárně škálovatelných algoritmů. [4], [10]

9 PRAM MODEL POČÍTAČE

PRAM je velmi jednoduchý model SIMD architektury počítače a vychází z klasického sekvenčního RAM modelu. [4], [10]

9.1 RAM (Random Access Machine) model

- Základem RAM je výpočetní jednotka s uživatelsky definovaným programem
- S okolním světem komunikuje pomocí čtení vstupních dat a zápisu výstupních dat
- Počet lokálních paměťových buněk není nijak omezený
- Paměťové buňky jsou schopny obsahovat čísla neomezené velikosti.
- Instrukční sada zahrnuje instrukce pro přesuny dat mezi paměťovými buňkami, pro aritmetické a logické operace a pro větvení.
- Výpočet začne první instrukcí a skončí po provedení instrukce HALT.
- Všechny instrukce trvají jednotkový čas bez ohledu na délku operandů.
- Časová složitost je definovaná jako počet provedených instrukcí.
- Prostorová složitost je definovaná jako počet použitých paměťových buněk. [4], [10]

9.2 PRAM (Parallel RAM) model

Oproti klasickému RAM je u tohoto modelu použito více procesorů připojených ke společné paměti, které jsou donuceny pracovat synchronně jako u SIMD počítače. [4], [10]

- PRAM obsahuje neomezený počet procesorů RAM označených P_1, P_2…
- Paměť je tvořena neomezeným počtem sdílených paměťových buněk.
- Každí P_i má vlastní (neomezenou) lokální paměť (registry) a zná svůj index i.
- Každý procesor může přistupovat do kterékoli buňky sdílené paměti v jednotkovém čase.
- Vstup PRAM algoritmu se skládá z n položek uložených v n buňkách sdílené paměti.

- Výstup PRAM algoritmu se skládá z n` položek uložených v $n$` buňkách sdílené paměti.
- PRAM instrukce tvoří vždy třífázové cykly:

    1. Přečtení dat ze sdílené paměti do svého registru (Global Read)
    2. Provedení lokálního výpočtu nad svými registry
    3. Zápis dat ze svého registru do buňky sdílené paměti (Global Write)

- Procesory provádějí tyto třífázové PRAM instrukce synchronně.
- Konflikty typu R-R a W-W při souběžném přístupu do sdílené paměti je třeba explicitně ošetřit. Jediný způsob, jak si procesory mohou předávat či vyměňovat data, je zápisem do či čtením z buněk sdílené paměti.
- P_1 má speciální aktivační registr, obsahující nejvyšší index aktivního procesoru.

    1. Na počátku je aktivní pouze P_1.
    2. P_1 si spočítá počet požadovaných aktivních procesorů a nastaví aktivní registr.
    3. Pak začnou provádět své programy ostatní procesory.

- Výpočet běží až do doby, kdy se P_1 zastaví. V té chvíli jsou ostatní dříve aktivní procesory zastaveny.
- Paralelní časová složitost se rovná času výpočtu na P_1.

9.3 Typy PRAM (ošetření konfliktů při přístupu do paměti)

- Exclusive Read Exlucive Write (EREW) PRAM: každou paměťovou buňku lze přečíst nebo do ní zapsat v jednu chvíli pouze jedním procesorem najednou.
- Concurrent Read Exclusive Write (CREW) PRAM: každou paměťovou buňku lze přečíst libovolným počtem procesorů, ale může do ní v jednu chvíli zapisovat pouze jeden.
- Exclusive Read Concurrent Write (ERCW) PRAM: tento typ PRAM se nepoužívá
- Concurrent Read Concurrent Write (CRCW) PRAM: do každé paměťové buňky mohou v jednu chvíli všechny procesory jak zapisovat, tak číst.

- Shodný (common) CRCW: pokud v danou chvíli zapisují všechny procesory do paměťové buňky stejnou hodnotu, je vše v pořádku. V opačném případě není stav počítače definován.
- Náhodný (arbitrary) CRCW: libovolný procesory může zapsat do paměťové buňky, avšak není možné zjistit který. Ostatní procesory jsou při přístupu odmítnuty.
- Prioritní (priority) CRCW: každý procesor má přidělenou pevnou prioritu a zápis je možný procesorům podle dané priority, tzn. do dané paměťové buňky, může zapisovat pouze procesor s nejvyšší prioritou. [4] [10]

9.4 Výpočetní síla PRAM podmodelů

Definice: PRAM podmodel A je výpočetně silnější než podmodel B (A≥B), jestliže jakýkoli algoritmus napsaný pro PRAM počítač B poběží ve stejně velkém PRAM počítači A beze změny a s tímtéž paralelním časem. [4], [10]

$$Prioritní\ CRCW \geq Náhodný\ CRCW \geq Shodný\ CRCW \geq CREW \geq EREW$$

9.5 Cena, optimalizace a efektivnost PRAM algoritmů

Definice: Nechť K je problém s množinou vstupních dat o velikosti n. Předpokládejme, že K lze řešit na p-procesorovém počítači PRAM algoritmem A v čase $T(n,p)$, pak:

- A je **časově efektivní**, jestliže
  - $T(n,p) = O(\log^{O(1)} n)$
  - $C(n,p) = O(SU(n)\log^{O(1)} n)$
- A je **cenově optimální a časově efektivní**, jestliže
  - $T(n,p) = O(\log^{O(1)} n)$
  - $C(n,p) = O(SU(n)) = pT(n,p)$
- A je plně **paralelní**, jestliže
  - $T(n,p) = O(1)$
  - $C(n,p) = O(SU(n))$ [4] [10]

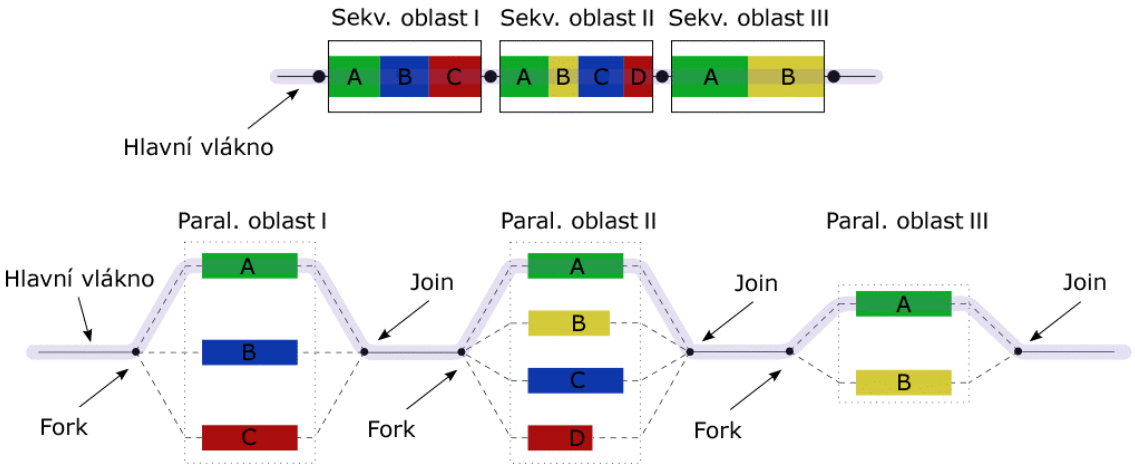
10 OPENMP

OpenMP je multiplatformní API pro programování aplikací využívajících víceprocesorové resp. vícejádrové počítače se sdílenou pamětí. Podporuje programovací jazyky c/c++ a Fortran pod operačními systémy resp. v aplikacích pro Windows, Linux a MacOS.

OpenMP je implementace multithreadingu, což je metoda paralelizace, kdy hlavní vlákno rozdělí zpracování určitých dat určitým způsobem podřízeným vláknům.

Část kódu, který se má zpracovávat více vlákny, je od ostatního kódu oddělen speciálními příkazy určenými pro preprocesorové zpracování, které způsobí, že kód nebude vykonán pouze hlavním vláknem, ale jeho vykonání bude rozděleno mezi dostupný nebo uvedených počet vláken (anglicky Fork). Každé vlákno má své vlastní ID, které může být pomocí určité funkce získáno, pokud je potřeba. Jednotlivá ID vláken mají celočíselnou hodnotu začínající od 1. ID 0 je rezervována pro hlavní vlákno. Po vykonání paralelní části všemi vlákny následuje ukončení všech spuštěných vláken (anglicky Join) a pokračování sekvenční části programu.

Jednotlivá vlákna mohou zpracovávat přidělená data stejným úsekem programu, nebo mohou být jednotlivým procesorům přiděleny paralelní zpracování různých částí programu, pokud to dané použití v programu umožňuje.



Obrázek 12 Multithreading [11]

Na vrchní části obrázku je znázorněno, jak vypadá program, pokud není zpracováván paralelně, tzn. je celý sekvenční. Jak je patrné, všechny části jsou zpracovávány postupně a tím dochází ke zdržení, jelikož následující mechanismy musí čekat, až procesor dokončí operaci s mechanismy předchozími.

V kontrastu je s tím druhá část obrázku, kdy je patrné, že dochází k rozdělení zpracování různých informací jednotlivými vláky a sníží se tím tak výsledný čas zpracovávání programu. U druhého rozdělení do vláken je vidět, že ne každé vlákno se vykonává konstantní dobu, ale je zde možné použít analogii s příslovím, že řetěz je tak silný jako jeho nejslabší článek, neboli paralelní oblast trvá tak dlouho, jako její nejdéle se provádějící vlákno. [11]

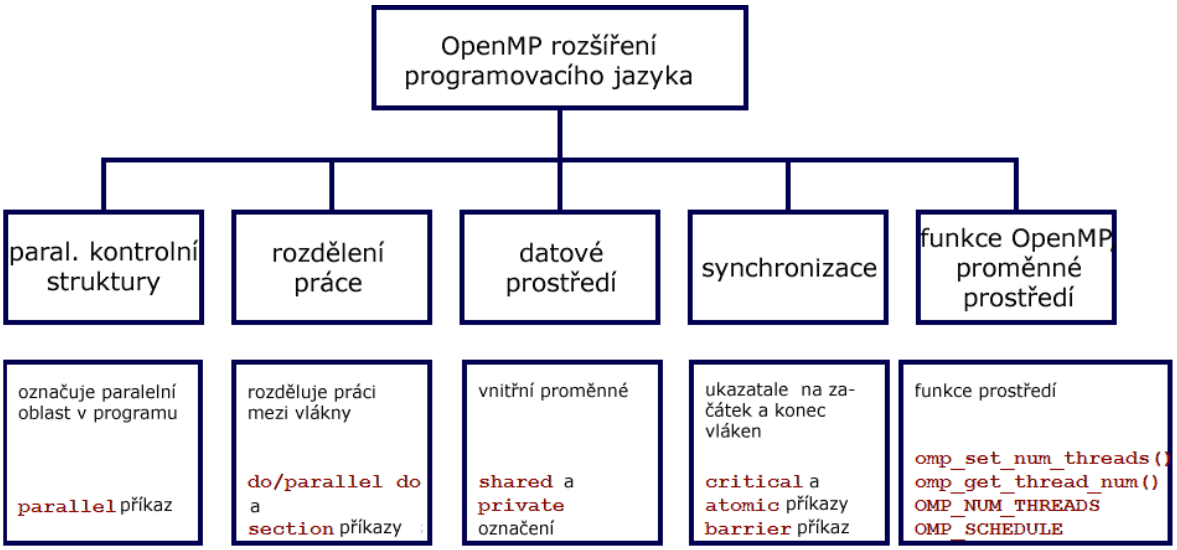
10.1 Historie

První oficiální specifikace pro OpenMP, a to pro OpenMP 1.0, byly uveřejněny v říjnu 1997 pro programovací jazyk fortran a o rok později pro C/C++. V roce 2000 byla uvolněna verze 2.0 pro fortran a v roce 2002 i pro C/C++. Od verze 2.5 vydané v roce 2005 jsou vydávány specifikace společně jak pro fortran, tak pro C/C++. Poslední verze 3.0 z května 2008 přidávající hlavně lepší práci s rekurzivními funkcemi. Tato však ještě dosud není nikde implementována.

Jen pro ilustraci implementace OpenMP, jedná se o API, jehož podporu je ještě potřeba implementovat do jednotlivých kompilátorů. V současné době (3 měsíce po uvedení OpenMP 3.0) jej ještě nezvládl implementovat žádný překladač.

Druhým příkladem pro ilustraci je např. vlastní překladač Intelu, který obsahuje jak OpenMP, tak svá vlastní rozšíření, která však zaručeně pracují pouze na procesorech tohoto výrobce. [11]

10.2 Struktura OpenMP



Obrázek 13 Struktura OpenMP [11]

10.2.1 Direktivy preprocesoru

Jednotlivé prvky OpenMP použité v kódu C/C++ musí mít před sebou tzv. **pragmu**, která určuje, že se má daný kód zpracovat jako direktiva preprocesoru. Pragma začíná mřížkou a poté následuje výraz *pragma*. Následující výraz *omp* definuje, že se další část týká OpenMP. [11]

```
1  #pragma omp <zbytek pragmy>
```

10.2.2 Vytvoření vlákna

Vytvoření vlákna předchází inicializace Forku, čímž se rozdělí běh programu z hlavního vlákna mezi definovaný počet podřízených vláken.

Následuje ilustrační příklad, který vypíše na obrazovku „Hello world“. Jelikož je tento výpis v paralelní oblasti bez dalšího upřesnění, vypíšou jej všechny vlákna „najednou“ (v uvozovkách proto, že jednotlivá Hello Word budou vypisována v závislosti na tom, jak rychle se dostanou jednotlivá vlákna k možnosti poslat výpis na obrazovku. [8], [11]

```
1  int main(int argc, char* argv[])
2  {
3    printf("Sekvencni oblast.");
4    #pragma omp parallel
5      printf("Hello world.");
6      printf("Opet sekvencni oblast.");
7    return 0;
8  }
```

Pokud bude např. počet dostupných procesorů resp. vláken 8, bude výpis následující:

```
 1  Sekvencni oblast.
 2  Hello world.
 3  Hello world.
 4  Hello world.
 5  Hello world.
 6  Hello world.
 7  Hello world.
 8  Hello world.
 9  Hello world.
10  Opet sekvencni oblast.
```

10.2.3 Rozdělení práce mezi vlákna

Tyto direktivy určují, jak rozdělit práci jednotlivým nebo všem vláknům.

- *Omp for* nebo *omp do:* rozdělí provedení cyklu mezi vlákna
- *Section:* rozdělení částí kódů, které jsou na sobě nezávislé, mezi více vláken
- *Single:* kód v této oblasti bude vykonán pouze jedním vláknem. Týká se vnořené oblasti, kde běží zároveň více vláken. Po ukončení této části je automaticky vložena bariéra, aby následující části paralelního algoritmu pokračovaly zároveň.
- *Master:* Podobně jako Single, avšak tato část kódu je vykonána pouze hlavním vláknem, tzn. s ID 0 a po ukončení není vložena bariéra.

Následující příklad paralelizuje cyklus for mezi dostupná vlákna, která provedou každé na své části zvětšení obsahu buňky v poli a o dvojnásobek její pozice. [11]

```
 1  #define N 100000
 2  int main(int argc, char *argv[])
 3  {
 4    int i, a[N];
 5    #pragma omp parallel for
 6    for (i=0;i<N;i++)
 7    {
 8       a[i]= 2*i;
 9    }
10    return 0;
11  }
```

10.2.4 OpenMP označení

Pro lepší práci v paralelní oblasti je potřeba, aby obsahovala některé možnosti, které pomohou zvýšit její efektivitu a možnosti použití na širší počet paralelních algoritmů.

Jelikož je direktiv a označení v OpenMP celkem velké množství, budou v této práci zmíněny jenom ty více používané. [11]

10.2.4.1 Typy viditelnosti proměnných

Výchozí hodnotou proměnných, které vstupují do paralelní oblasti je, že jsou společné pro všechny vlákna, tzn. všechna vlákna k nim mohou přistupovat a rovněž do nich zapisovat. Problém nastane, pokud by do stejné proměnné resp. do stejné buňky pole zapisovalo více vláken najednou. Potom by nastal souběžný zápis do stejné paměťové buňky a není jasné, jaká hodnota by do ní byla uložena. Proto mohou mít jednotlivá vlákna vlastní soukromé proměnné, se kterými pracuje pouze to dané vlákno. [5], [11]

- *shared* – tyto proměnné jsou sdíleny mezi všemi vlákny. Je to výchozí hodnota viditelnosti proměnné.
- *private* – data jsou soukromá pro každé vlákno, tzn. každé vlákno má svůj rezervovaný paměťový prostor pro svoji soukromou proměnnou. Např. pokud má aplikace 4 vlákna a má soukromou proměnnou s integer polem o velikosti pěti paměťových buněk, budou po startu paralelní oblasti vytvořeny v paměti 4 integer pole o velikosti 5, každé dostupné tomu danému vláknu.

10.2.4.2 Synchronizační příkazy

- *crytical section* – uzavřený blok v paralelní oblasti je vykonán všemi vlákny, avšak postupně, protože většinou jedno vlákno v tomto případě potřebuje informace od předchozího nebo aby nedošlo k souběžnému zápisu dat.
- *ordered* – strukturovaný blok je vykonán v pořadí, v jakém by byl vykonán v sekvenčním programu.
- *barrieer* – bariéra slouží pro synchronizaci vláken, pokud je vložena v paralelní oblasti. Např. pokud se vytvoří paralelní oblast za účelem jednoho výpočtu a za ní by měla následovat druhá paralelní oblast, není nutné ji vytvářet dvakrát, ale stačí vložit bariéru. Bariéra je automaticky vložena při ukončení paralelní oblasti.
- *nowait* – umožňuje vláknu, aby pokračovalo v práci i po dokončení své části části práce v paralelním algoritmu a nečekalo na dokončení ostatními vlákny. Pokud

není vloženo *nowait*, pokračuje, jako by v ní nebyla bariéra (paralelní *for*). [7], [10], [11]

10.2.4.3 Příkazy na rozdělování dat vláknům

Umožňuje vláknům, aby si vzala určité množství dat na zpracování. Buďto souvislý blok, nebo je možno modifikovat, po jak velkých dílech dat si je mají jednotlivá vlákna brát.

- *static* – je to výchozí možnost rozdělení dat vláknům a určuje, že si každé vlákno vezme souvislý blok dat. Pokud je u něj uveden ještě celočíselný výraz, označuje, jak velký souvislý blok dat si vlákno vezme. V prvním případě v příkladě, který následuje, by si vzalo každé vlákno blok pro výpočet s i=0…i=25. Ve druhém i=0…i=4, i=17…i=20 atd. Bráno pro *schedule(static)*.
- *dynamic* – vlákna si berou jednotlivá data podle toho, jak jsou vytížená. Pokud je jedno vlákno vytíženo jedním výpočtem kratší dobu než jiné, okamžitě si bere na zpracování první volnou část dat, která je k dispozici. [11]

```
1  #pragma omp for schedule (dynamic)
2  //popř. Schedule (static), schedule(static,4)
3  for(i = 1; i <= 100; i++)
4  {
5   foo(i);//neznámý čas vykonání, proto foo
6  }
```

10.2.4.4 Inicializace vstupních proměnných

- *firstprivate* – vstupní proměnné jsou soukromé pro každé vlákno, mají ovšem svoji inicializovanou hodnotu získanou před vstupem do paralelní oblasti.
- *lastprivate* – vstupní proměnné jsou soukromé pro každé vlákno, pokud nejsou navíc *firstprivate*, nemají definovánu počáteční hodnotu. Vlákno, které skončí v paralelní oblasti jako poslední, uloží hodnotu této soukromé proměnné do její veřejné sekvenční obdoby.
- *threadprivate* – podobně jako *firstprivate*, pouze s tím rozdílem, že instance proměnné je uchovávána mezi jednotlivými inicializacemi paralelních oblastí. [10], [11]

10.2.4.5 Redukce

- *reduction* – instance proměnné je v každém vlákně soukromá, ale před vstupem do sekvenční oblasti je na všech hodnotách těchto instancí provedena společná matematická operace (sečtení, vynásobení…). [11]

10.3 Ukázkový příklad

Jelikož se bude práce zabývat použitím OpenMP v prostředí C/C++, bude uveden krátký příklad v tomto jazyku, nikoliv ve fortranu. [11]

```
1  #include <omp.h>//hlavičkka s OpenMP direktivami
2  #include <stdio.h>
3
4  int main(int argc, char *argv[])
5  {
6      int M=10,N=40;
7      int i,j,k,b[M][N];
8      /*začátek paralelní oblasti, která bude mít dynamické přidělování
       dat po jednom a soukromé instance proměnných j a k*/
10     #pragma omp parallel schedule(dynamic, 1) private(j,k)
11     {
12         /*první for za pragma omp for má soukromou instanci proměnné,
           proto není potřeba tuto proměnnou označovat jako private*/
13         #pragma omp for
14         for(i = 2; i <= N-1; i++)
15         {
16             /*následující for-y jsou zpracovány vždy jedním vláknem a
               nejsou rozdělovány mezi vlákna jako v případě prvního
               for-u*/
17             for(j = 2; j <= i; j++)
18             {
19                 for(k = 1; k <= M; k++)
20                 {
21                     b[i][j] += a[i-1][j]/k + a[i+1][j]/k;
22                 }
23             }
24         }
25     }
26     return 0;
27 }
```

II. PRAKTICKÁ ČÁST

11 PC KONFIGURACE PRO TESTOVÁNÍ ALGORITMŮ

Počítačem určeným k vývoji a testování algoritmů pro tuto práci byl počítač Fujitsu-Siemens Celsius se dvěma čtyřjádrovými procesory Intel Xeon.

Počítač	Siemens-Nixdorf Celsius R540
Procesor (2x)	Intel Quad-Core Xeon E5310 Takt jader: 1,6MHz L2 Cache: 2x 4MB FSB: 1066MHz
Paměť (2x)	2048MB FB-DIMM PC2-5300 ECC
Pevný disk (2x)	250GB SATA 2, 7200 ot/min.

Tabulka 2 HW konfigurace testovacího počítače

Tento počítač byl umístěný na Fakultě aplikované informatiky, UTB ve Zlíně na lokální síti, avšak měl namapovanou svoji vlastní veřejnou IP adresu, na kterou se připojoval klient přes vzdálenou plochu.

Jako klient byl použit program NX Client for Windows 3.2.0-10 pod operačním systémem Windows Vista, který umožňuje vzdálené připojení k počítači s operačním systémem Linux přes svou klientskou aplikaci, která je kromě operačního systému Windows dostupná také pro Linux, Mac OS X a Solaris. Přes tohoto klienta je možné se kromě operačního systému Linux připojit také do operačního systému Solaris, který musí být, stejně jako Linux, vybaven vhodnou instalací programu NX.

Operační systém	Linux
Verze	Ubuntu 8.04
Grafické prostředí	Gnome

Vývojové prostředí	Code::Blocks IDE, build 5182
Kompilátory	GCC 4.2.3 Intel C/C++ compiler 10.1.015

Tabulka 3 SW konfigurace testovacího počítače

Celý systém byl nakonfigurován pro bezproblémový chod a pro co nejsnadnější orientaci i studenta, který ještě více s tímto operačním systémem krom příkazové řádky nepracoval. Kromě vývojového prostředí byl ještě používán při práci souborový manažer Gnome Commander pro práci se soubory a jejich zálohu na FTP.

12 TESTOVÁNÍ ALGORITMŮ

12.1 Výběr algoritmů

Byly vybrány algoritmy ze skript profesora Tvrdíka, které jsou zřejmě nejlepším snadno dostupným zdrojem informací o paralelních algoritmech a paralelním programování.

Výběr paralelních algoritmů by měl obsáhnout jejich základní vlastnosti a možnosti resp. potenciál, který mají při paralelním zpracováním v kontrastu se zpracováním sekvenčním.

12.2 Vývojové diagramy

Vývojové diagramy byly vytvořeny dle norem UML, avšak pro upřesnění je dobré provést ještě pár doplňujících informací.

Začátek resp. konec paralelní oblasti je značený obrazcem *fork/join*. U něj lze jednoduše poznat, který z nich *fork* a který *join*, protože do *fork*-u přichází jedno vlákno a vychází jich více a u *join*-u přesně naopak. Ilustračně jsou uvedena pouze rozdělení na dvě vlákna, avšak je tím obecně myšleno rozdělení na více vláken. V této práci dostupných maximálně osm. Podle složitosti vývojového diagramu či některé jeho části je tento buďto nakreslen jako celek se všemi funkčními částmi nebo jsou uvedeny v posloupnosti jednoduše bloky, které následující obrázky (diagramy) rozkreslují.

12.3 Postupy měření a analýzy výsledků

Většina měření jsou provedena na dostatečně velké množině dat, aby bylo možné určit čas jejich zpracování. Aby byly v práci relativně co nejpřesnější výsledky, byla všechna měření provedena desetkrát a výsledný čas byl získán jejich zprůměrováním. Díky velmi podobné charakteristice u všech pokusů nebylo potřeba provádět žádné přídavné statistické úpravy naměřených časů.

V programech bylo místo, pro většinu běžného, použití práce s poli typu `pole[pozice]` použity ukazatele, které jsou při práci s poli mnohem rychlejší. [6]

12.4 Měření času algoritmů

Nejdůležitějším faktorem určujícím vlastnosti resp. kvalitu paralelních algoritmů oproti jejich sekvenční verzi, je výkonnost neboli skutečné zrychlení zpracování dat daným algoritmem.

Aby mohl být čas algoritmů změřen, je nutné implementovat do zdrojového kódu obsahujícího daný algoritmus také kód, který zajistí změření času. Pro co nejpřesnější změření času je nutné umístit kód pro jeho start před kód algoritmu a ukončení měření času přímo za konec měřeného algoritmu. Do měření tudíž nejsou započítány časy pro vytvoření proměnných, jejich případné výpisy a další operace, které přímo nesouvisí s daným algoritmem.

Pro čítání času v Linuxu je potřeba pracovat s knihovnou `sys/time.h` (řádek 4). Ta obsahuje vlastní knihovny a typy proměnných, vhodné pro čítání času. Pro potřeby této práce byly použity struktura `timezone`, typ proměnné `timeval` a funkci `gettimeofday`.

Struktura `timezone` je schopná ukládat i časové jednotky v řádech mikrosekund. Tento čas je pomocí funkce `gettimeofday` uložen v proměnných `nStart` a `nEnd`.

Měření času bylo pro přesnější výsledky provedeno vždy desetkrát a následně proveden aritmetický průměr pro každou hodnotu, aby došlo k co nejpřesnějšímu měření. Jelikož byly mezi jednotlivými měření minimální rozdíly, nebylo potřeba provádět měření pro zpřesnění více pokusů.

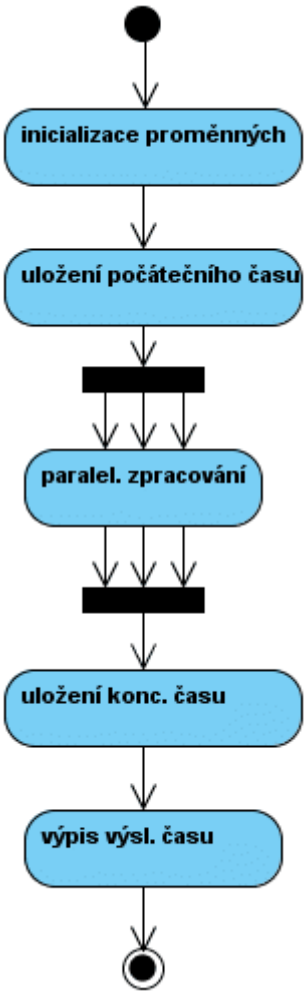
12.4.1 Popis programu

Pro ilustraci byl vybrán jednoduchý příklad, ve kterém jsou inicializována všechna vlákna pro výpočet součtu 100 čísel od 0 do 99 pro každé vlákno a následně redukce proměnné `nSum` sumou všech jejích instancí v jednotlivých vláknech. [2]

Nejasnosti by mohlo pouze způsobit vypsání času na řádku 22 a 23. Struktura `timeval` obsahuje členské proměnné `tv_sec` a `tv_usec`. `Tv_usec`, ve které je uložen čas v mikrosekundách v rozsahu $1\text{-}10^6$. V případě překročení času 1s jde čítání v této proměnné opět od začátku, proto je nutné použít i proměnnou `tv_sec`. Ve výsledku je nutné `tv_usec` zmenšit o 6 řádů, jelikož je tato hodnota celočíselná a potřebujeme ji spojit s, o šest řádů vyšší hodnotou sekund.

V ostatních částech práce nebude kvůli zbytečnému opakování stejného schématu toto uváděno a bude se počítat s tím, že při inicializaci proměnných budou inicializovány i proměnné pro měření času a po skončení algoritmu budou vypsány na obrazovku. Toto bude pouze ve zdrojovém kódu, ale kvůli případné velikosti již ne ve vývojovém diagramu.

```
 1  #include <omp.h>
 2  #include <stdlib.h>
 3  #include <stdio.h>
 4  #include <sys/time.h>//načtení knihovny pro práci s časem
 5
 6  #define MAX 100
 7
 8  int main(int argc, char* argv[])
 9  {
10    /*definování proměnných pro uložení startu a konce času čítání */
11    timeval nStart, nEnd;
12    int nSum=0;//pomocná proměnná, pro operace vláken
13
14    /*začátek počátečního času*/
15    gettimeofday(&nStart, (struct timezone*)0);
16
17    /*začátek paralelní oblasti*/
18    #pragma omp parallel shared(nSum)
19    {
20      #pragma omp for schedule(static) reduction (+:nSum)
21      for(long i = 0; i < MAX; i++)
22      {
23        nSum += i;
24      }
25    }
26    /*konec paralelní oblasti a zapsání času konce čítání*/
27    gettimeofday(&nEnd, (struct timezone*)0);
28    printf("Calculation done in %lf seconds.", (nEnd.tv_sec -
      nStart.tv_sec) + (nEnd.tv_usec - nStart.tv_usec)*1.0e-6);
29    return 0;
30  }
```



Obrázek 14 Diagram měření času algoritmu

V souvislosti s diagramy je ještě nutné uvést několik druhů proměnných, které budou uvedeny bez toho aniž by se objevily ve zdrojovém kódu. Je to z důvodu principu práce paralelních algoritmů a pro jejich lepší pochopení.

- `Thread_id` – id procesoru zpracovávajícího dané vlákno
- `Cycle` – čítač cyklů, které provedou všechna vlákna. Např. při paralelizaci cyklu `for(i=0;i<20;i++)` a počtu 4 vláken se maximum hodnoty `cycle=4`. Hodnota `cycle` se zvyšuje o 1 při každém návratu na začátek příslušné paralelní oblasti.

Při uvádění paměťové náročnosti aplikací nebude uváděna paměťová náročnost proměnných na čítání času, jelikož je pro změření a vyhodnocení bezvýznamná a rovněž nebudou uváděny při řádově milionech proměnné velikosti jednotek, které mají také nulový význam a pouze zvětšují výslednou rovnici bez jakéhokoliv praktického poznatku.

12.5 Násobení matic

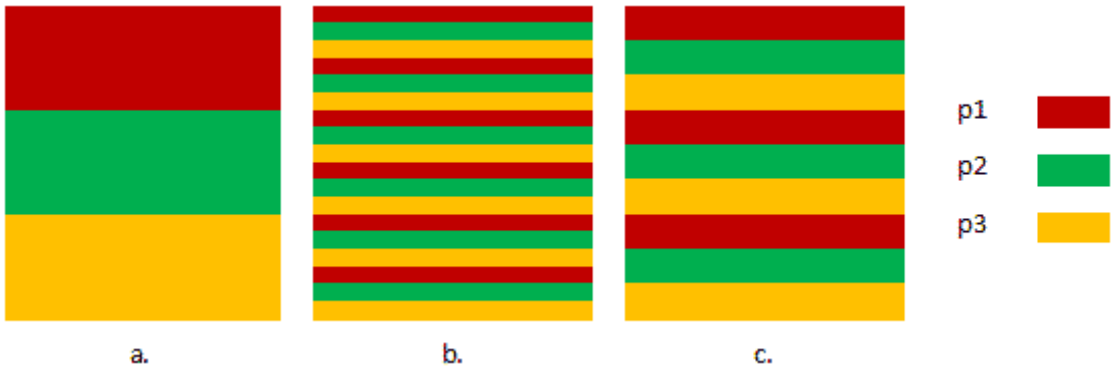
Toto je jeden z nejjednodušších algoritmů, který lze pomocí paralelního programování řešit a také jeden z nejvíce variabilních, jelikož lze dojít k výsledku několika způsoby (viz. Obrázky 15 a 16).

Proužkové mapování (Obrázek 15) [4]

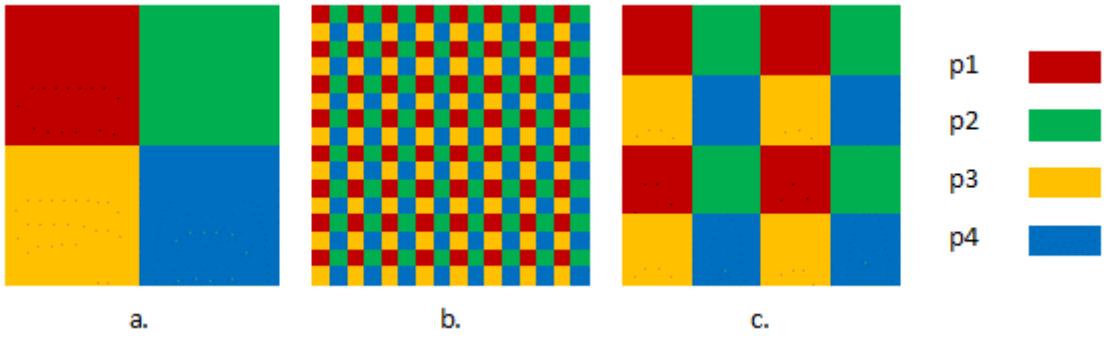
- **po řádcích** nebo **po sloupcích**
- **blokově** (Obr. 15 a.)**, cyklicky** (Obr. 15 b.) nebo **blokově-cyklicky** (Obr. 15 c.)

Šachovnicové mapování (Obrázek 16)

- **blokově** (Obr. 16 a.)**, cyklicky** (Obr. 16 b.) nebo **blokově-cyklicky** (Obr. 16 c.)



Obrázek 15 Proužkové mapování matic po řádcích [4]



Obrázek 16 Šachovnicové mapování matic [4]

12.5.1 Popis programu

Na začátku programu byly definovány konstanty pro vytvoření matice. Konstanta `ELEMENTS` vyjadřuje orientačně počet prvků, který bude vygenerován a `WIDHT` počet

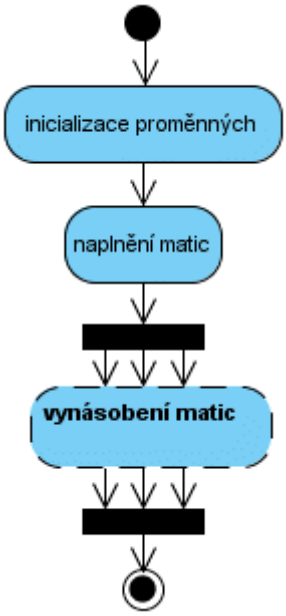
sloupců, do kterých se elementy rozdělí. Pokud by bylo nevhodně zvolené číslo `ELEMENTS` nebo `WIDHT`, tím je myšleno, že by nedošlo k celočíselnému podělení beze zbytku, nevadí to, jelikož matice dále pracují už pouze s parametry `WIDTH` a `HEIGHT`, které jsou odvozeny od `ELEMENTS`.

Při vytváření náhodných hodnot v paměťových buňkách musel být zvolen vhodný rozsah, aby nedošlo k přetečení hodnot ve výsledné matici. Velikost max. hodnoty ve výsledné matici lze dostat jako vynásobení `WIDHT` a druhou mocninu maximálního čísla, které je možné vygenerovat pro jednotlivé matice. V tomto případě číslo ze zřejmé, že matematická operace `rand()%2000` (řádky 21-25) vygeneruje rozsah hodnot 0-1999. Maximální hodnota je tedy 1999. Vynásobením vznikne max. hodnota čísla v paměťové buňce: 1 000 * 2000 * 2000=4 000 000 000. Pro toto číslo nám stačí paměťová buňky typu `unsigned long`, který má velikost 4 Byty. [2]

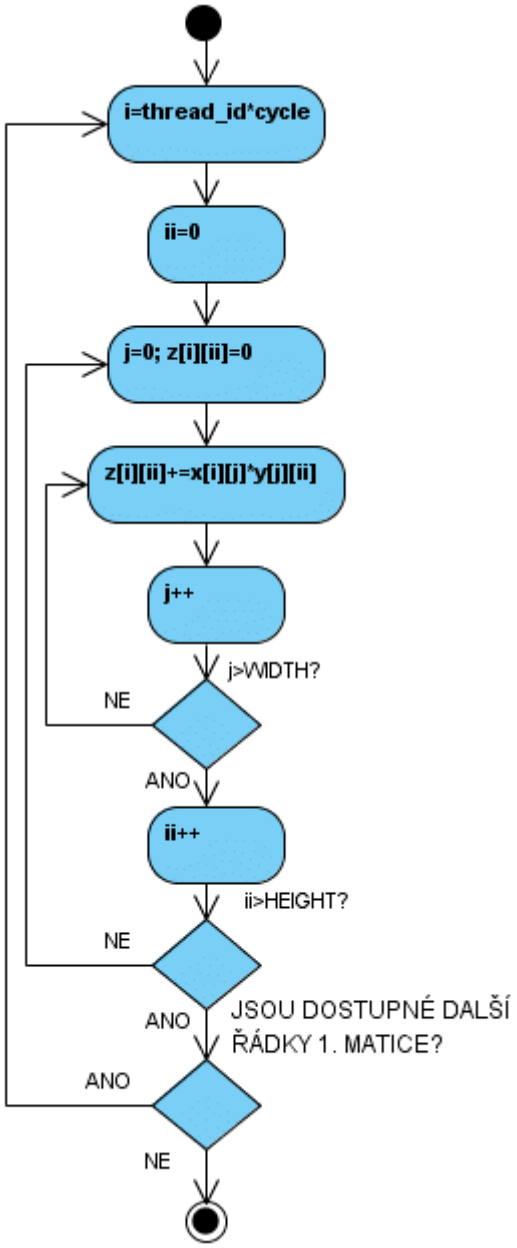
Někoho by mohlo překvapit, že např. i oblast pro naplnění matic (řádky 20-23), by mohly být vytvořeny více procesory. Pravdou však je, že funkce rand závisí na generátoru pseudonáhodných čísel a ten je v PC pouze jeden – došlo by k opačnému efektu. Ke generátoru by se snažilo přistupovat více procesorů zároveň a čas potřebný pro vytvoření matice by se zvyšoval.

V programu je řádek první matice označen proměnnou `i` a sloupec `j`. Pro druhou matici jsou použity proměnné `j` pro řádek a `i` pro sloupec. Jelikož musí být řádky násobených matic stejného počtu je použita proměnná `j` v obou maticích. Výsledná matice má rozměry `i` řádků a `ii` sloupců.

Samotný algoritmus zpracovával matice pomocí cyklického proužkového mapování, což znamená, že každý procesor si vzal řádek `n*p` (`n` – počet cyklů, `<1,ceil(ELEMENTS/p)>;` `p` – počet procesorů).



Obrázek 17 Diagram algoritmu násobení matic



Obrázek 18 Rozšiřující diagram k algoritmu násobení matic: vynásobení matic

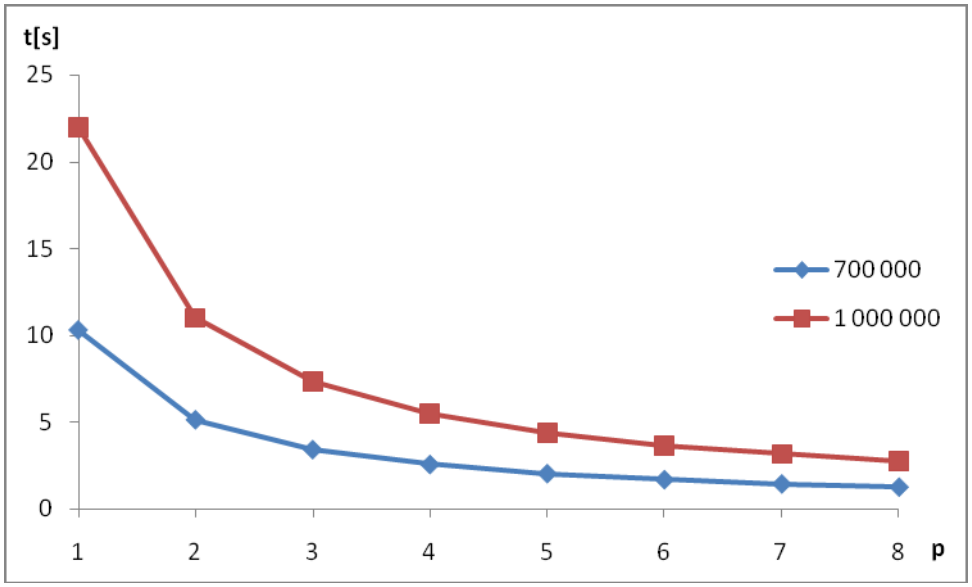
```
 1  #include <omp.h>
 2  #include <stdlib.h>
 3  #include <stdio.h>
 4  #include <math.h>
 5  #include <sys/time.h>//nacteni knihovny pro praci s casem

 6  #define ELEMENTS 700000//pocet prvku v nasobenych maticich
 7  #define WIDTH 1000//pocet sloupcu prvni(radku druhe) matice

 8  int main(int argc, char* argv[])
 9  {
10    int nThreads = 2;//pocet aktivnich vlaken

11    unsigned int i,ii,j;//promenne pro jednotliva vlÃ˘kna

      /*definovani poctu radku prvni(=sloupcu druhe) matice*/
```

```
12    unsigned int HEIGHT = floor(ELEMENTS/WIDTH);

13    /*vytvoreni ukazatelu a pametoveho prostoru o pozadovane delce v
      pametovem prostoru, na ktery ukazatele odkazuji*/
14    unsigned int *x = NULL, *y = NULL;
15    unsigned long *z = NULL;
16    x = (unsigned int*)malloc(HEIGHT*WIDTH*sizeof(unsigned int));
17    y = (unsigned int*)malloc(HEIGHT*WIDTH*sizeof(unsigned int));
18    z = (unsigned long*)malloc(HEIGHT*HEIGHT*sizeof(unsigned long));

19    timeval nStart, nEnd;

20    /*vytvoreni presudonahodnych cisel v rozsahu 0-1999*/
21    for(i=0;i<ELEMENTS;i++)
22    {
23      *(x+i) = rand()%2000;
24      *(y+i) = rand()%2000;
25    }
26    omp_set_num_threads(nThreads);
27    gettimeofday(&nStart, (struct timezone*)0);//start mereni
28    #pragma omp parallel shared(z) private(i,ii,j)
29    {
30      #pragma omp for schedule(static,1)
31      for (i=0;i<HEIGHT;i++)
32      {
33        for (ii=0 ; ii < HEIGHT ; ii++)
34        {
35          *(z+i*HEIGHT+ii)=0;
36          for (j=0;j<WIDTH;j++)
37          {
38            *(z+i*HEIGHT+ii)+=*(x+i*WIDTH+j) * *(y+j*HEIGHT+ii);
39          }
40        }
41      }
42    }
43    gettimeofday(&nEnd, (struct timezone*)0);//konec mereni
44    printf("Vynasobení matic %d procesory bylo provedeno za %.5lf
      sekund.", nThreads, (nEnd.tv_sec - nStart.tv_sec) + (nEnd.tv_usec -
      nStart.tv_usec)*1.0e-6);

45    return 0;
46  }
```

12.5.2 Výkonnost algoritmu

	rychlost zpracování prvků v maticích [s]	
procesorů	700 000 prvků	1 000 000 prvků
1	10,322227	22,01489
2	5,163768	11,024287
3	3,449916	7,359756
4	2,589361	5,530347
5	2,066864	4,414584
6	1,721082	3,675631
7	1,47347	3,190182
8	1,292842	2,768926
sekvenční	8,443603	18,32702

Tabulka 4 Rychlost výpočtu násobení matic pro 2 velikosti vstupních polí



Obrázek 19 Rychlost výpočtu násobení matic pro 2 velikosti vstupních polí

Paměťová náročnost algoritmu je:

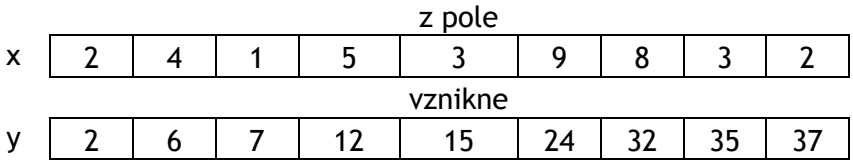
- `Pole x: HEIGHT * WIDTH * 2 Byte`
- `Pole y: HEIGHT * WIDTH * 2 Byte`
- `Pole z: HEIGHT * WIDTH * 4 Byte`
- `Proměnné nThreads, Height, i: 3 * 2 Byte`
- `Proměnné ve vláknech ii, j: HEIGHT * p * 2 Byte * (1 + WIDTH)`
- Paměťová náročnost pro 1 000 000 prvků: `(1000 * 1000) * 2 * 2 * 4 + 3 * 2 + 1000 * 8 * 2 * 1001= 15,29MB.`

Asymptotické funkce:

- $T(n,p) = O\left(\frac{i \cdot ii \cdot j}{p}\right)$
- $C(n,p) = O(i \cdot ii \cdot j)$

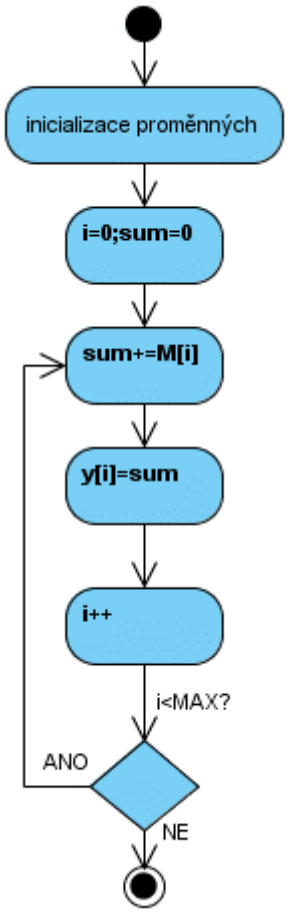
12.6 Paralelní prefixový součet na EREW PRAM

Tento součet je způsob paralelní redukce, kdy následující hodnota v poli dat obsahuje součet svojí a předcházející buňky. Jak je patrné z obrázku Obrázek 20 Výpočet prefixového součtu, $y_i=x_1+x_2+\ldots+x_i$.

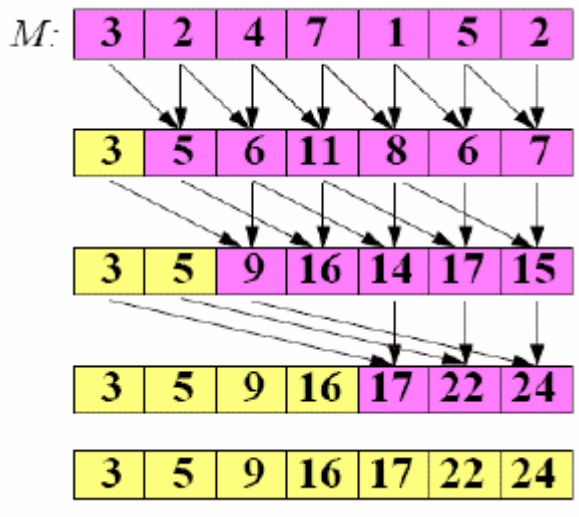


Obrázek 20 Výpočet prefixového součtu [4]

Při představě sekvenčního prefixového součtu napadne většinu lidí základní algoritmus, který se sečítá postupně jedna buňka s následující atd. v předem daném pořadí.



Obrázek 21 Diagram sekvenčního prefixový součet



Obrázek 22 Výpočet PPS [4]

Při výpočtu PPS na EREW PRAM funguje algoritmus zcela jinak.

V 1. kroku provede každý procesor součet své buňky s paměťovou buňkou nalevo od něj, posunutou o 2^0 a výsledek uloží do druhého pole. Např. první procesor sečte čísla 3 a 2 (index 0 a 1), druhý procesor čísla s indexy 2 a 4 (index 1 a 2), třetí 4 a 7 (index 2 a 3) atd. Operace se začíná s číslem rovnajícímu se číslu posunutí, v tomto případě 2^0, tedy na čísle s indexem 1.

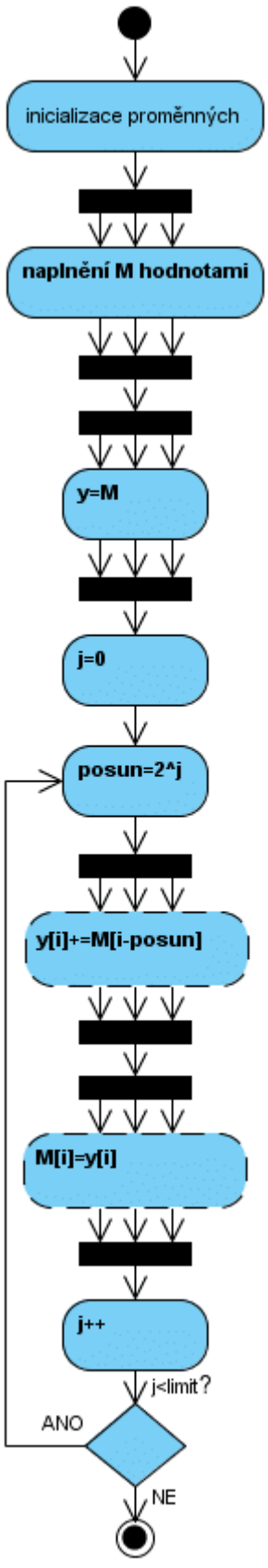
Ve 2. kroku se zvětší exponent posunutí o 1, tzn. z 2^0 na 2^1. V závislosti na tom se zvětší index prvního prvku přiřazeného prvnímu vláknu rovněž na 2^1, čemuž odpovídá i posun čísla, které se s tímto sečte. V tomto případě se k číslu 6 přičte 3 (index 2 a 0), k 11 se přičte 5 (index 3 a 1) atd.

Třetí, poslední krok, opakuje toto vše s navýšením posunu na 2^2.

Tento algoritmus je použitelný pouze pro počet buněk rovný počtu procesorů. Pokud by velikost matice `M` překročila počet prvků, stane se tento algoritmus v porovnání se sekvenčním neefektivním.

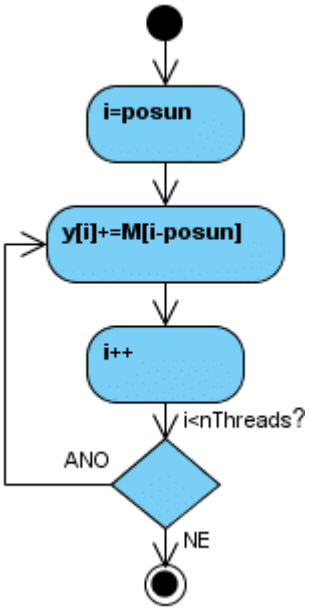
Počet kroků nutných pro výslednou matici PPS je logaritmus z `n` o základu 2. [4]

12.6.1 Popis programu

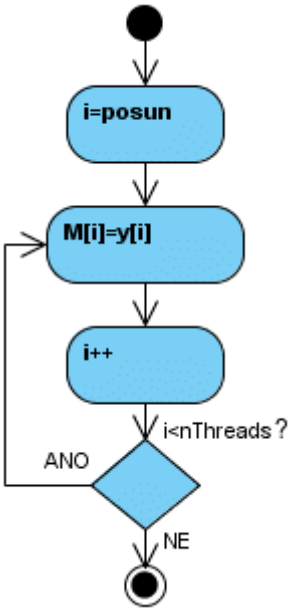


Obrázek 23 Diagram PPS na EREW PRAM

Z diagramu by nemuselo být zcela zřejmé, proč těsně za sebou následuje konec a start paralelní oblasti. Je to z důvodu synchronizace dat, kdy je potřeba před dalším paralelním zpracováním dat, aby byla potřebná data zpracována všemi vlákny.



Obrázek 24 Rozšiřující diagram k PPS: y[i]+=M[i-posun]



Obrázek 25 Rozšiřující diagram k PPS: M[i]=y[i]

Program samotný obsahuje dvě jednorozměrná pole `M` a `y` o stejných rozměrech `MAX`, což je počet prvků v poli a potažmo počet použitých procesorů. V první části se provede naplnění pole `M` hodnotami `M[i]=i` v interval `<0,MAX>`. V tomto případě šlo pouze o co

nejrychlejší způsob naplnění pole a v praxi jsou vstupní data zcela odlišná. Tato část sice využívá paralelního rozdělení mezi více vláken, ale není součástí měření, jelikož slouží pouze pro výše uvedené vygenerování dat a prezentace paralelního naplnění pole.

Poté se spustí samotné měření času běžícího algoritmu. Nejprve jsou paralelně zkopírována data do pole `y`. Je naprosto irelevantní, zdali se vygenerují data do `M` a jsou zkopírována do `y` či naopak

Do proměnné `limit` je uložena hodnota dvojkového logaritmu z množství vstupních dat, která udává počet opakování nutný pro dokončení výpočtu prefixového součtu a také pro definování posunutí prvku určeného k přičtení zpracovávané buňky pole (proměnná `posun`).

V další fázi jsou provedeny dva paralelní cykly `for`, které mají za účel zpracovat pole pro daný stupeň nastaveného posunu až do jeho max. hodnoty. První cyklus přičte k hodnotě `z[i]` hodnotu `y[i-posun]`. Zde je důvod, proč jsou potřeba dvě totožná pole. Došlo by totiž ke konfliktu, kdy v jednom vláknu by se použily hodnoty z oblasti, kterou ještě nezpracovalo vlákno druhé. Druhý cyklus slouží k synchronizaci obou polí, aby mohl pokračovat algoritmus opět od začátku své sekvenční smyčky. [7], [9]

```
 1  #include <omp.h>
 2  #include <stdlib.h>
 3  #include <stdio.h>
 4  #include <math.h>
 5  #include <sys/time.h>
 6
 7  #define MAX 8//pocet prvku ve vstupnim poli-urceni poctu nThreads
 8
 9  int main(int argc, char* argv[])
10  {
11    int i,j, limit, posun, nThreads=MAX, sum;
12    unsigned int *M=NULL, *y=NULL;// ukazatele na pole y a M
13    M = (unsigned int*)malloc(MAX*sizeof(unsigned int));
14    y = (unsigned int*)malloc(MAX*sizeof(unsigned int));
15
16    //definovani promennych pro citace tiku
17    timeval nStart, nEnd;
18
19    omp_set_num_threads(nThreads);
20    /*naplneni vstupniho pole M hodnotami 0 az MAX*/
21    #pragma omp parallel
22    {
23      #pragma omp for schedule(static)
24      for(i=0;i<nThreads;i++)
25      {
26        *(M+i)=i;
27      }
28    }
29    gettimeofday(&nStart, (struct timezone*)0);//start mereni
30    /*zkopirovani pole M do pole y vice procesory*/
```

```
31    #pragma omp parallel shared(M, y)
32    {
33      #pragma omp for
34      for (i=0;i<nThreads;i++)
35      {
36        *(y+i) = *(M+i); //y[i]=M[i];
37      }
38    }
39    /*pocet cyklu pro scitani prvku-logaritmus o zakladu 2*/
40    limit = ceil(log(nThreads)/log(2));
41
42    for (j=0;j<limit;j++)
43    {
44      posun = pow(2,j);/*definovani posuvu pro prvek, ktery se ma
      secist s aktivnim prvkem pole*/
46      #pragma omp parallel shared(M, y)
47      {
48        #pragma omp for schedule(static)
49        for (i=posun;i<nThreads;i++)
50        {
51          *(y+i) += *(M+i-posun); //y[i] += M[i-posun];
52        }
53        #pragma omp for schedule(static)
54        for (i=posun;i<nThreads;i++)
55        {
56          *(M+i) = *(y+i); //M[i] = y[i];
57        }
58      }
59    }
60    gettimeofday(&nEnd, (struct timezone*)0);//konec mereni
61    printf("pro %d procesoru byl dokoncen v case %.5lf sekund.",
      nThreads, (nEnd.tv_sec - nStart.tv_sec) + (nEnd.tv_usec -
      nStart.tv_usec)*1.0e-6);
62    return 0;
63  }
```

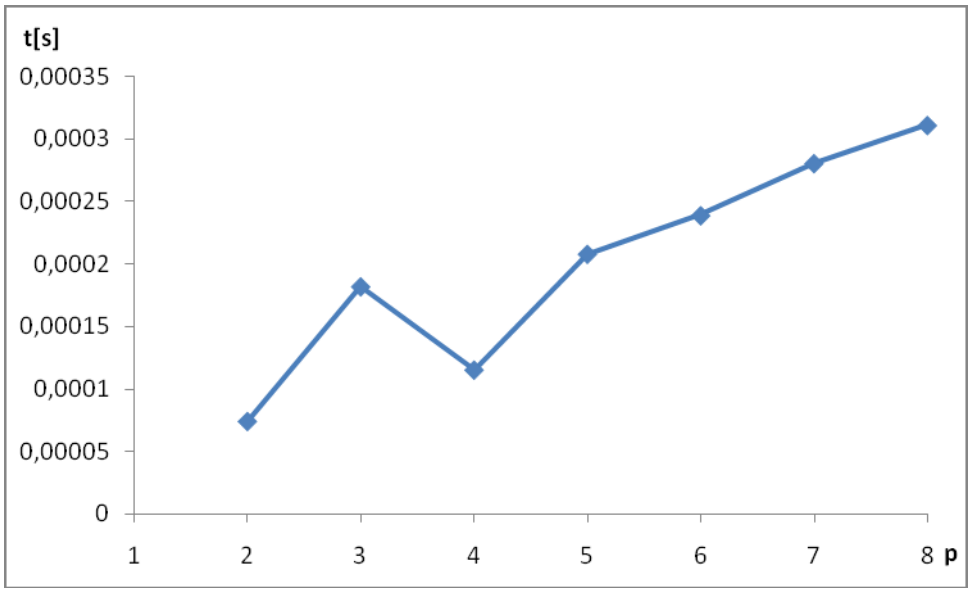
Jelikož je v práci použito i srovnání se sekvenčním prefixovým součtem, následuje jeho jádro, které využívá stejné proměnné jako výše uvedený zdrojový kód. Jeho algoritmu je uveden v teoretické části o PPS na EREW PRAM.

```
1     //naplneni pole M hodnotami
2     for(i=0;i<MAX;i++)
3     {
4       *(M+i)=i;
5     }
6
7     //sekvencni vypocet:
8     sum = 0;//suma vybraneho a predchazejiciho prvku je na pocatku=0
9
10    gettimeofday(&nStart, (struct timezone*)0);
11    for (i=0;i<MAX;i++)
12    {
13      sum += *(M+i); //sum += M[i];
14      *(y+i) = sum; //y[i] = sum;
15    }
16
17    gettimeofday(&nEnd, (struct timezone*)0);
18    printf("je %llu",*(y+MAX-1));
19    printf("vypocet byl dokoncen v case %.8lf sekund.", (nEnd.tv_sec
      - nStart.tv_sec) + (nEnd.tv_usec - nStart.tv_usec)*1.0e-6)
```

12.6.2 Výkonnost algoritmu

procesorů	rychlost zpracování PPS [s]
2	0,0000739
3	0,0001819
4	2,589361
5	0,000208
6	0,0002389
7	0,00028073
8	0,0003113

Tabulka 5 Rychlost výpočtu PPS



Obrázek 26 Rychlost výpočtu PPS

Rychlost sekvenční prefixového součtu je znatelně vyšší než PPS a je řádově rychlejší s časy stále rovny 0 i při zobrazení řádu 10^{-6}.

Paměťová náročnost algoritmu je:

- `Pole y: MAX * 2 Byte`
- `Pole M: MAX * 2 Byte`
- `Proměnné i, j, limit, posun, nThreads, sum: 6 * 2 * Byte`
- Paměťová náročnost pro 8 prvků: `8 * (2 + 2) + 6 * 2 = 44B.`

Asymptotické funkce:

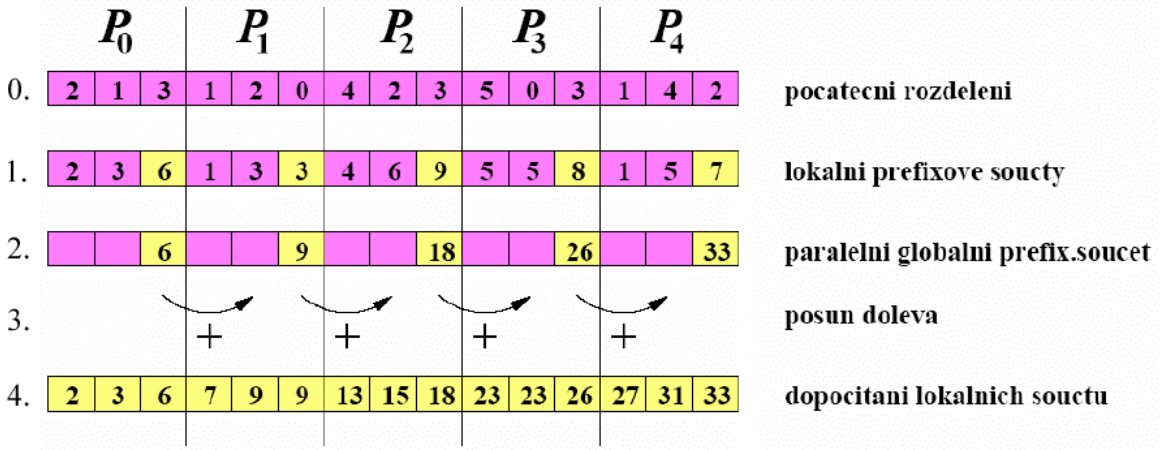
- $T(n,p) = O\left(\frac{n}{p} + \log p\right)$

- $C(n, p) = O\left(n + p \log p\right)$

12.7 Škálovatelný PPS

Tento algoritmus vychází ze spojení PPS a sekvenčního PS. Má možnost škálovatelnosti, a lze jej tedy použít na velké množství vstupních dat.

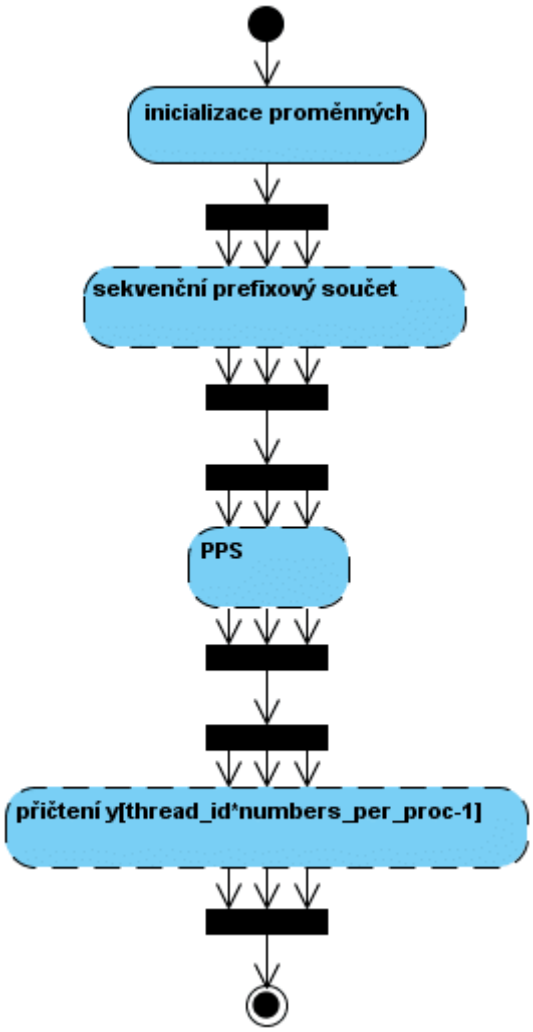
Jak algoritmus pracuje, je patrné z následujícího obrázku.



Obrázek 27 Princip škálovatelného PPS [4]

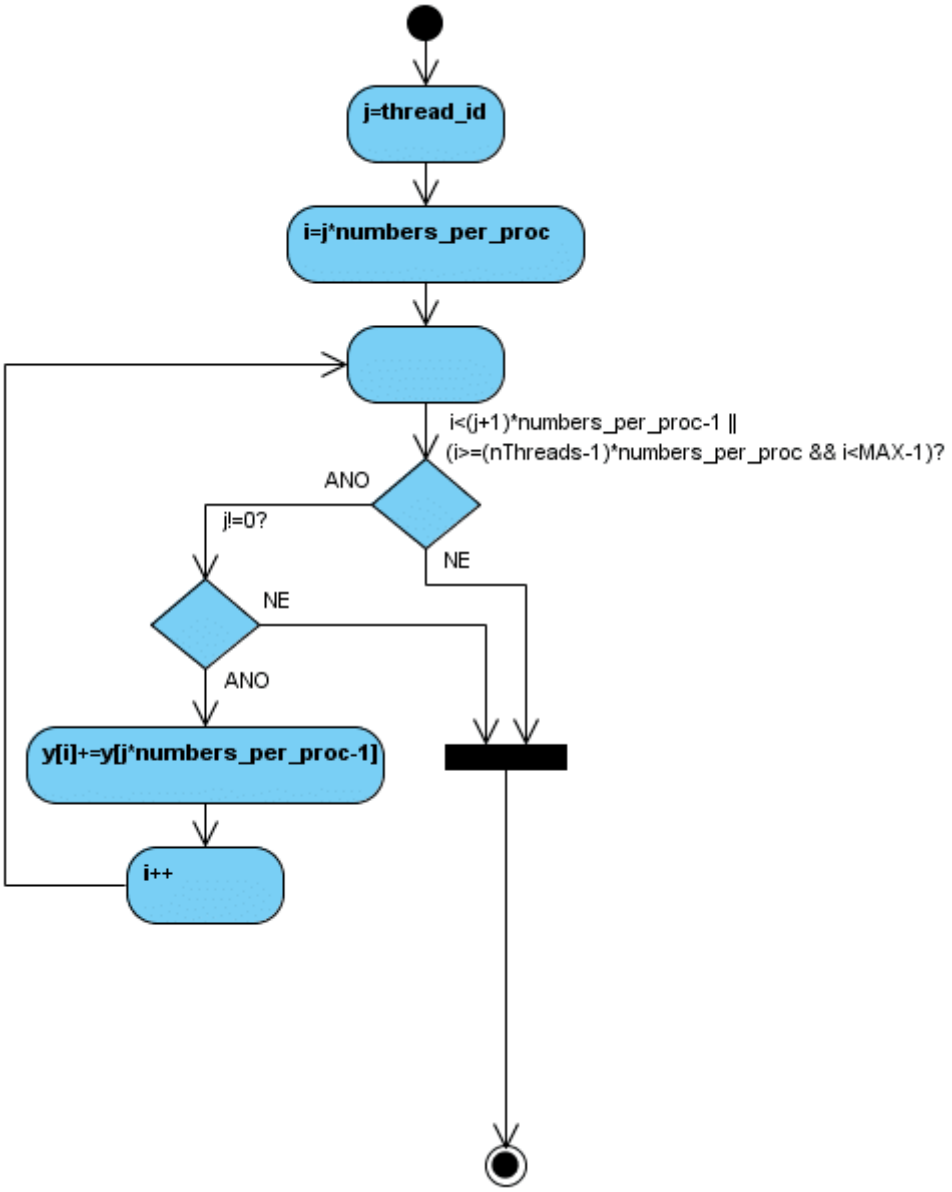
Princip spočívá v tom, že vstupní pole se rozdělí rovnoměrně na počet částí odpovídajících počtu procesorů (krok 0) a u těchto částí se provede sekvenční prefixový součet, který je na menších polích velmi rychlý (krok 1). V dalším kroku je proveden PPS posledních čísel ze skupiny přiřazené jednotlivým procesorům na EREW PRAM (krok 2) a tento se přičte k prvkům pole procesoru s id vyšším o 1(krok 3). Výsledkem je pole, na kterém byl vykonán prefixový součet. [4], [10]

12.7.1 Popis programu



Obrázek 28 Diagram škálovatelného PPS

Jelikož jsou již algoritmy sekvenčního prefixového součtu i PPS na EREW PRAM mezi vývojovými diagramy uvedeny (Obrázek 21, Obrázek 23), nebudou už dále rozkresleny. Jejich modifikace je popsána dále v této kapitole dostatečným způsobem.



Obrázek 29 Rozšiřující diagram ke Škálovatelnému PPS: přičtení y[thread_id*numbers_per_proc-1]

V programu jsou inicializována dvě pole: `y` je vstupní pole se všemi čísli a pole `z` slouží pro výpočet pomocí PPS, viz následující text.

Pro rovnoměrné rozdělení zátěže mezi procesory je použita proměnná `numbers_per_proc`, která obsahuje vydělený počet prvků ve vstupním poli počtem procesorů a zaokrouhlený dolů – pokud nebude počet čísel dělitelný beze zbytku počtem procesorů, zbude na poslední vlákno výpočet množiny čísel v rozsahu `<MAX-(p-1)*numbers_per_proc,MAX>`.

Následně dojde k vygenerování čísel do vstupního pole se vzrůstající hodnotou `<0,MAX>`. Od této chvíle začíná počítání času samotného algoritmu, který má všechny potřebné informace a data pro svoji práci.

Nejprve se provede sekvenční prefixový součet každým vláknem nad množinou přidělených dat, která je v rozsahu `<thread_id*numbers_per_proc,(thread_id+1)*numbers_per_proc - 1>`. Algoritmus zde využívá rychlosti sekvenčního prefixového součtu uvedeného v předcházejícím měření.

Další část je téměř totožná s PPS na EREW PRAM kromě nutnosti brát ohled na poslední množinu z čísel, která nemusí mít poslední prvek na pozici `p*numbers_per_proc`. Ten je jednoznačně určen celkovým počtem prvků ve vstupním poli `MAX-1`. Druhým rozdílem je zápis do a čtení z pole `y`, kdy je potřebné zapsat výsledné hodnoty do nadefinovaných proměnných pro práci se vstupním polem. Místo výrazu `*(y+i-posun)` je nutné použít konstrukci `*(y+(i-posun+1)*numbers_per_proc-1)`.

Na závěr každé vlákno `thread_id` přičte ke všem číslům v přidělené množině číslo vypočtené pomocí PPS, které je uložené na pozici `(thread_id-1)*numbers_per_proc-1`.

```
 1  #include <omp.h>
 2  #include <stdlib.h>
 3  #include <stdio.h>
 4  #include <math.h>
 5  #include <sys/time.h>
 6
 7  #define MAX 100000000
 8
 9  int main(int argc, char* argv[])
10  {
11    long posun,limit,i;
12    int j,nThreads=2;
13    unsigned long long *y = NULL, *z = NULL, sum;
14    unsigned long numbers_per_proc;
15
16    y = (unsigned long long*)malloc(MAX*sizeof(unsigned long long));
17    z = (unsigned long long*)malloc(nThreads*sizeof(unsigned long
      long));
18
19    timeval nStart, nEnd;
20
21    numbers_per_proc = floor(MAX/nThreads);//pocet cisel zpracovanych
      jednim procesorem
22
23    omp_set_num_threads(nThreads);
24
25    /*naplneni vstupniho pole y hodnotami 0 az MAX*/
26    #pragma omp parallel
27    {
28      #pragma omp for schedule(static)
29      for(i=0;i<MAX;i++)
30      {
```

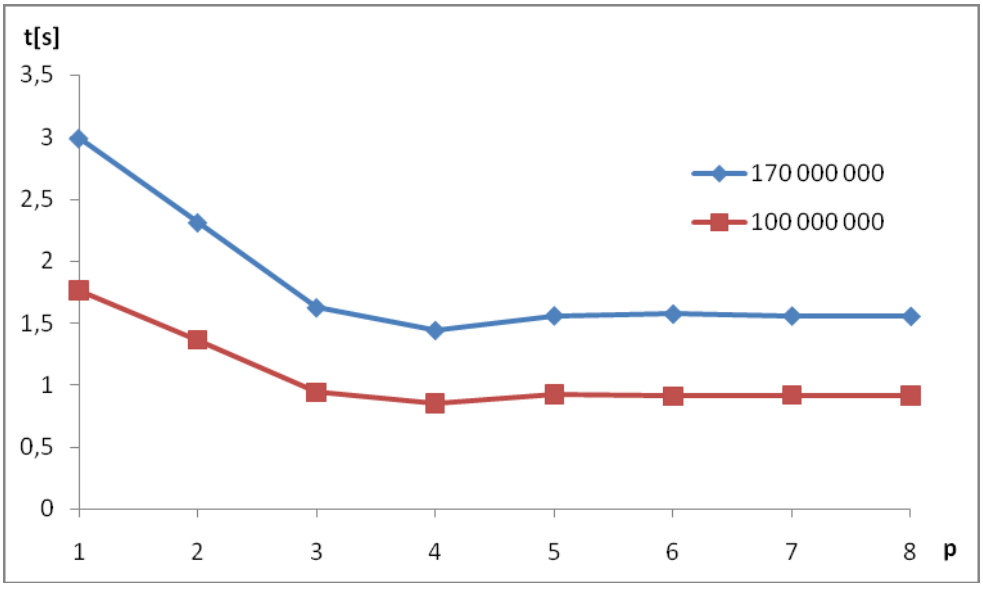
```
31          *(y+i)=i;
32        }
33      }
34
35      gettimeofday(&nStart, (struct timezone*)0);
36
37      /*sekvencni prefixovy soucet pro oblast zpracovavanou danym
        procesorem*/
38      #pragma omp parallel shared(y,z,nThreads,numbers_per_proc)
        private(i,j,sum)
39      {
40        j = omp_get_thread_num();//do promenne j se ulozi id vlakna
41        sum=0;
42        for(i=j*numbers_per_proc;(i<(j+1)*numbers_per_proc) ||
          ((i>=nThreads*numbers_per_proc) && (i<MAX));i++)
43        {
44          sum += *(y+i);
45          *(y+i) = sum;
46        }
47        *(z+j)=sum;//ulozi posl. hodnotu pref. souctu do zvlastniho pole
48      }
49
50      /*pocet cyklu pro scitani prvku-log o zakladu 2*/
51
52      /*PPS*/
53      limit = ceil(log(nThreads)/log(2));
54      for (j=0;j<limit;j++)
55      {
56        posun=pow(2,j);
57        #pragma omp parallel shared(y,z,numbers_per_proc,nThreads)
58        {
59          #pragma omp for schedule(static)
60          for (i=posun;i<nThreads;i++)
61          {
62            *(z+i) += *(y+(i-posun+1)*numbers_per_proc-1);
63          }
64          #pragma omp for schedule(static)
65          for (i=posun;i<nThreads;i++)
66          {
67            /*u posledniho vlakna neni jiste, zda je stejny pocet bunek
              jako v ostatnich*/
68            if(i == nThreads-1)
69            {
70              *(y+MAX-1) = *(z+i);
71            }
72            else
73            {
74              *(y+(i+1)*numbers_per_proc-1) = *(z+i);
75            }
76          }
77        }
78      }
79
80      /*pricteni prefix. souctu predesleho vlakna k cislum aktivniho
        vlakna*/
81      #pragma omp parallel shared(y,numbers_per_proc,nThreads)
        private(i,j)
82      {
83        j = omp_get_thread_num();
84        for(i=j*numbers_per_proc ; ( i<(j+1)*numbers_per_proc-1 ) ||
          (i>=(nThreads-1)*numbers_per_proc) && (i<MAX-1) ) ; i++)
85        {
86          if(j!=0)
87          {
```

```
88            *(y+i) += *(y+j*numbers_per_proc-1);
89          }
90        }
91      }
92
93      gettimeofday(&nEnd, (struct timezone*)0);
94      printf("Vypocet pro %d procesoru byl dokoncen v case %.5lf
        sekund.", nThreads, (nEnd.tv_sec - nStart.tv_sec) + (nEnd.tv_usec -
        nStart.tv_usec)*1.0e-6);
95      return 0;
96    }
```

12.7.2 Výkonnost algoritmu

procesorů	Škálovatelný PPS [s]	
	100 000 000 prvků	170 000 000 prvků
1	1,760859	2,991018
2	1,362655	2,312473
3	0,939763	1,626911
4	0,85089	1,444656
5	0,924995	1,56237
6	0,909227	1,576588
7	0,917996	1,561414
8	0,912339	1,557914
sekvenční	1,09216	1,834125

Tabulka 6 Rychlost výpočtu škálovatelného PPS



Obrázek 30 Rychlost výpočtu škálovatelného PPS

Paměťová náročnost algoritmu je:

- Pole y: MAX * 8 Byte
- Pole z: nThreads * 8 Byte

- `Proměnné posun, limit, number_per_proc: 3 * 4 Byte`
- `Proměnné ve vláknech i, j, sum: 8 * (4 + 2 + 8) Byte`
- `Proměnná nThreads: 2 Byte`
- Paměťová náročnost pro 170 000 000 prvků: `170 000 000 * 8 + 8 * 8 + 3 * 4 + 8 * 14 = 1 296,997MB.`

Asymptotické funkce:

- $T(n,p) = O\left(\frac{n}{p} + \log p\right)$
- $C(n,p) = O\left(n + p\log p\right)$

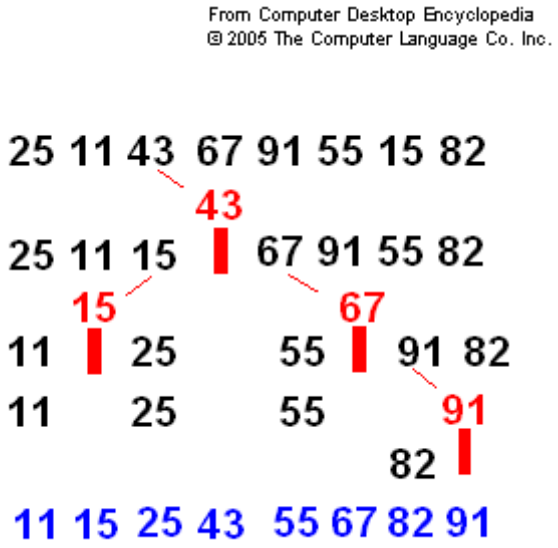
12.8 QuickSort

QuickSort je rekurzivní třídící algoritmus patřící rovněž do skupiny tzv. Divide-and-Conquer (Rozděl-a-panuj) algoritmů. Je to jeden z nejvíce používaných a rovněž jeden z nejrychlejších třídících algoritmů.

Při zavolání rekurzivní funkce se vybere určitým způsobem, popsaným níže, prvek v poli zvaný p*ivot*, od kterého jsou rozděleny dvě skupiny čísel - jedna větší než *pivot* a druhá menší než *pivot*. Když dojde k seřazení prvků vůči *pivotovi*, zavolá funkce rekurzivně sama sebe s intervaly pro setřídění prvků menších než *pivot* a větších než *pivot*. Tímto způsobem funkce pokračuje, dokud nejsou všechny prvky seřazeny. [11]

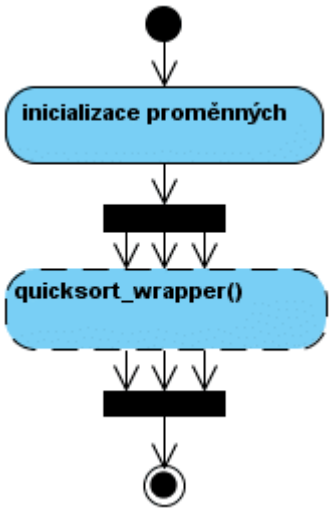
Výběr *pivota*:

- První prvek – je vybrán prvek na předem určené pozici. Např. konec nebo střed pole.
- Náhodný prvek – v praxi se často používá generátor pseudonáhodných čísel.
- Metoda mediánu tří – náhodně se vyberou tři nebo jiný počet veličin a určí se z nich medián, který je označen za *pivota*.

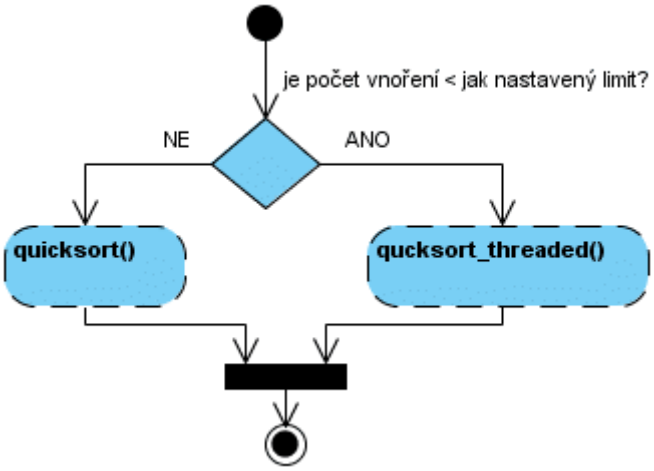


Obrázek 31 Princip řazení QuickSortu [12]

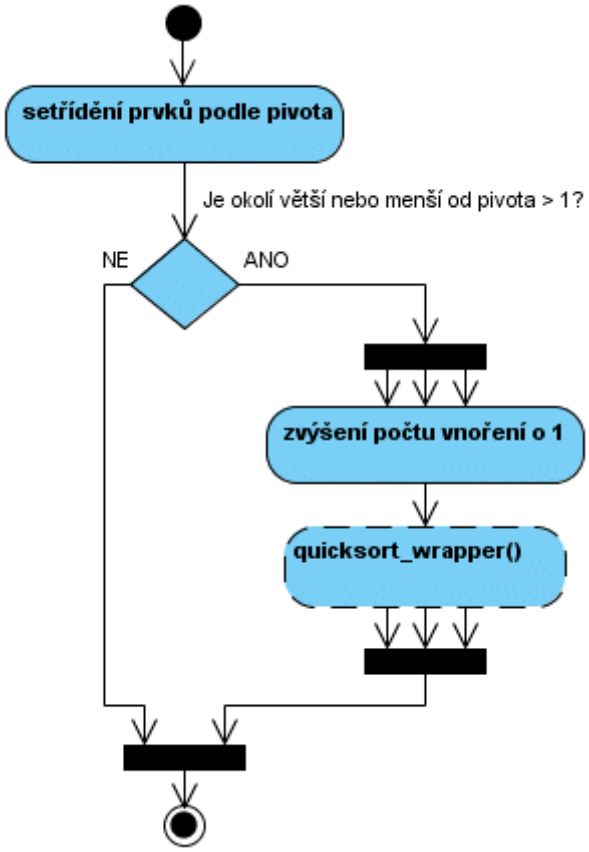
12.8.1 Popis programu



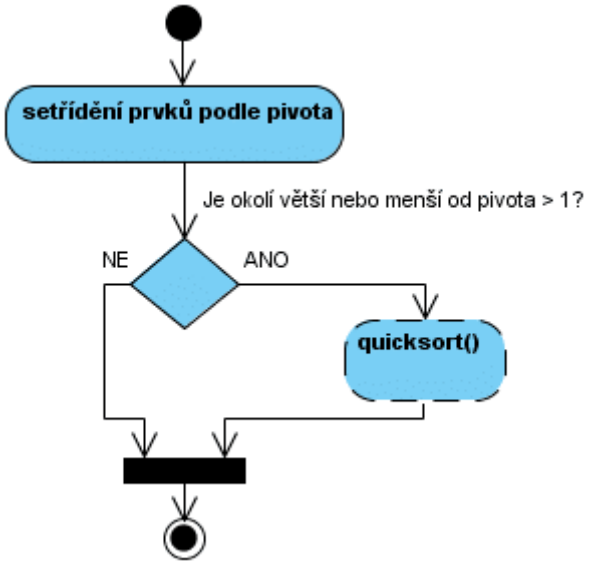
Obrázek 32 Diagram algoritmu Quicksort



Obrázek 33 Rozšiřující diagram ke Quicksortu: Quicksort_wrapper()



Obrázek 34 Rozšiřující diagram ke Quicksort_wrapper(): Quicksort_threaded()



Obrázek 35 Rozšiřující diagram ke Quicksort_wrapper(): Quicksort()

U tohoto algoritmu bylo nutné použít zvláštní kompilátor od firmy Intel s jeho vlastními OpenMP *pragmami*, jelikož OpenMP 2.5, které je implementováno v překladači GCC 4.2.3 nepodporuje rekurzivní volání funkcí. Tato možnost byla přidána až ve verzi

OpenMP 3.0, která zatím není v žádném překladači implementována. Jedná se konkrétně o *pragmu taskq* a *task.*[13]

Pragma taskq vytvoří prostředí pro inicializování úloh, které jsou zpracovávány jednotlivými vlákny. Při zavolání *taskq* se k němu přiřazený blok zdrojového kódu prování pouze jedním vláknem a ostatní volná vlákna čekají, dokud se nedokončí všechny instrukce uvedené v daném bloku.

Pragma task vytváří tzv. úlohu, která je vykonána v daném prostředí. Uvnitř této úlohy může být definováno opět prostředí pro vytváření úloh (*pragma taskq)* a takto lze vnořovat libovolný počet úloh do sebe. Všechna tato prostředí existují do ukončení všech úloh, které obsahují.

Konstanta `TASKQ_DEPTH` určuje, do jaké hloubky se bude možno vnořovat pomocí `pragma taskq`. Je nutné určit pokusnými měřeními výkonnosti algoritmu optimální velikost vnoření, jelikož při malé hloubce může dojít ke zbytečně nerovnoměrnému rozložení vytížení procesorů a naopak při velké hloubce rostou časy potřebné na režii správy paralelní oblasti.

V programu jsou použity 3 funkce. `Quicksort_wraper()` pro určení, zda se má provádět sekvenční (`Quicksort()`) nebo paralelní QuickSort (`Quicksort_threaded()`). Druhou a třetí funkcí jsou samotné QuickSorty. Jak je na obou verzích QuickSortu vidět, jsou téměř totožné, kromě definování prostředí pro úlohy u paralelního QuickSortu. Při požadavku algoritmu o rekurzivní výpočet paralelním QuickSortem je potřeba nejprve ověřit, zda není překročen počet vnoření pomocí funkce `Quicksort_wraper()`.

Poslední část programu, která zůstala po vývojových diagramech a výše zmíněném popisu možná ne zcela objasněna, je funkční princip samotného algoritmu *QuickSort*. Do funkce vstupuje vždy celé pole, které má oblast určenou pro zpracování danou minimem a maximem. Následně se určí prostřední prvek `x` z rozsahu určeného pro daný stupeň *QuickSortu*. Tento prvek je zvolen *pivotem* a následné třídění větších a menších čísel probíhá na základě srovnávání s ním. Pokud je při ukončení *QuickSortu* nalevo či napravo od *pivota* více jak jeden prvek, je tato oblast zpracována v dalším rekurzivním QuickSortu.

```
1  #include <omp.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <sys/time.h>
5
6  #define MAX 15000000
```

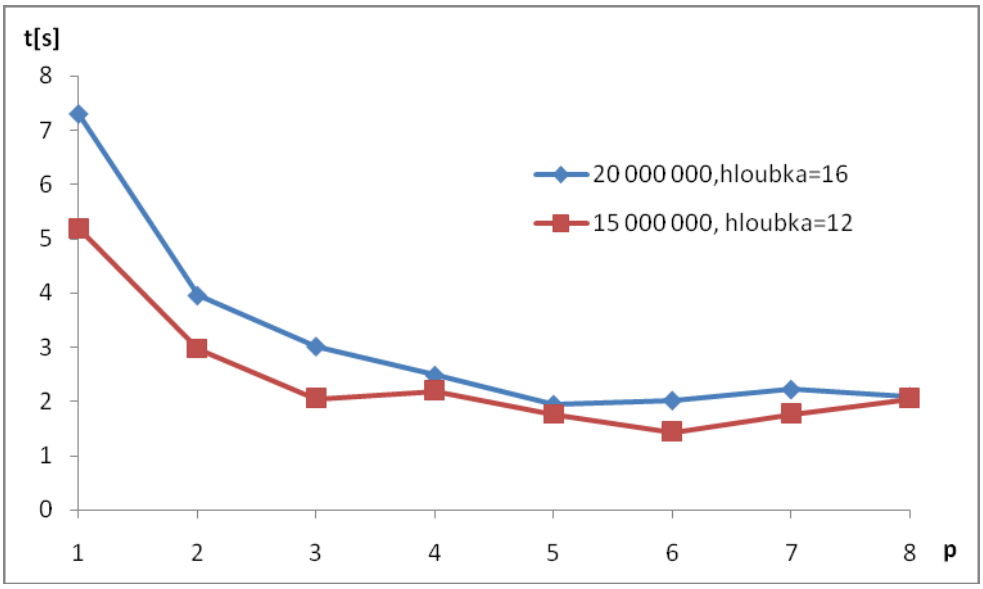
```
 7  #define TASKQ_DEPTH 16
 8
 9  void quicksort_threaded (int *a,int low,int high,int taskq_depth);
10  void quicksort_wrapper (int *a,int low,int high,int taskq_depth);
11  void quicksort (int *a,int low,int high);
12
13  //sekvencni quicksort
14  void quicksort (int *a,int low,int high)
15  {
16    //low=spodni index, high=horni index, x=pivot
17    int i=low, j=high, temp;
18    int x=*(a+((low+high)/2));
19
20    //preskupeni prvku podle pivota
21    do
22    {
23      while (*(a+i)<x) i++;
24      while (*(a+j)>x) j--;
25      if (i<=j)
26      {
27        temp=*(a+i);
28        *(a+i)=*(a+j);
29        *(a+j)=temp;
30        i++; j--;
31      }
32    }
33    while (i<=j);
34
35    //rekurzivni volani quicksortu
36    if (low < j)
37    {
38      quicksort(a,low,j);
39    }
40    if (i < high)
41    {
42      quicksort(a,i,high);
43    }
44  }
45
46  //paralelni verze quicksortu
47  void quicksort_threaded (int *a,int low,int high,int taskq_depth)
48  {
49    //low=spodni index, high=horni index, x=pivot
50    int i=low, j=high, temp;
51    int x;
52    #pragma intel omp taskq
53    {
54      x=*(a+((low + high)/2));
55      //preskupeni prvku podle pivota
56      do
57      {
58        while (*(a+i)<x) i++;
59        while (*(a+j)>x) j--;
60        if (i<=j)
61        {
62          temp=*(a+i);
63          *(a+i)=*(a+j);
64          *(a+j)=temp;
65          i++;
66          j--;
67        }
68      }
69      while (i <= j);
```

```
70      /*rekurzivni volani funkce quicksort_wraper se zvysenim vnoreni
        o 1*/
71      #pragma intel omp task
72      {
73        if (low < j) quicksort_wrapper(a,low,j,taskq_depth+1);
74      }
75      #pragma intel omp task
76      {
77        if (i < high) quicksort_wrapper(a,i,high,taskq_depth+1);
78      }
79    }
80  }
81
82  /*funkce rozhodujici, zda se provede sekvencni nebo paralelni
    quicksort v zavislosti na hloubce vnoreni*/
83  void quicksort_wrapper (int *a,int low,int high,int taskq_depth)
84  {
85    if(taskq_depth < TASKQ_DEPTH)
86    {
87      quicksort_threaded(a, low, high, taskq_depth+1);
88    }
89    else
90    {
91      quicksort(a, low, high);
92    }
93  }
94
95  int main(int argc, char *argv[])
96  {
97    int i,nThreads=8;
98    int *a=NULL;
99    a = (int*)malloc(MAX*sizeof(int));
100
101   timeval nStart, nEnd;
102
103   omp_set_num_threads(nThreads);
104
105   //vygenerovani pseudonahodnych cisel pro setrideni do pole a
106   for(i=0;i<MAX;i++)
107   {
108     *(a+i)=rand()%32767;
109   }
110
111   gettimeofday(&nStart, (struct timezone*)0);
112
113   /*inicializace paralelni oblasti a volani quicksort_wraper s
      nulovou hodnotou vnoreni*/
114   #pragma intel omp parallel taskq num_threads (nThreads)
115   {
116     #pragma intel omp task
117     quicksort_wrapper(a, 0, MAX-1, 0);
118   }
119
120   gettimeofday(&nEnd, (struct timezone*)0);
121   printf("%ld procesory byl dokoncen v case %.5lf sekund.",
      nThreads, (nEnd.tv_sec - nStart.tv_sec) + (nEnd.tv_usec -
      nStart.tv_usec)*1.0e-6);
122   return 0;
123 }
```

12.8.2 Výkonnost algoritmu

procesorů	Paralelní QuickSort [s]	
	15 000 000 prvků	20 000 000 prvků
1	5,192002	7,294932
2	2,976449	3,950805
3	2,062061	3,012374
4	2,201043	2,479804
5	1,768042	1,943757
6	1,439189	2,018647
7	1,770192	2,220628
8	2,055185	2,090782
sekvenční	4,413123	5,933446

Tabulka 7 Rychlost třídění paralelním QuickSortem



Obrázek 36 Rychlost třídění paralelním QuickSortem

Paměťová náročnost algoritmu je:

- `Pole y: MAX * 8 Byte`
- `Pole z: nThreads * 8 Byte`
- `Proměnné ve vláknech low, hight, temp, x: 8 * 4 * 2 Byte`
- Paměťová náročnost pro 170 000 000 prvků: `170 000 000 * 8 + 8 * 8 + 8 * 4 * 2 = 1 296,997MB.`

Asymptotické funkce:

- $T(n,p) = O(n^2)$

- $C(n, p) = O\left(pn^2\right)$

ZÁVĚR

Paralelní programování je velmi zajímavou problematikou s reálnou možností praktického využití, zvláště v dnešní době dostupných vícejádrových PC i pro domácnosti. Pozitivní věcí je, že některé vhodné algoritmy lze díky paralelizaci výrazně zrychlit. Odvrácenou stranou bohužel je, že ne každý algoritmus může být převeden na zpracování více vlákny a i pokud je to možné, zdaleka ne u všech paralelních algoritmů dochází se vzrůstajícím počtem dostupných procesorů/jader také k (lineárnímu) nárůstu výkonu algoritmu.

V této práci byly shrnuty algoritmy jak výkonné, které svojí paralelizací a navyšováním počtu využitých procesorů/jader získávají na rychlosti, tak rovněž algoritmy, na které nemá testovaná architektura osobního vícejádrového počítače prakticky žádný vliv na zrychlení, ba naopak, dosahuje i opačného efektu, a to zpomalení. Jedná se zejména o algoritmy, které předpokládají velké množství dostupných procesorů/jader (řádově desítky až stovky) a hlavním důvodem jejich nízké rychlosti je režie potřebná k práci s více jádry, zajištění komunikace a synchronizace mezi nimi a mezi pamětí. Tyto důvody jsou obecně v paralelních algoritmech, které byly v této práci na testovacím serveru zkoušeny, zdrojem výrazných zpomalení, i když jejich teoretická rychlost je řádově vyšší, než u jejich sekvenčních variant.

Jedním z nejjednodušších algoritmů je násobení matic. U něho je možné pozorovat s nárůstem počtu jader výrazné zrychlení. Už od počtu dvou vláken je čas potřebný pro vynásobení matic výrazně kratší. Při plném využití osmi jader na testovaném serveru je výsledný čas zlomkový proti sekvenčnímu násobení.

Na opačném konci výkonnosti je PPS na EREW PRAM, u kterého je efektivita téměř nulová, vzhledem k tomu, že většinu času spotřebuje režie algoritmu a výpočetní síla se zde nemůže projevit. Tento algoritmus je stavěn na počítače s velkým množstvím procesorů, a nikoliv pro osobní vícejádrové počítače. Je však dobrým základem pro kombinaci sekvenčního PS a PPS na EREW PRAM v podobě škálovatelného PPS. Zde je nárůst výkonu znatelný, avšak pouze do počtu 4 procesorů. Poté už je algoritmus na testovaném stroji bez jakéhokoliv navýšení výkonu. Avšak při dnešních čtyřjádrových procesorech v domácích PC by to byl optimální algoritmus na aplikaci pro osobní počítač.

Posledním testovaným algoritmem byl paralelní QuickSort. Jeho zprovoznění provázelo mnoho komplikací, protože se dlouhou dobu nedařilo nalézt žádnou aplikaci ani způsob,

jak efektivně využívat více vláken pro rekurzivní funkce. Vše vyřešil kompilátor od firmy Intel, který má mnoho nadstandardních funkcí oproti verzím OpenMP 2.5 nebo OpenMP 2.0. Funkce potřebné v QuickSortu jsou dostupné ve verzi OpenMP 3.0, ale ten bohužel zatím není součástí překladačů.

CONCLUSION

Parallel programming offers possibilities of its practical usage, especially nowadays when multi-core PCs are available for ordinary people. The positive point is the fact that thanks to parallelization some convenient algorithms can be accelerated. The disadvantage is that not all algorithms can be transferred into processing by more threads, and if so, not with all parallel algorithms the power increase goes hand in hand with the increasing number of processors / cores.

In the thesis I summarized both power algorithms, which due to their parallelization and increasing of the number of processors / cores gain speed, and algorithms in which no practical influnce of the tested architecture of the multi-core computer on their speed was proved. Futhermore, the effect on the latter algorithms can be contradictive and they might be slowered down. This can be true for those algorithms which expect a great number of available processors / cores and the main reason of their slow speed is in the cost and providing communication and synchronization between them and the memory. These reasons are the main source of deceleration of the parallel algorithms although their theoretical speed is higher than with their sequence varieties.

One of the easiest algorithms is multiplying matrices in which significant deceleration with the increase of the number of cores can be observed. The time necessary for multiplying matrices is significantly shorter starting from the number of two threads. If eight cores on the testing server are used, the final time is incomparably shorter than with sequence multiplying.

On the oposite scale of power is PPS on EREW PRAM for which the efectivity is almost negligible because the majority of time is used up by the cost of the algorithm and the computing power cannot apply here. This algorithm is supposed to be used with computers containing a large number of processors, not for personal multi-core computers. However, it is a good base for combining sequence PS and PPS on EREW PRAM – scalable PPS. Here, the increase in power is significant, but only up to the number of four processors. From then onwards, the algorithm on the tested server is with no acceleration of power. With four-core processors, which are used in PCs nowadays, it would be an optimal algorithm, though.

The last tested algorithm was parallel QuickSort. Its beginnings was followed by many problems due to the fact that for a long time no aplication or way of effective usage of more threads for recursive functions could not be found. The solution has been brougt in by the compiler by Intel which has many quality functions compared to versions OpenMP 2.5 or OpenMP 2.0. Functions needed for QuickSort are available in the version OpenMP 3.0, which is unfortunately not part of nowadays compilers yet.

SEZNAM POUŽITÉ LITERATURY

[1] Brian W. Kernighan, Dennis. M. Ritchie, Programovací jazyk C, Computer Press, 2006, ISBN 80-251-0897-X.

[2] Stephen Prata, Mistrovství v C++, 3. vydání, Computer Press, 2007, ISBN 978-80-251-1749-1.

[3] Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J.,Menon, R., Parallel Programming in OpenMP, Morgan Kaufmann Publishersm, 2001, ISBN 1-55860-670-8.

[4] Prof. P. Tvrdí, Paralelní systémy a algoritmy, Nakladatelství ČVUT, 2006, ISBN 80-01-03565-6

[5] J. Keller, Ch. W. Keßler, J. L. Träff, Practical PRAM Programming, John Wiley & Sons, Inc., 2001, ISBN 0-471-35351-5

[6] Jesse Liberty, Naučte se C++ za 21 dní, Computer Press, 2002, ISBN 80-7226-774-4

[7] MSDN:OpenMP in Visual C++ [online]. Dostupný z WWW: http://msdn2.microsoft.com/en-us/library/tt15eb9t.aspx

[8] OpenMP [online]. Dostupný z WWW: http://www.openmp.org/blog/

[9] OpenMP [online]. Dostupný z WWW: https://computing.llnl.gov/tutorials/openMP/

[10] Ing. Michal Bližňák Ph.D., Prezentace do předmětu Paralelní procesy a programování

[11] Wikipedia [online]. Dostupná z WWW: http://www.wikipedia.com

[12] PC MAG Network [online]. Dostupný z WWW: http://www.pcmagnetwork.com/

[13] Intel® C++ Compiler for Linux* Systems User's Guide [online]. Dostupný z WWW: ftp://download.intel.com/support/performancetools/c/linux/v8/c_ug_lnx.pdf

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

API	Application programming interface – rozhraní pro programování aplikací.
PC	Personal Computer – osobní počítač.
ID	Identifikační číslo.
IP	Internetový protokol.
OpenMP	Open Multi-Processing.
FTP	File transfer protokol – protokol pro přenos souborů na internetu.
$\forall$	Definice: pro každý prvek.
$\exists$	Definice: existuje alespoň jeden prvek.
N^+	Definice: množina kladných přirozených čísel.
R^+	Definice: množina kladných reálných čísel.
$\rightarrow$	Definice: implikace, přiřazení.

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM PŘÍLOH

PŘÍLOHA P I: CD-ROM

Přiložené CD obsahuje tuto práci v elektronické podobě (formát PDF) a všechny zdrojové kódy, na kterých byly prováděny testy.