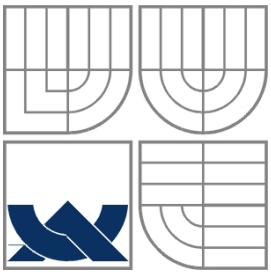


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA STROJNÍHO INŽENÝRSTVÍ
ÚSTAV AUTOMATIZACE A INFORMATIKY

FACULTY OF MECHANICAL ENGINEERING
INSTITUTE OF AUTOMATION AND COMPUTER SCIENCE

MOŽNOSTI REALIZACE PARALELNĚ ZPRACOVÁVANÝCH ÚLOH V PROGRAMOVACÍCH JAZYCÍCH

POSSIBILITIES OF IMPLEMENTATION PARALLEL TASKS IN PROGRAMMING LANGUAGES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **VÁCLAV ZEJDA**
AUTHOR

VEDOUCÍ PRÁCE **Ing. PAVEL HOUŠKA, Ph.D.**
SUPERVISOR

BRNO 2009

ZADÁNÍ ZÁVĚREČNÉ PRÁCE

(na tomto místě je vloženo zadání práce)

LICENČNÍ SMLOUVA

(na tomto místě je vložen podepsaný list formuláře licenčního ujednání)

ABSTRAKT

Tato práce se zabývá problematikou realizace paralelně zpracovávaných úloh v různých programovacích prostředích.

První část je věnována obecnému úvodu do paralelizace. Uvádí zejména kdy se vyplatí paralelizovat, jaké typy paralelizace se používají a rozdíly v použité architektuře.

V dalších částech jsou již popsány jednotlivé způsoby a prostředky používané pro tvorbu aplikací s podporou paralelismu. Od doplňků pro klasické programovací rozhranní (C/C++, .NET), přes tvorbu aplikací v grafických vývojových prostředích až po rozhranní využívající vedle procesoru počítače (CPU) i procesory grafických karet (GPU).

ABSTRACT

This work deals with the problem of implementation of parallel tasks in various programming environments.

First part of the work follows the basics of parallelisation. It features especially when the parallelisation is suitable, what kinds of parallelisation are used and differences between architecture of various systems.

Following parts describes various techniques in developing applications with the parallelisation support. From parallelisation support for common programming languages to graphics development tools and to environments that uses graphic processing unit (GPU) in combination with central processing unit (CPU).

KLÍČOVÁ SLOVA

Datový paralelismus, úlohový paralelismus, OpenMP, MPI, .NET, CPU, GPU

KEYWORDS

Data parallelism, Task parallelism, OpenMP, MPI, .NET, CPU, GPU

PODĚKOVÁNÍ

Děkuji panu Ing. Pavlu Houškovi Ph.D. za celkové vedení a směřování práce a čas strávený na konzultacích.

Obsah

1 ÚVOD

Paralelní zpracování úloh bylo vždy vhodnou metodou ke zvýšení výpočetního výkonu. Pořízení jednoho vysoce výkonného počítače pro zpracování náročné operace bylo obvykle velmi nákladnou záležitostí. Proto bylo výhodné takto náročnou operaci rozdělit na operace jednoduší a tyto následně zpracovávat na větším množství méně výkonných počítačů.

Dalším důvodem pro využívání paralelizace jsou fyzikální limity spojené se zvyšováním výkonu klasických procesorů. Toho se dosahovalo zejména neustálým zvyšováním pracovní frekvence a počtu tranzistorů integrovaných na procesoru. Jak se ukázalo již před několika lety, zvyšování pracovní frekvence je možné pouze k určité hranici, kdy začne neúměrně stoupat spotřeba procesoru a dochází k jeho nadměrnému ohřívání. Proto se výrobci procesorů vydali cestou zvyšování počtu procesorů v jednom obvodu - více jádrové procesory.

V dřívější době se v případě paralelizace jednalo spíše o záležitost vědeckého a výzkumného rázu a normální uživatel s ní nepřicházel běžně do styku. Rozšíření víceprocesorových počítačů a počítačů s více jádrovými procesory zpřístupnilo celou oblast široké veřejnosti. Více výpočetních jednotek však samo o sobě nezaručí, že stávající aplikace využijí plně jejich potenciál. Aby toto nastalo, musí se již během návrhu aplikace počítat s možností paralelního zpracování a podporu paralelizace do aplikace implementovat.

V klasickém případě se program vykonává ve formě vlákna na jednom procesoru. Ke zpracování úlohy je využita pouze jedna jednotka i v případě, že je počítač vybaven více procesory nebo jedním více jádrovým procesorem. Místo, aby ostatní jednotky v době zpracování aktuální úlohy nic nedělaly nebo vykonávali vlákna jiných programů, je možné jich využít ke zpracování programu s podporou paralelizace.

Z historického hlediska není programování paralelních aplikací žádnou novinkou. Od vydání rozhranní Win32 mají programátoři a vývojáři k dispozici mnoho prostředků pro programování aplikací zpracovávaných ve více vláknech. Vytváření vláken umožňuje například programovací prostředí Borland Delphi nebo Visual Studio od firmy Microsoft a to i ve starších verzích. Vlákna však bylo zapotřebí manuálně vytvářet, rušit, synchronizovat apod. Nevýhodou tohoto přístupu jsou nemalé požadavky kladené na programátora.

Tato práce se proto zabývá zejména moderními přístupy k problému paralelizace, tj. využití standardizovaných knihoven a dalších nástrojů, které programátorovi značně ulehčují práci (např. knihovna OpenMP či vizualizovaná paralelizace ve vývojovém prostředí LabView). Nejnovější přístup v oblasti paralelizace pak představuje využití výpočetních jednotek grafických karet (GPU), které od hlavního procesoru mohou přebírat úlohy všeobecného zaměření, tj. ne pouze úlohy týkající se grafiky, a tím urychlovat výsledné zpracování.

2 PARALELIZACE

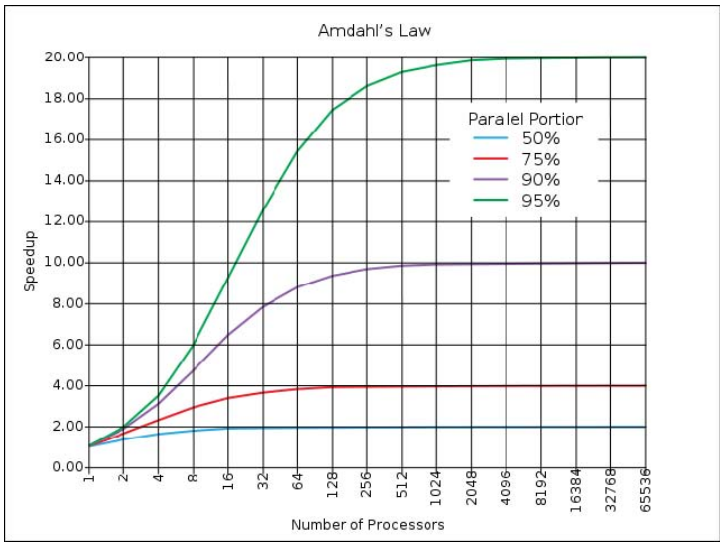
2.1 Kdy paralelizovat

V praxi se nejčastěji vyskytují úlohy, kde se střídají části zpracovávané sekvenčně s částmi zpracovávanými paralelně. Mezi části, které je třeba zpracovávat sekvenčně, patří zejména vstupní a výstupní operace. Veškeré ostatní výpočty lze obvykle paralelizovat.

Ne vždy však platí, že čím větší bude paralelizace, tím rychleji bude úloha zpracována, resp. neplatí přímá úměra mezi počtem výpočetních jednotek a rychlostí zpracování (obr. 1). K určení, zda se ještě další paralelizace vyplatí, slouží tzv. Amdahlovo pravidlo (pojmenované podle počítačového konstruktéra Gena Amdahla). [3]

$$Sz = \frac{1}{(1 - fp) + \frac{fp}{N}}$$

Sz - zrychlení
fp - podíl části kódu, kterou lze provádět paralelně
N - počet procesorů



Obr. 1: Závislost zrychlení na počtu procesorů.[38]

2.2 Proces, vlákno

Při spuštění programu vytvoří operační systém tzv. proces, kterému je přidělena paměť a jedno prováděcí vlákno, označované jako primární. Běh programu se realizuje prostřednictvím prováděcího vlákna, kterému je přidělován procesorový čas. Vlastní proces tedy nedostává přidělen procesorový čas. Ten dostávají přidělena pouze vlákna. V rámci procesu je ale možné vytvořit množství vláken, která mohou dostávat přidělen procesorový čas. Jednoduše lze říct, že když na čtyř jádrovém procesoru proces vytvoří čtyři vlákna, tak každé toto vlákno může běžet na "svém vlastním" jádru a maximálně využít možností procesoru.

Vlákna jednoho procesu sdílí všechny jeho prostředky, tzv. kontext procesu včetně adresového prostoru v paměti [3].

2.3 Přístupy k řešení paralelního zpracování

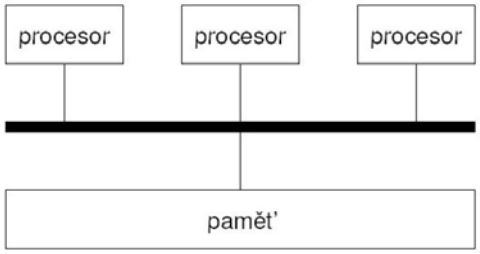
Paralelní zpracování můžeme primárně rozdělit do dvou skupin:

- úlohově založené (task based) – úloha se rozloží na více jednodušších. Každou z takto vzniklých dílčích úloh poté vykonává jedno vlákno. Tyto vlákna jsou následně přidělovány jednotlivým výpočetním jednotkám (procesorům). Tento druh paralelismu má smysl i na jednoprocesorových počítačích [2];
- datově založené (data based) – používá se v případě relativně jednoduché úlohy, která však zpracovává velké množství dat. Tyto data se tedy rozdělí na menší části. Každé vlákno poté provádí stejnou úlohu nad svojí částí dat. Tento druh paralelismu prakticky nemá smysl vykonávat na jednoprocesorovém stroji [2].

2.4 Používané architektury

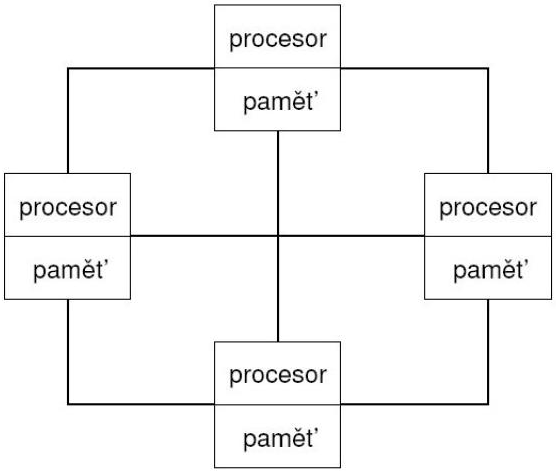
Hlavním kritériem pro posuzování architektur používaných pro paralelní zpracování úloh je typ používané paměti. Rozlišují se zejména následující druhy:

- sdílená paměť (obr. 2) – jednotlivé procesory či jádra sdílejí společnou paměť. Typické pro dnešní vícejádrové procesory [1];



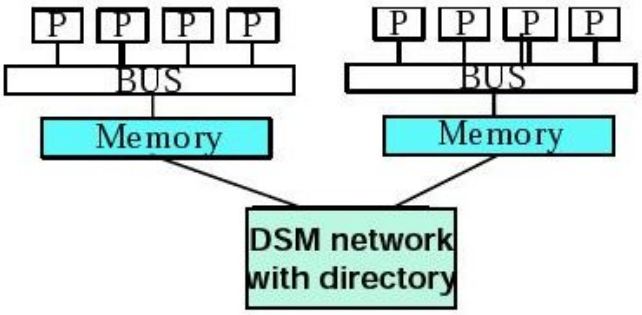
Obr. 2: Sdílená paměť.[2]

- distribuovaná paměť (obr. 3) – jedná se o více (většinou velké množství) strojů spojených dohromady pomocí komunikační sítě. Jednotlivé procesory spolu obvykle komunikují pomocí zasílání zpráv [1], [2];



Obr. 3: Distribuovaná paměť [2]

- architektura NUMA (obr. 4) – jedná se o speciální architekturu, kde jsou jednotlivé skupiny procesorů připojeny ke své lokální paměti přes svou paměťovou sběrnici. Tyto skupiny jsou pak vzájemně propojeny vysokorychlostním spojením. Následná rychlost přístupu do paměti závisí na tom, zda procesor přistupuje do své lokální paměti nebo do paměti jiné skupiny. [8]



Obr. 4: Architektura NUMA [8].

3 PROSTŘEDKY PRO TVORBU PARALELNÍCH APLIKACÍ

Paralelní aplikace je možné realizovat využitím následujících skupin prostředků:

- Programovací jazyky s přímou podporou paralelního zpracování – jedná se o jazyky, kde je podpora paralelismu přímo implementována (je přímou součástí jazyka) ve formě příkazů, procedur, funkcí apod. Typickými představiteli jsou jazyk ADA a High Performance Fortran.
- Překladače s podporou paralelismu – umožňují použití univerzálního jazyka (C, Fortran).
- Paralelizační knihovny – umožňují použití univerzálního jazyka a univerzálního překladače. Typicky používanými jazyky jsou C, C++, Fortran, Java. [2]

Dále následuje popis nejdůležitějších prostředků pro tvorbu paralelních aplikací.

3.1 OpenMP

Programovací rozhranní OpenMP je určeno pro programování na systémech se sdílenou pamětí v jazyce C/C++ a Fortran (samo o sobě není určeno pro použití na distribuovaných systémech). Lze jej využít jak k úlohovému, tak k datovému paralelismu. Aby bylo dosaženo paralelismu, je nutné pro překlad kódu použít překladač podporující OpenMP. [5]

3.1.1 Základní charakteristika

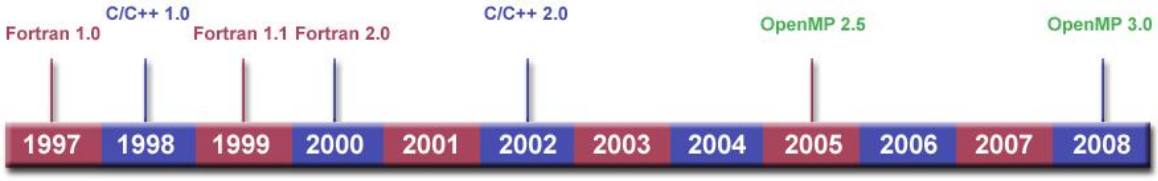
Cílem projektu OpenMP je zjednodušit vývoj paralelně zpracovávaných aplikací tak, aby se programátor nemusel zabývat vytvářením, spouštěním, synchronizací a ukončováním jednotlivých vláken, a aby se nemusel ani zabývat problémem kolik vláken a kdy vytvořit. Tvůrci OpenMP toho dosáhli tím, že vytvořili na platformě nezávislou skupinu direktiv, funkčních volání a proměnných, které přímo sdělují kompilátoru, jak a kde vložit do aplikace nové vlákno. [4]

OpenMP sestává ze tří hlavních komponent:

- OpenMP direktivy – jedná se o pokyny pro překladač, jak má určité části programu paralelizovat;
- knihovní funkce – nejčastěji se používají pro zjišťování informací o systému (např. kolik vláken právě běží, číslo vlákna atd.);
- systémové proměnné.

3.1.2 Standart OpenMP

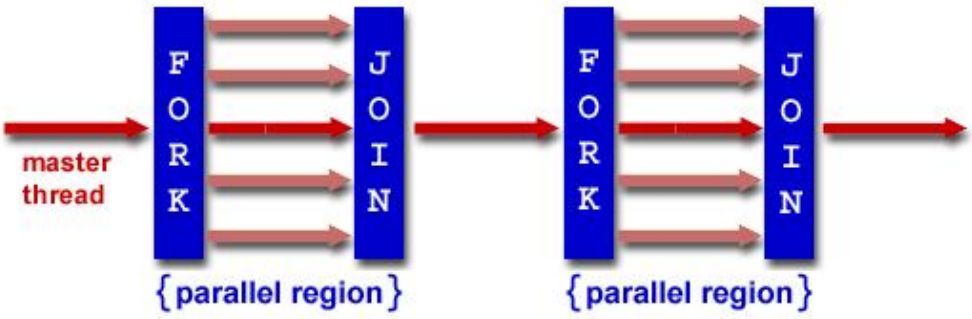
OpenMP bylo poprvé standardizováno v roce 1997, kdy byl přijat standard pro Fortran 1.0. Rok poté vychází standard pro jazyk C/C++ a OpenMP je postupně přijímáno výrobci překladačů jazyků Fortran a C/C++. Zatím poslední verze OpenMP 3.0 byla uvedena v roce 2008 (obr. 5).



Obr. 5: Historie standardu OpenMP. [5]

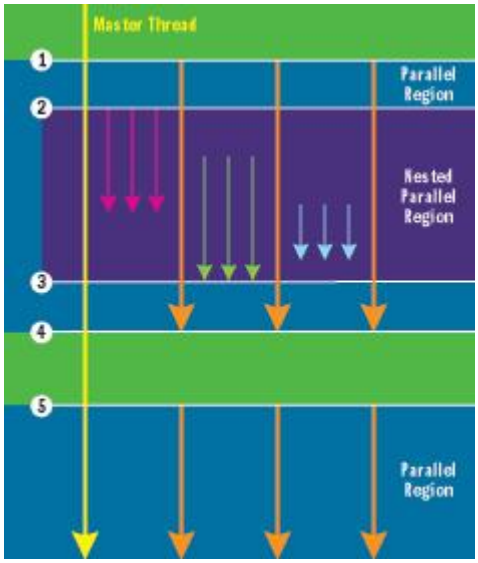
3.1.3 Paralelismus v OpenMP

Paralelismus v OpenMP je založen na tzv. FORK-JOIN modelu (obr. 6). OpenMP aplikace je spuštěna ve formě jednoho tzv. hlavního vlákna. Během svého vykonávání může aplikace narazit na oblast, která má být zpracována paralelně. V tomto případě hlavní vlákno vytvoří skupinu vláken (včetně sebe sama), které následně začnou danou oblast vykonávat (FORK). Jakmile je dosažen konec paralelní oblasti, dochází k synchronizaci a ukončení všech vláken kromě vlákna hlavního, které dále zpracovává aplikaci (JOIN). [5] [6]



Obr. 6: FORK-JOIN model.[5]

OpenMP též může podporovat vnořování paralelních oblastí, což znamená, že uvnitř paralelní oblasti, kterou zpracovává daná skupina vláken, může kterékoli z vláken vytvořit další tým vláken a pro tuto skupinu vláken se stát jejich hlavním (obr. 7). Tato možnost však nemusí být podporována ve všech implementacích OpenMP. Pokud podporována není, je vnořená paralelní oblast zpracována sériově aktuálním vláknem skupiny. [6]



Obr. 7: Vnořené paralelní oblasti. [6]

3.1.4 Konstrukce OpenMP

Zápis direktiv v jazyce Fortran i v jazyce C/C++ obecně dodržuje stejná pravidla. Liší se pouze syntaxí daného jazyka.

Direktivy v jazyce Fortran začínají indikátorem, následuje název direktivy a žádný, jeden nebo více příkazů.

Příklad zápisu direktivy v jazyce Fortran:

```
!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(BETA, PI)
```

Direktivy v jazyce C/C++ dodržují obecná pravidla psaní příkazu překladači tj. všechny direktivy začínají #pragma omp, což zajistí, že překladač, který nepodporuje OpenMP bude tyto direktivy ignorovat. Následuje opět název direktivy a případně příkazy. Nakonec musí povinně následovat nový řádek. [5]

Příklad zápisu direktivy v jazyce C++:

```
#pragma omp parallel default(shared) private(beta, pi)
```

3.1.5 Direktiva parallel

Jedná se o základní direktivu OpenMP. Označuje blok kódu, který bude zpracován paralelně více vlákny. Počet vytvořených vláken se řídí následujícími pravidly seřazenými podle priority:

1. vyhodnocení příkazu IF
   Po direktivě parallel může následovat příkaz if podmiňující paralelizaci dané sekce kódu;
2. nastavení klauzule NUM_THREADS;
3. použití knihovní funkce omp_set_num_threads;
4. nastavení členské proměnné OMP_NUM_THREADS;
5. základní nastavení (obvykle počet CPU).

Počet použitých vláken také závisí na tom, zda je povolena dynamická úprava počtu vláken. Pokud je povolena, pak požadovaný počet představuje maximální možný počet vláken, která budou danou oblast zpracovávat, což je vhodné zejména pro případ, kdy žádaný počet vláken překračuje možnosti systému. K zapnutí nebo vypnutí dynamické úpravy počtu vláken slouží knihovní funkce omp_set_dynamic nebo členská proměnná OMP_DYNAMIC. [5], [9]

3.1.6 Direktivy pro rozdělení práce mezi vlákna (konstrukce Work-Share)

Konstrukce Work-Share rozděluje zpracování dané oblasti mezi vlákna vytvořené direktivou parallel (tj. nevytváří se žádná nová vlákna). [5],[10]

- Direktiva for – direktiva pro paralelizaci cyklu. Jednotlivé iterace bezprostředně následujícího for cyklu jsou rozděleny mezi vlákna daného týmu;

```
#pragma omp for [seznam příkazů...]
for(;;)
```

- direktiva sections – neiterační paralelizace. Práce, kterou vykonává takto označená část kódu je rozdělena mezi jednotlivá vlákna pomocí vložených direktiv section;

```
#pragma omp sections [seznam příkazů...] {
  [#pragma omp section]
        blok_kódu
  [#pragma omp section]
        blok_kódu
...
}
```

- direktiva single – označuje část kódu, kterou má vykonat pouze jedno vlákno z týmu, což je vhodné zejména pro I/O operace.

```
#pragma omp single [seznam příkazů...]
  blok_kódu
```

3.1.7 Synchronizační direktivy

- Direktiva critical – specifikuje oblast kódu, kterou v dané chvíli může vykonávat pouze jedno vlákno;
- direktiva barrier – synchronizuje vlákna dané skupiny. Vlákno v tomto místě čeká tak dlouho, dokud všechny ostatní vlákna ze skupiny tohoto místa také nedosáhnou. Na konci každé Work-share konstrukce se synchronizace provádí automaticky;
- direktiva atomic – zjednodušená direktiva critical. Vztahuje se pouze na bezprostředně následující výraz (používá se zejména pro čítače);
- direktiva flush – každé vlákno si udržuje své lokální kopie proměnných a jejich hodnoty. Direktiva flush zajišťuje konzistenci dat (tj. zapsání lokálních hodnot do sdílené části paměti a naopak). Pokud se za direktivou neuvede seznam proměnných, na které má být aplikována, provádí se na všech sdílených datech;

- direktiva ordered – používá se v kombinaci s paralelizovaným cyklem for. Specifikuje část kódu uvnitř cyklu, kterou je třeba zpracovávat postupně podle jednotlivých iterací cyklu, tak jak jdou po sobě. V definici paralelního for cyklu musí být uveden příkaz ordered. [5] [11]

3.1.8 OpenMP příkazy

Zápis direktivy mohou následovat příkazy, které specifikují jak přistupovat k daným datům a jak s nimi pracovat. Mohou též upřesňovat způsob paralelizace nebo synchronizace.

- Private – každé vlákno si vytvoří vlastní instanci proměnných uvedených v seznamu a s těmi poté pracuje;
- shared – dané proměnné jsou sdíleny mezi všemi vlákny;
- default – určuje, jak se bude přistupovat k proměnným, které nebyli nijak specifikovány;
- firstprivate – stejné chování jako private doplněné o nastavení počátečních hodnot uvedených proměnných. Jako inicializační hodnota se použije buď hodnota přímo předcházející vstupu do paralelní oblasti nebo hodnota předcházející vstupu vlákna do konstrukce Work-Share;
- lastprivate – stejné chování jako private doplněné o nastavení koncových hodnot proměnných uvedených v seznamu. Jako koncová hodnota proměnné se použije hodnota z vlákna, které provedlo poslední iteraci cyklu (direktiva for) nebo provedlo poslední sekci (direktiva section);
- copyin – umožňuje vláknům přistupovat k hodnotě proměnné v hlavním vláknu skupiny;
- copyprivate – poskytuje mechanismus k přenášení hodnoty proměnných z jednoho vlákna do všech kopií daných proměnných v ostatních vláknech. Použitelné pouze v kombinaci s direktivou single;
- reduction – redukce proměnných uvedených v seznamu za použití specifikovaného operátoru. Na lokální hodnoty uvedených proměnných pro dané vlákno se aplikuje zvolený operátor a výsledek se zapíše do sdílené paměti;
- ordered – příkaz vyžadovaný v definici paralelního cyklu for, pokud je použita direktiva ordered;
- schedule – příkaz používaný při paralelizaci cyklu for. Určuje, jak budou jednotlivé iterace rozděleny mezi vlákna;
- nowait – způsobí ignorování implicitních synchronizačních bariér u daných direktiv. [5]

Clause	Directive					
	PARALLEL	DO/for	SECTIONS	SINGLE	PARALLEL DO/for	PARALLEL SECTIONS
IF	●				●	●
PRIVATE	●	●	●	●	●	●
SHARED	●	●			●	●
DEFAULT	●				●	●
FIRSTPRIVATE	●	●	●	●	●	●
LASTPRIVATE		●	●		●	●
REDUCTION	●	●	●		●	●
COPYIN	●				●	●
COPYPRIVATE				●		
SCHEDULE		●			●	
ORDERED		●			●	
NOWAIT		●	●	●		

Obr. 8: Použitelnost jednotlivých příkazů pro jednotlivé direktivy. [5]

3.2 MPI (Message Passing Interface)

Jedná se o programovací rozhranní založené na zasílání zpráv mezi jednotlivými uzly, které se primárně používá na systémech s distribuovanou pamětí. Nejčastěji je implementováno v jazycích C/C++, Fortran, Java a Python.

3.2.1 Historie

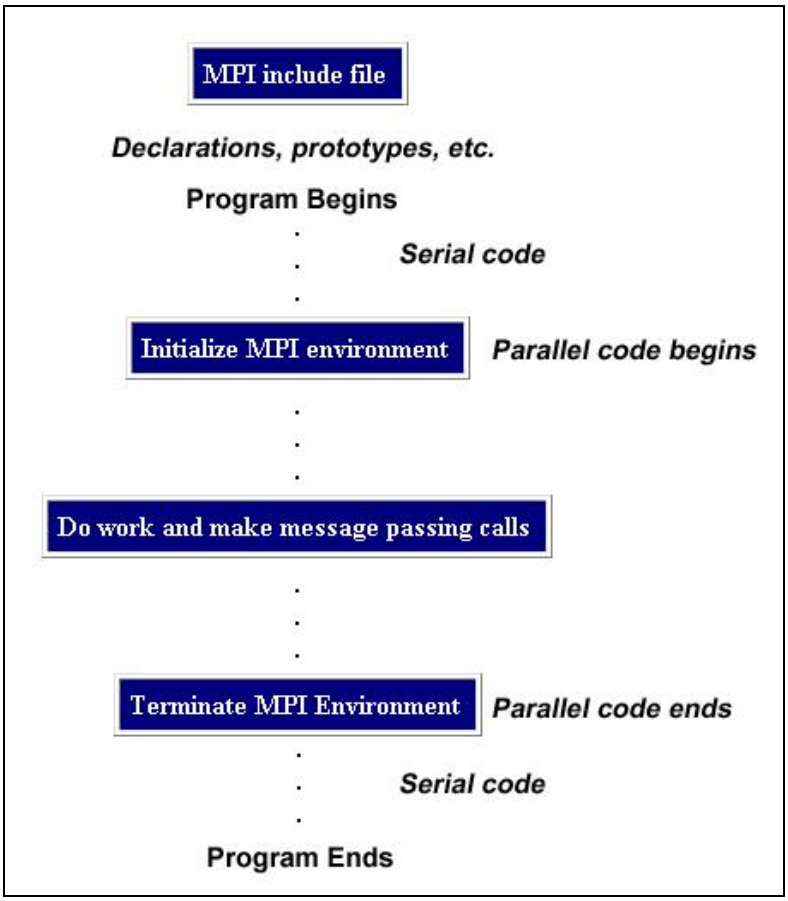
Systémy předávání zpráv byli vždy široce využívány, avšak až do počátku devadesátých let minulého století existovaly jejich realizace pouze jako produkty jednotlivých výrobců či vývojářských skupin. Mezi nejpopulárnější patřil například PVM (Parallel Vitual Machine). Nevýhodou těchto systémů však byla jejich vzájemná nekompatibilnost.

Proto byl iniciován vznik nového standardu a v roce 1994 byl uvolněn standard MPI 1.1. V současné době se nejvíce používá verze 1.2 (označovaná jako MPI 1) a verze 2.1 vydaná v roce 2008 (označovaná jako MPI 2). [7]

3.2.2 Základní funkce MPI

- MPI_Init – inicializace MPI výpočtu. Tato funkce musí být obsažena v každém MPI programu, musí být volána vždy jako první MPI funkce a volána může být vždy jen jednou;
- MPI_Comm_size – zjišťuje počet procesů v dané skupině spojené s daným komunikátorem. Obecně je používána s komunikátorem MPI_COMM_WORLD ke zjištění počtu procesů používaných v celé aplikaci;

- MPI_Comm_rank – zjištění čísla procesu v rámci skupiny určené daným komunikátorem. Každý proces obdrží při svém vytvoření unikátní číslo, podle kterého ho lze následně identifikovat, což je důležité zejména pro vzájemnou identifikaci při komunikaci mezi dvěma procesy.
- MPI_Finalize – ukončení MPI výpočtu. Tato funkce je volána jako poslední tzn. žádná další MPI funkce již nemůže následovat. [12], [13]



Obr. 9: Typická struktura programu MPI.[13]

3.2.3 Skupiny a komunikátory

Skupina je uspořádaná množina procesů, kde je každý proces identifikovatelný podle svého unikátního čísla. Tato čísla začínají na nule a končí na N-1, kde N představuje počet procesů ve skupině. Skupina je vždy spojena se svým komunikátorem.

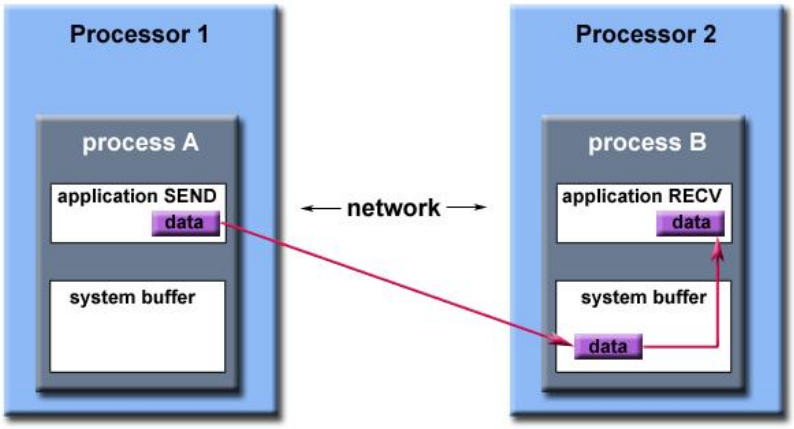
Komunikátor zajišťuje vzájemnou komunikaci mezi procesy skupiny, kterou daný komunikátor reprezentuje. Všechny MPI zprávy musí specifikovat, jaký komunikátor budou používat. Komunikátor MPI_COMM_WORLD reprezentuje všechny procesy dané aplikace.

Komunikátory/skupiny mohou být dynamicky vytvářeny a rušeny za běhu programu. Jednotlivé procesy mohou být členy více skupin, přičemž v rámci každé skupiny mají svá unikátní identifikační čísla. [12], [13]

3.2.4 Komunikace mezi uzly

Jedná se o komunikaci mezi dvěma procesy, kde jeden proces vykoná operaci odeslání a druhý proces k ní sdruženou operaci příjmu.

- Použití bufferu – v ideálním případě po každém odeslání následuje na přijímací straně okamžitě příjem zprávy. Jednotlivé implementace MPI však musí být schopny obsloužit komunikaci i v případě, kdy je z nějakého důvodu opožděn příjem zprávy. Nejčastěji se to řeší systémovým bufferem, kde jsou data na straně příjemce dočasně uložena (obr. 10);



Obr. 10: Použití bufferu na straně příjemce.[13]

- blokování vs. neblokování – většina komunikačních funkcí MPI může být provozována v blokovacím nebo neblokovacím režimu. Blokovací režim obecně znamená kontrolovaný přenos/příjem (tj. dokud přenos nebo příjem neskončí, nebude se provádět nic jiného);
- pořadí – standart MPI garantuje, že se jednotlivé zprávy od jednoho odesílatele nebudou předbíhat (tj. budou přijaty v pořadí v jakém byly odeslány).

Základní komunikaci tedy obsluhují dvě MPI funkce – MPI_Send a MPI_Recv. V parametrech funkce MPI_Send se nastavuje co se bude posílat, jaké množství se bude posílat, jakého jsou data typu, kam se budou posílat, typ zprávy a použitý komunikátor. Obdobně tomu je i v případě MPI_Recv. [13]

Prototypy zmíněných funkcí:

```
MPI_Send (&buf,count,datatype,dest,tag,comm);
MPI_Recv (&buf,count,datatype,source,tag,comm,&status)
```

3.3 Paralelismus na platformě Microsoft .NET Framework

Platformu Microsoft .NET Framework tvoří tyto části:

- základní běhové rozhranní CLR (Common Language Runtime) – obsluhuje vykonávání kódu;
- knihovna základních tříd BCL (Base Class Library) – představuje komplexní, objektově orientovanou sbírku funkcí, potřebných k vývoji aplikací různého zaměření (od tradičních až po webové aplikace). [14]

3.3.1 Paralelismus v .NET verze 3.5 a nižších

Knihovna základních tříd (Base Class Library) platformy .NET obsahuje řadu tříd a metod pro obsluhu paralelizace (tj. nástroje pro synchronizaci, nástroje pro vzájemnou komunikaci, vytváření nových vláken apod.) známých z moderních operačních systémů. Všechny důležité třídy a metody pro obsluhu paralelizace jsou obsaženy ve jmenném prostoru Systém.Threading.

Je možné nové vlákna vytvářet a rušit (třída Thread). Každý proces v .NET má též k dispozici svůj fond vláken tzv. thread pool (třída ThreadPool) Jedná se o zásobník vláken čekajících na přidělení práce. Jeho použití je jednoduší než v případě třídy Thread, neboť odpadá složité vytváření a rušení vlákna, což obsluhuje systém. Dále jsou obsaženy třídy pro synchronizaci (knihovna Mutex, Monitor, Interlocked atd.).

Tento přístup však klade značné nároky na programátora (lze takto nadělat mnoho programátorských chyb) a neliší se od přístupu z jiných prostředí. [15], [16], [17]

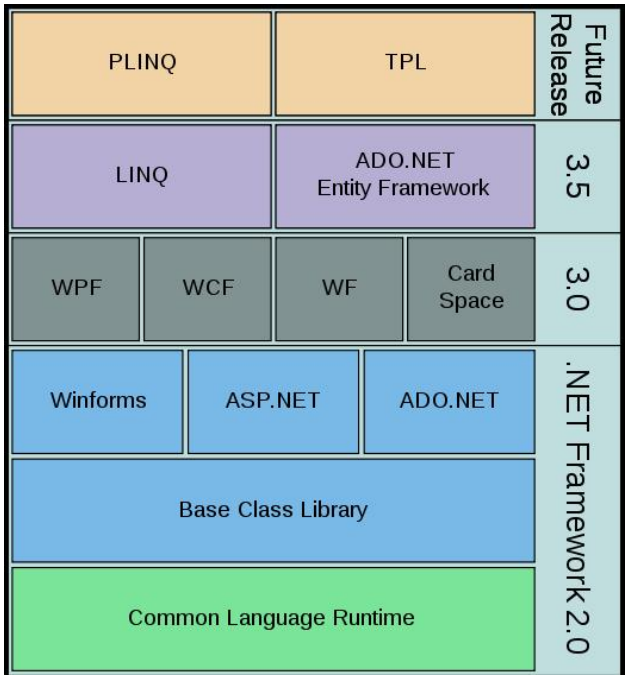
To, co je na .NET Frameworku nové, je mezi-procesová komunikace, označovaná jako Remoting, která umožňuje spouštět a připojovat se na procesy nejen na lokálním počítači, ale kdekoliv v síti.

Dále od verze 3.0 přibila podpora pro "Windows Workflow Foundation", který umožňuje jednodušší řešení běhu aplikace a další záležitosti potřebné pro paralelní běh, více v kap. 4.3.

3.3.2 Paralelismus v .NET verzi 4.0

Nejnovější verze platformy .NET 4.0 nabízí podporu jak datového, tak úlohového paralelismu. Paralelní nástavba pro .NET 4.0 se souhrnně označuje jako ParallelFX. Zahrnuje v sobě třídy pro podporu datového paralelismu (označované jako PLINQ) a úlohový paralelismus obsluhující knihovnu TPL (Task Parallel Library).

Použití ParallelFX umožňuje programování na vyšší úrovni abstrakce, čímž odpadá například práce spojená s vytvářením a rušením vláken. ParallelFX používá vlastní řízení běhu programu (tj. rozhoduje o vytváření nových vláken). [16]



Obr. 11: Struktura platformy .NET [39]

3.3.3 PLINQ

PLINQ je nástroj pro deklarativní paralelizaci dat (tj. specifikuje se pouze co se má paralelizovat a ne jak se to má paralelizovat). Jedná se o paralelní nástavbu systému LINQ, základní použití PLINQ je tedy totožné s používáním LINQ. LINQ (Language Integrated Query) je implementován v .NET 3.5 a představuje zcela nový způsob práce s daty. LINQ rozšiřuje platformu .NET tím, že v sobě integruje dotazování na objekty, databáze a XML data. Použití LINQ umožňuje psát dotazy přímo v programovacím jazyce podporujícím .NET bez nutnosti použití jiných jazyků (např. SQL). Hlavní výhodou tohoto přístupu je zejména odhalení chyb již při kompilaci kódu a ne až při běhu programu. [16]

Podmínkou pro použití LINQ resp. PLINQ je, že dotazovaná data musí být zapouzdřena v objektu. Pokud se dotazovaná data primárně neukládají jako objekt, musí se na nějaký objekt namapovat. [18]

Příklad dotazu PLINQ:

```
IEnumerable<T> data = ...;
var q = from x in data.AsParallel() where ( … ) orderby ( …
) select ( … );
```

Rozšiřující metoda IEnumerable.AsParallel představuje základní rozdíl mezi LINQ a PLINQ. Metoda převádí kolekci dat z IEnumerable na IparallelEnumerable. [19]

3.3.4 TPL (Task Parallel Library)

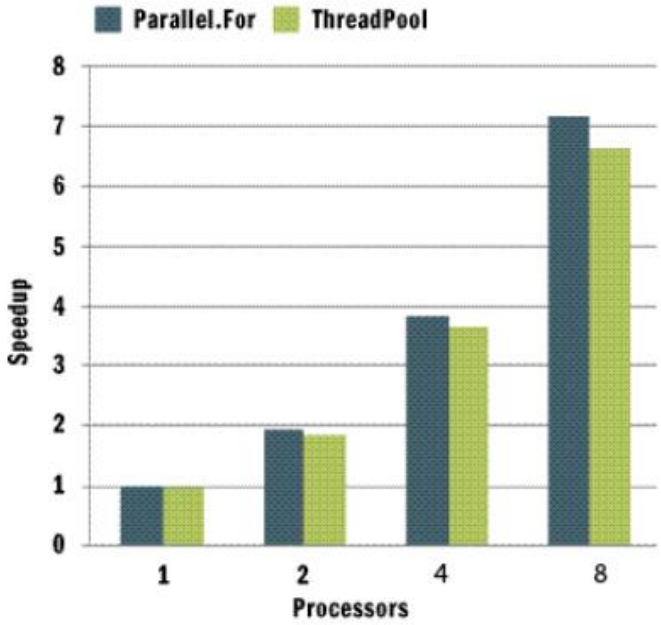
Knihovna TPL poskytuje abstrakci pro paralelní zpracování ve formě třídy Parallel, třídy Task a třídy Future. Knihovna TPL striktně paralelismus negarantuje, pouze

ho umožňuje (např. na jednoprocesorovém systému je paralelní for cyklus zpracován klasicky jedním vláknem).

Knihovna obsahuje pokročilé algoritmy pro dynamické rozdělování práce a automaticky rozděluje pracovní zátěž v závislosti na systému. V knihovně je implementován správce úloh, který implicitně přiděluje jedno pracovní vlákno jedné výpočetní jednotce. Každé pracovní vlákno má svou frontu úloh, kam je ukládána práce, kterou má vykonat. Pokud je fronta prázdná, začne dané pracovní vlákno prohledávat fronty ostatních pracovních vláken a případně převádět jejich práci na sebe. [16], [20]

- Třída Parallel poskytuje paralelizaci cyklů for a foreach. Příklad zápisu paralelizovaného cyklu for, který sečte všechna prvočísla menší než 100:

```
int sum = 0;
Parallel.For(0, 100, delegate(int i) {
if (isPrime(i)) {
  lock (this) { sum += i; }
}
```



Obr. 12: Srovnání výkonu klasického přístupu a použití knihovny TPL. [20]

- Třída Task vytváří nové úlohy, které mohou být zpracovány paralelně. Zavoláním funkce Task.Create() však primárně nevzniká nové vlákno, ale pouze úloha.
- Třída Future vytváří nové úlohy s návratovou hodnotou. Představuje specializaci třídy Task. Návratovou hodnotu lze získat pomocí vlastnosti Value. Value je blokovací vlastnost tj. pokud v době volání ještě není hodnota k dispozici, čeká se na dokončení úlohy (interně je volána metoda Wait()). Příklad použití třídy Future pro určení počtu uzlů v binárním stromu:

```
int CountNodes(Tree<T> node) {
  if(node == null) return 0;
  var left= Future.Create(() => CountNodes(node.Left));
```

```
    int right= CountNodes(node.Right);
    return 1 + left.Value+ right;
}
```

4 PARALELISMUS V GRAFICKÝCH VÝVOJOVÝCH PROSTŘEDÍCH

V tradičním textovém programování musí programátor jasně definovat části kódu, které se mají zpracovávat paralelně. Jelikož textové programování je z vlastní podstaty sériové, je poměrně obtížné si samotný paralelismus vizuálně představit. K usnadnění vlastní představy lze proto použít některé z dostupných grafických vývojových prostředí.

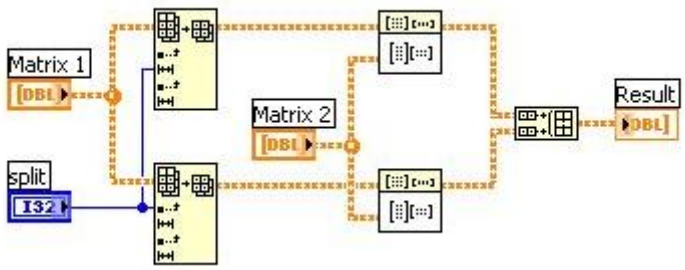
4.1 NI LabVIEW

Grafické vývojové prostředí NI LabVIEW je vyvíjeno firmou National Instruments od roku 1986. Od roku 2007 je ve verzi 8.5 implementována podpora paralelního zpracování.

LabVIEW svou grafickou podstatou vkládání a spojování jednotlivých bloků umožňuje programátorům jednoduše vizualizovat a programovat paralelní aplikace. Navíc LabVIEW automaticky generuje vlákna pro paralelní sekce programu, tudíž je použitelné i pro vývojáře, kteří nejsou zběhlí v klasickém programování. [21]

4.1.1 Datový paralelismus

Datový paralelismus je technika rozdělení velkého objemu dat na menší celky, jejich následné paralelní zpracování a konečné složení výsledku. V grafickém prostředí LabView lze názorně zobrazit jednotlivé kroky tak, jak po sobě následují (obr. 13).

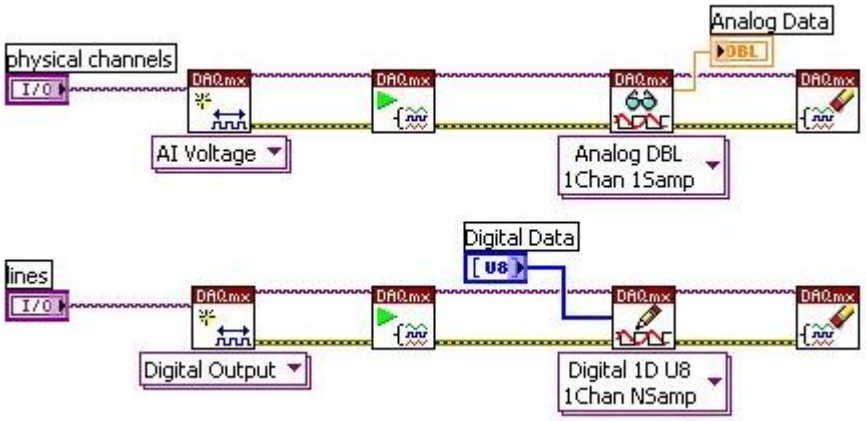


Obr. 13: Násobení dvou matic s použitím datového paralelismu v LabView [22].

Lze použít i klasické násobení matic, ovšem pouze za předpokladu, že samotná funkce násobení matic již je paralelizována a optimalizována pro více výpočetních jednotek. [22]

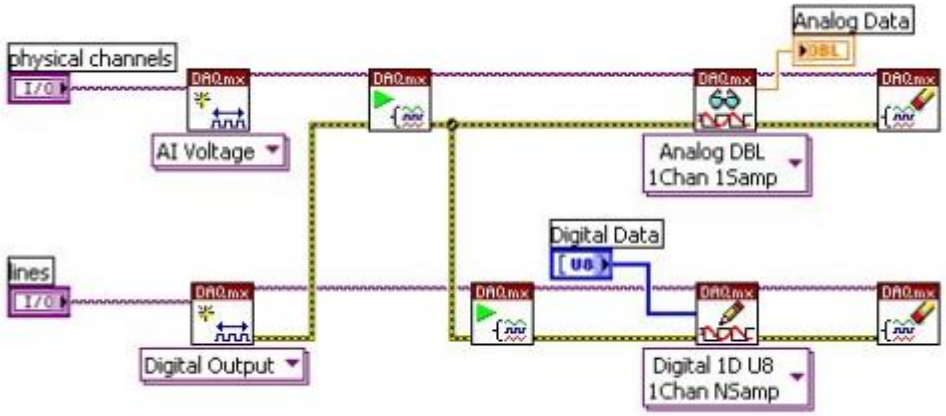
4.1.2 Úlohový paralelismus

Jedná se o současné zpracovávání více nezávislých úloh (viz. 1.2). Dvě vzájemně datově nezávislé úlohy jsou v LabVIEW automaticky zpracovávány paralelně bez nutnosti jakéhokoliv dalšího programování (obr. 14). V systému s více výpočetními jednotkami může každá úloha běžet na vlastní jednotce, čímž se výrazně zrychlí doba zpracování.



Obr. 14: Nezávislé sekce kódu budou automaticky zpracovány paralelně. [23]

Při vývoji paralelních aplikací s použitím úlohového paralelismu je nutné mít na paměti, aby dvě úlohy byly skutečně vzájemně nezávislé (tj. nesdíleli spolu zdroje apod.). [23]



Obr. 15: Pokud jedna úloha závisí na druhé, nemohou být zpracovány paralelně. [23]

4.2 MATLAB/Simulink

MATLAB je vývojové prostředí vyvíjené firmou Mathworks od roku 1984. Jedná se o prostředí určené zejména k vývoji výpočetně náročných úloh.

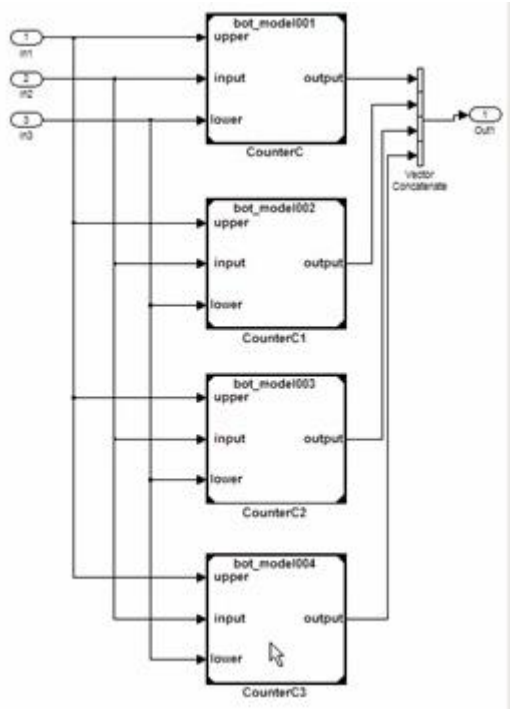
Simulink je grafickou nástavbou MATLABu určenou k simulaci a testování chování nejrůznějších systémů.

4.2.1 Implementace paralelismu

V MATLABu/Simulinku lze programovat paralelní aplikace či spustit simulace s podporou paralelního zpracování za pomoci modulu Parallel Computing Toolbox. Pomocí tohoto modulu lze s použitím příkazu matlabpool spustit daný počet pracovních vláken, mezi která bude následně rozdělena požadovaná práce. Modul Parallel Computing

Toolbox obsahuje příkazy jak pro úlohový paralelismus (např. parfor), tak pro datový paralelismus (např. spmd). Takto lze jednoduše přepsat již existující aplikace tak, aby podporovali paralelismus. [24]

V případě simulací v grafickém prostředí Simulink budou paralelně automaticky zpracovávány vzájemně nezávislé bloky (obr. 16). [25]



Obr. 16: Čtyři bloky, které jsou v Simulinku zpracovány čtyřmi pracovními vlákny. [25]

4.3 Windows Workflow Foundation

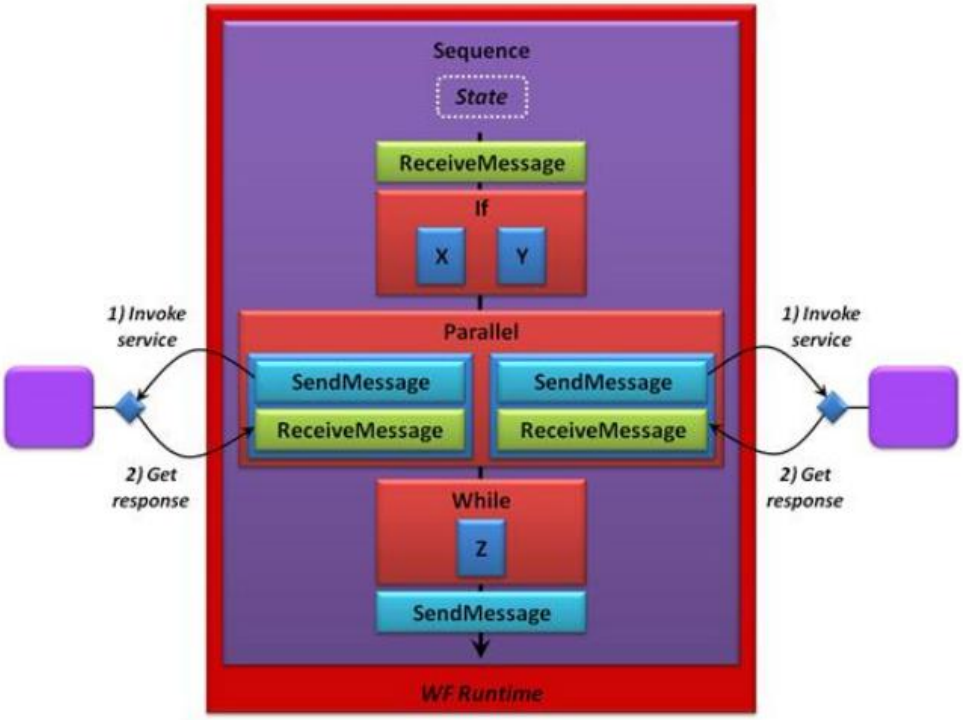
Windows Workflow Foundation je součástí platformy .NET a slouží především k vizualizaci vlastní funkce aplikace. Při pohledu na klasický kód je totiž někdy velice obtížné představit si vazby mezi jednotlivými částmi kódu a vlastní funkce těchto částí v konečném celku.

Windows Workflow Foundation umožňuje řešený problém vizuálně rozkreslit do bloků (jednotlivé činnosti) a ty pospojovat vazbami. Práce prováděná uvnitř daných činností je následně psána v klasickém kódu. Zjednodušeně lze říci, že workflow schéma označuje co se má dělat a klasický kód určuje jak se to má dělat. [26]

4.3.1 Koordinace paralelní práce

Základní knihovna činností Windows Workflow Foundation obsahuje činnost parallel. Tato činnost způsobí rozvětvení do dvou nebo více souběžných činností. Ačkoliv ve schématu startují a běží rozvětvené činnosti současně, ve skutečnosti tomu tak není. Runtime rozhranní Windows Workflow Foundation zpracovává každé workflow v jednom vlákně. Tudíž i všechny rozvětvené činnosti sdílejí toto jedno pracovní vlákno. Jednotlivé činnosti jsou tedy zpracovány postupně (tj. jakmile jedna skončí, začne se zpracovávat další), přičemž pořadí zpracování není dáno.

Výhoda činnosti parallel se však ukáže v momentě, kdy jedna nebo více rozvětvených činností je blokovací (čeká na vstup od uživatele, na I/O operaci apod.). V tomto případě se nečeká až dojde k odblokování dané činnosti, ale okamžitě se začne zpracovávat činnost další. K ukončení činnosti parallel dochází v okamžiku, kdy je dokončeno zpracování všech jejich větví. [27], [28]



Obr. 17: Příklad použití činnosti parallel při volání dvou různých služeb v daném workflow. [27]

5 PARALELIZACE S POMOCÍ GPU (GPGPU)

GPGPU (General-Purpose computation on Graphics Processing Units) je technika využívající vysokovýkonné mnohojádrové procesory grafických karet ke zrychlení různých aplikací (nejen grafických).

Již před mnoha lety napadlo některé programátory, že lze procesory grafických karet využít nejen ke grafickým výpočtům. Avšak v té době neexistoval žádný přijatelný programovací model, díky kterému by bylo možné toho dosáhnout a také jednotky grafických karet nebyly těmto požadavkům uzpůsobeny.

Všeobecného rozšíření se tato technika proto dočkala až koncem roku 2006, kdy přední světový výrobci představili svá řešení. [29]

5.1 nVidia CUDA

Jedná se o všestranně použitelnou architekturu pro paralelní zpracování, která do výpočetní jednotky procesoru grafických karet nVidia vkládá podporu paralelních výpočtů. Díky tomu je možné vyřešit mnoho komplexních výpočetních problémů ve zlomku času, který by byl potřeba na CPU. Architektura zahrnuje instrukční sadu CUDA (ISA) a paralelní výpočetní engine GPU. [30]

nVidia se v začátcích snažila svůj standard CUDA nabídnout i ostatním výrobcům grafických procesorů, avšak nikdo se ke standardu nepřipojil.

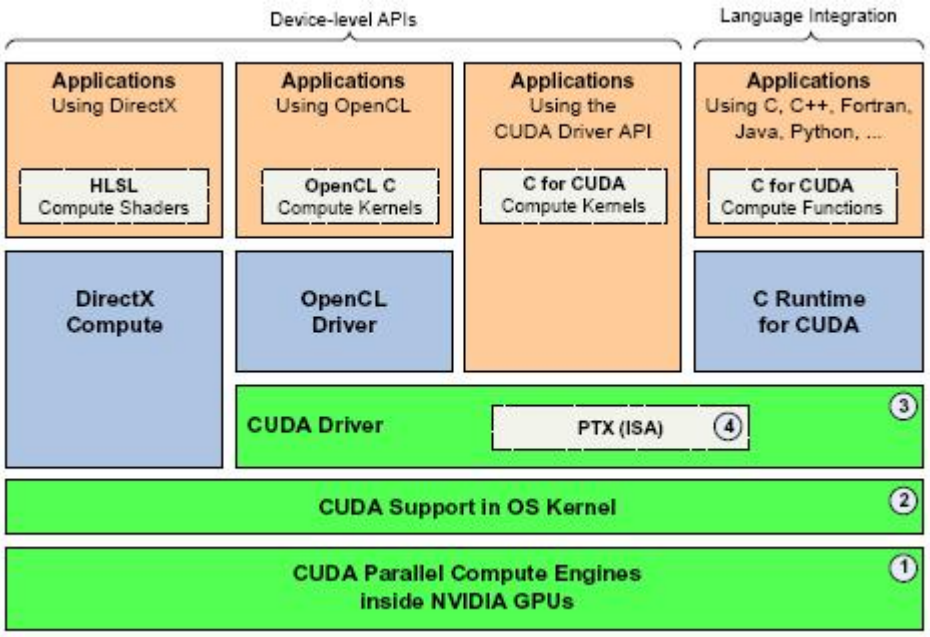
5.1.1 Programovací prostředky

Vývojové prostředí CUDA poskytuje následující nástroje [31]:

- knihovny – knihovny obsahující standardní matematické operace (lineární algebra, Fourierovy transformace atd.) optimalizované pro architekturu CUDA;
- C rozhranní – C rozhranní pro architekturu CUDA poskytuje podporu zpracování standardních funkcí jazyka C na GPU a umožňuje též provázání s dalšími programovacími jazyky vyšší úrovně jako Fortran, Java a Python;
- ostatní vývojové nástroje – nVidia C kompilátor, CUDA debugger, dokumentace atd.

Programování v architektuře CUDA umožňuje dva přístupy:

- programování na úrovni zařízení (device-level programming interface) – vývojář píše zpracovávané úlohy v programovacím jazyce podle zvoleného API (DirectX, OpenCL, CUDA). API následně přímo nastaví GPU a zpracuje dané úlohy;
- integrované rozhranní (language integration programming interface) – zpracovávané funkce se píší v jazyce C (či v jiném jazyce vyšší úrovně) a C rozhranní automaticky nastaví GPU pro zpracování těchto funkcí.



Obr. 18: Architektura CUDA. [31]

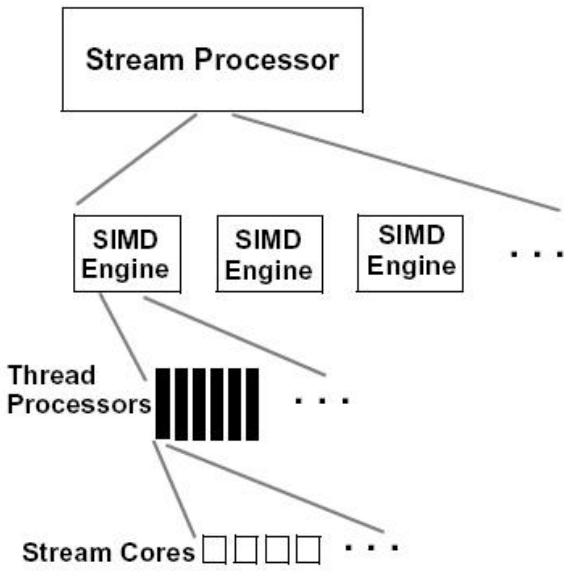
5.2 ATI Stream

Jedná se o skupinu pokročilých hardwarových a softwarových technologií, které umožňují grafickým procesorům AMD (GPU) pracovat ve spolupráci s centrálním procesorem (CPU), čímž je dosaženo zrychlení mnoha aplikací, nejen grafiky. [32]

5.2.1 Proudový procesor ATI Stream

Základní jednotkou zodpovědnou za jednotlivé výpočty jsou jádra vláknového procesoru. Jednotlivá jádra jsou programovatelná a mohou provádět celočíselné operace, operace v pohyblivé řádové čárce a transcendentní operace. Vláknové procesory jsou následně sdruženy do skupin – SIMD zařízení (engine). Proudový procesor pak tvoří daný počet těchto skupin (obr. 19).

Například proudový procesor ATI Radeon 3870 obsahuje 4 SIMD zařízení, každé zařízení obsahuje 16 vláknových procesorů a každý vláknový procesor tvoří 5 jader. [33]



Obr. 19: Základní struktura proudového procesoru. [33]

5.2.2 Programovací prostředky

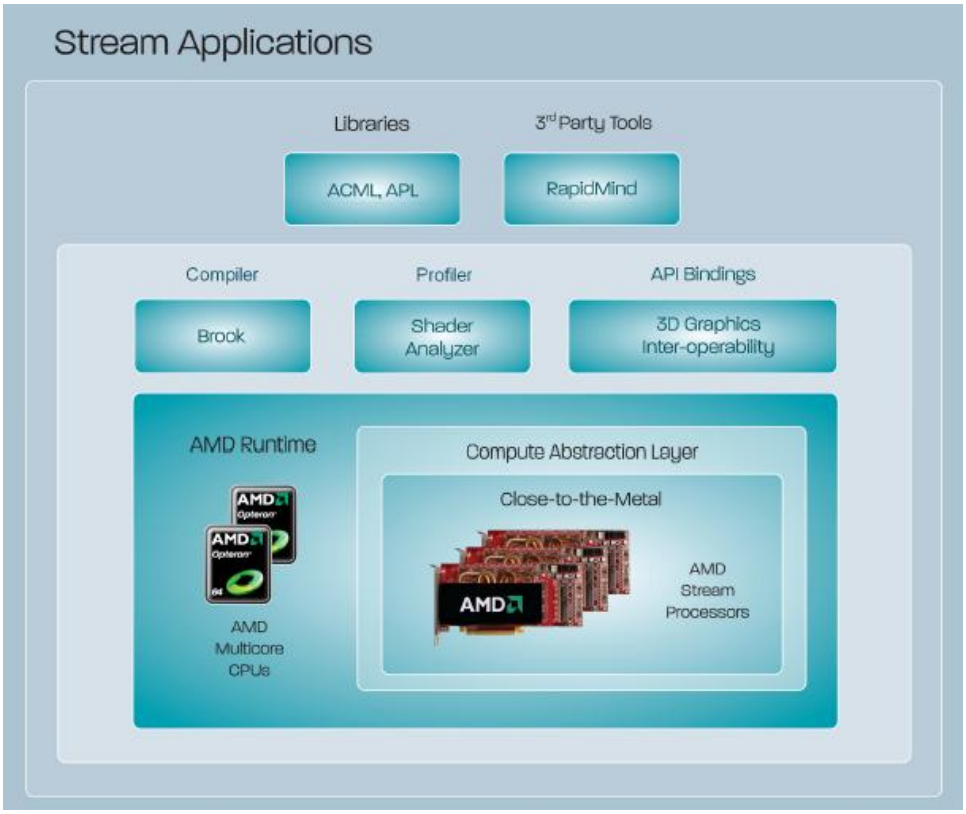
ATI poskytuje pro vývojáře následující prostředky [33], [34]:

- knihovna ACML (AMD Core Math Library) – soubor běžně používaných matematických funkcí optimalizovaných pro použití na platformě ATI (funkce lineární algebry, Fourierovy transformace, generátor náhodných čísel apod.);
- překladač Brook+ a RapidMind – Brook+ je variací volně dostupného překladače Brook C/C++, který byl upraven tak, aby podporoval programování pro GPU. Tato úprava umožňuje psát software zpracovávaný na grafických jednotkách všem programátorům. Dva klíčové prvky jazyka Brook jsou:
  - proud – kolekce dat stejného typu jež budou zpracována paralelně;
  - kernel – paralelní funkce, jež se bude vykonávat nad každým prvkem datového proudu.

  Příklad zápisu funkce, která sčítá dva vstupní proudy dat a výsledek odesílá do výstupního proudu:

```
kernel void sum( float a<>, float b<>, out float c<> )
{
      c = a + b;
}
```

  RapidMind představuje kompletní integrované vývojové rozhranní – editor kódu, kompilátor, debugger atd.;
- ovladače CAL (Compute Abstraction Layer) – CAL umožňuje vývojářům přistupovat k jednotlivým částem GPU (tj. psát kód přímo pro GPU bez nutnosti znalosti specifického programovacího jazyka);
- nástroje pro analýzu výkonu – GPU ShaderAnalyzer a AMD CodeAnalyst.



Obr. 20: Struktura programovacího rozhranní ATI Stream. [34]

5.3 OpenCL

Jedná se o první standard všeobecného paralelního programování heterogenních systémů spojujících v sobě využití počítačových procesorů (CPU), procesorů grafických karet (GPU) či dalších druhů procesorů (např. DSP – digitální signálový procesor apod.). OpenCL poskytuje vývojářům jednotné programovací rozhranní pro psaní efektivního a přenositelného kódu pro vysoce výkonné výpočetní servery i stolní počítače a různá další zařízení (mobilní zařízení atd.). [35]

Anatomie OpenCL:

- jazyková specifikace – programovací rozhranní založené na jazyce C, které obsahuje rozsáhlou sadu vestavěných výpočetních funkcí;
- vrstva zařízení (Platform layer API) – abstraktní vrstva zahrnující pod sebou všechny výpočetní jednotky systému. Vybírá a inicializuje jednotlivé výpočetní jednotky. Obsluhuje výpočetní kontext a frontu výpočtů;
- běhové rozhranní (Runtime API) – provádí jednotlivé výpočty. Obsluhuje plánování a práci s pamětí.

5.4 DirectX 11

DirectX je standardizované programovací rozhranní vyvíjené firmou Microsoft od roku 1995 pro obsluhu multimediálních úloh a grafiky, zejména se využívá pro programování her.

Rozhranní DirectX 11 je prakticky totožné s verzí DirectX 10. Největším rozdílem a hlavní výhodou je podpora multi-threadingu. Všechny kroky ve vykreslování, rasterizaci a zobrazování jsou prováděny postupně, ale DirectX 11 umožňuje aplikaci vytvářet zdroje, měnit stav a příkazy pro vykreslování současně. Pro efektivní řízení všech změn stavů a vykreslování je proto lepší využít více vláken, které samostatně ovládají pouze určité typy funkcí. [36]

Z hlediska paralelizace je dále nejdůležitější částí rozhranní DirectX11 výpočetní shader (compute shader). Ten umožňuje vývojářům využít procesor grafické karty k jiným než grafickým výpočtům (tj. umožňuje to stejné co nVidia CUDA či ATI Stream). Oproti těmto firemním řešením však lze DirectX 11 použít s jakýmkoliv kompatibilním hardwarem. [37]

Klíčové prvky DirectX11:

- plná podpora systému Windows Vista a budoucích verzí Windows;
- kompatibilita s hardwarem pro DirectX 10 a DirectX 10.1;
- technologie umožňující využití procesoru grafické karty jako paralelního výpočetního procesoru (výpočetní shader);
- podpora tesselace (vyhlazování ploch)

6 ZÁVĚR

Tato práce přináší všeobecný přehled současných způsobů tvorby paralelně zpracovávaných programů a vymezuje základní pojmy, které je třeba si při tvorbě těchto aplikací uvědomit.

Z práce vyplývají i jednotlivé rozdíly mezi zmíněnými přístupy. OpenMP či MPI jsou již zaběhlé programovací techniky využívající klasický programovací přístup a hodí se i pro implementaci paralelismu do již existujících aplikací. Klasický programovací přístup využívá i nově implementovaný paralelismus na platformě Microsoft .NET. Při používání těchto přístupů je nutné zejména dbát na správné sdílení a synchronizaci dat. Naproti tomu při využití grafických vývojových prostředí tato nutnost do značné míry odpadá, jelikož tyto záležitosti jsou obsluhovány automaticky a vývojář se proto může plně soustředit na tvorbu aplikace.

Nový a výkonově velmi zajímavý přístup představují též rozhraní nVidia CUDA a ATI Stream pro využití velkého výpočetního výkonu moderních grafických karet, který v některých operacích výrazně převyšuje možnosti hlavních procesorů počítačů. Velmi významným krokem v této oblasti je přijetí standardu OpenCL výrobci GPU, protože sjednotí přístup k paralelizaci na GPU. Stejný přínos lze očekávat i od DirectX 11, které bude na trh uvedeno na podzim 2009.

Přes všechen pokrok, který v paralelizaci za poslední léta proběhl, stále neexistuje jednoduchý přístup, jak efektivně řešit paralelizaci libovolného problému, bez alespoň základních znalostí této problematiky.

SEZNAM POUŽITÉ LITERATURY

[1] The Code Project. *Begin Parallel Programming with OpenMP* [online]. Vydáno: 5.6.2007 [citováno 2009_2_19].
Dostupné z: <http://www.codeproject.com/KB/cpp/BeginOpenMP.aspx>

[2] KOPETSCHKE, Igor. *DPG – paralelní zpracování a hardware* [online]. [citováno 2009_2_23].
Dostupné z: <http://www.nti.tul.cz/wiki/images/7/70/DPG-2.pdf>

[3] KOPETSCHKE, Igor. *DPG – distribuované programování* [online]. [citováno 2009_2_23].
Dostupné z: <http://www.nti.tul.cz/wiki/images/b/bf/DPG-1.pdf>

[4] GERBER, Richard. *Getting started with OpenMP* [online]. 27.6.2007 [citováno 2009_2_25].
Dostupné z: <http://software.intel.com/en-us/articles/getting-started-with-openmp/>

[5] BLAISE, Barney. *OpenMP* [online]. 30.1.2009 [citováno 2009_3_8].
Dostupné z: <https://computing.llnl.gov/tutorials/openMP/>

[6] GATLIN, Kang Su, ISENSEE, Pete. Reap the Benefits of Multithreading without All the Work. *MSDN Magazine* [online]. 2005 říjen [citováno 2009_3_8].
Dostupné z: <http://msdn.microsoft.com/cs-cz/magazine/cc163717(en-us).aspx>

[7] VOLNÝ, Miroslav. *MPI jako standart* [online]. [citováno 2009_3_15].
Dostupné z: <http://www.cs.vsb.cz/jakl/pds/volny/texty/kap2_2.html>

[8] Wikipedia, the free encyclopedia. *Non-Uniform Memory Access* [online]. [citováno 2009_5_11].
Dostupné z <http://en.wikipedia.org/wiki/Non-Uniform_Memory_Access>

[9] Microsoft corporation. *2.3 Parallel construct* [online]. [citováno 2009_4_10]
Dostupné z <http://msdn.microsoft.com/en-us/library/bx15e8hb(VS.80).aspx>

[10] Microsoft corporation. *2.4 Work-Sharing construct* [online]. [citováno 2009_4_10]
Dostupné z <http://msdn.microsoft.com/en-us/library/y0x14tx2(VS.80).aspx>

[11] Microsoft corporation. *2.6 Master and Synchronization Directives* [online]. [citováno 2009_4_12]
Dostupné z <http://msdn.microsoft.com/en-us/library/dkk0z30f(VS.80).aspx>

[12] ČÍRTEK, Pavel, RACEK, Stanislav, KOUTNÝ Tomáš. *Stručný přehled vlastností a funkcí MPI* [online]. Červenec 2006 [citováno 2009_4_10].
Dostupné z: <http://www.kiv.zcu.cz/studies/predmety/ppr/mat_mpi_navod.php>

[13] BLAISE, Barney. *Message Passing Interface (MPI)* [online]. 26.1.2009 [citováno 2009_4_10].
Dostupné z: <https://computing.llnl.gov/tutorials/mpi/>

[14] Microsoft corporation. *.NET Framework Conceptual Overview* [online]. [citováno 2009_5_22]
Dostupné z <http://msdn.microsoft.com/cs-cz/library/zw4w595w(en-us).aspx>

[15] Microsoft corporation. *System.Threading Namespace* [online]. [citováno 2009_5_1]
Dostupné z <http://msdn.microsoft.com/en-us/library/system.threading.aspx>

[16] KEPRT, Aleš. *PARALLEL FX A PARALELNÍ PROGRAMOVÁNÍ NA PLATFORMĚ .NET 4.0* [online]. Listopad 2008 [citováno 2009_5_1].
Dostupné z: <www.keprt.cz/texty/parallelfx.pdf>

[17] Microsoft corporation. *ThreadPool Class* [online]. [citováno 2009_5_1]

Dostupné z <http://msdn.microsoft.com/en-us/library/system.threading.threadpool.aspx>
[18] Wikipedia, the free encyclopedia. *Language Integrated Query* [online]. [citováno 2009_4_25].
Dostupné z <http://en.wikipedia.org/wiki/Linq>
[19] DUFFY, Joe, ESSEY, Ed. Running Queries On Multi-Core Processors. *MSDN Magazine* [online]. 2007 říjen [citováno 2009_4_25].
Dostupné z: <http://msdn.microsoft.com/en-us/magazine/cc163329.aspx>
[20] LEIJEN, Daan, HALL, Judd. Optimize Managed Code For Multi-Core Machines. *MSDN Magazine* [online]. 2007 říjen [citováno 2009_4_25].
Dostupné z: <http://msdn.microsoft.com/cs-cz/magazine/cc163340(en-us).aspx>
[21] National Instruments. *Overcoming Multicore Programming Challenges with LabVIEW* [online]. [citováno 2009_4_15]
Dostupné z <http://zone.ni.com/devzone/cda/tut/p/id/6099>
[22] National Instruments. *Programming Strategies for Multicore Processing: Data Parallelism* [online]. [citováno 2009_4_15]
Dostupné z <http://zone.ni.com/devzone/cda/tut/p/id/6421>
[23] National Instruments. *Programming Strategies for Multicore Processing: Task Parallelism* [online]. [citováno 2009_4_15]
Dostupné z <http://zone.ni.com/devzone/cda/tut/p/id/6420>
[24] The Mathworks. *Parallel Computing Toolbox 4.1* [online]. [citováno 2009_5_17]
Dostupné z <http://www.mathworks.com/products/parallel-computing/description2.html>
[25] POPINCHALK, Seth. *Parallel Computing with Simulink: Model Reference Builds* [online]. 31.3.2009 [citováno 2009_5_17]
Dostupné z <http://blogs.mathworks.com/seth/2009/03/31/parallel-computing-with-simulink-model-reference-builds/>
[26] The Problem Solver. *Windows Workflow Foundation* [online]. [citováno 2009_5_18]
Dostupné z <http://www.windowsworkflowfoundation.eu/default.aspx>
[27] CHAPPELL, David. *The Workflow Way: Understanding Windows Workflow Foundation* [online]. Duben 2009 [citováno 2009_5_18]
Dostupné z <http://msdn.microsoft.com/en-us/library/dd851337.aspx>
[28] ESPOSITO, Dino. Cutting Edge. *MSDN Magazine* [online]. 2006 březen [citováno 2009_5_18].
Dostupné z: <http://msdn.microsoft.com/en-us/magazine/cc163640.aspx>
[29] GPGPU. *About GPGPU.org* [online]. [citováno 2009_4_26]
Dostupné z <http://gpgpu.org/about/>
[30] nVidia. *What is CUDA* [online]. [citováno 2009_4_26]
Dostupné z <http://www.nvidia.com/object/cuda_what_is.html>
[31] nVidia. *NVIDIA CUDA Architecture Introduction & Overview* [online]. Duben 2009 [citováno 2009_5_1].
Dostupné z: <http://developer.download.nvidia.com/compute/cuda/docs/CUDA_Architecture_Overview.pdf>
[32] ATI. *ATI Stream Technology* [online]. [citováno 2009_4_26]
Dostupné z <http://ati.amd.com/technology/streamcomputing/>
[33] ATI. *ATI Stream Computing* [online]. 2009 [citováno 2009_4_26].

Dostupné z: <http://developer.amd.com/gpu_assets/Stream_Computing_Overview.pdf>

[34] ATI. AMD *Stream Computing: Software Stack* [online]. 2007 [citováno 2009_4_26]. Dostupné z: <http://ati.amd.com/technology/streamcomputing/firestream-sdk-whitepaper.pdf>

[35] Khronos Group. *OpenCL* [online]. Únor 2009 [citováno 2009_5_18] Dostupné z <http://www.khronos.org/developers/library/overview/opencl_overview.pdf>

[36] ŠTEFEK, Petr. DirectX 11 - seznámení s budoucností - DirectX 11 - co nás čeká za novinky?. *Svět hardware* [online]. 16.2.2009 [citováno 2009_5_18]. Dostupné z: <http://www.svethardware.cz/art_doc-4AD4EFEBBA7403F7C125755E0063D90D.html>

[37] ŠTEFEK, Petr. DirectX 11 - seznámení s budoucností - DirectX 11 - zázrak jménem Compute Shader (CS). *Svět hardware* [online]. 16.2.2009 [citováno 2009_5_18]. Dostupné z: <http://www.svethardware.cz/art_doc-C7D7A0E67AD06B68C125755E0064BA2B.html>

[38] Wikipedia, the free encyclopedia. *Amdahl's law* [online]. [citováno 2009_4_25]. Dostupné z <http://en.wikipedia.org/wiki/Amdahl%27s_law>

[39] Wikipedia, the free encyclopedia. .*NET Framework* [online]. [citováno 2009_4_25]. Dostupné z <http://en.wikipedia.org/wiki/.NET_Framework>