VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

# IMPLEMENTACE VÝPOČETNĚ NÁROČNÝCH ALGORITMŮ V PARALELNÍM PROSTŘEDÍ GLOBAL ARRAYS

IMPLEMENTATION OF HPC ALGORITHMS IN THE GLOBAL ARRAYS ENVIRONMENT

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE Ondřej Kuchař
AUTHOR

VEDOUCÍ PRÁCE Ing. Jiří Jaroš, Ph.D.
SUPERVISOR

BRNO 2015

## Abstrakt

Komunikace mezi procesy v paralelním prostředí může probíhat buď explicitně mezi procesy, nebo prostřednictvím sdíleného paměťového prostoru. Knihovna Global Arrays druhý zmíněný způsob komunikace pomocí sdíleného paměťového prostoru umožňuje. V této práci byla testována rychlost obou způsobů komunikace na několika zvolených úlohách a poté byly výsledky porovnány. Ve většině případů se knihovna Global Arrays ukázala jako rychlejší a jednodušší na použití. To však neznamená, že je lepší tuto knihovnu použít pro řešení podobných úloh za odlišných podmínek, jako je jiná velikost dat nebo jiný počet použitých procesů.



## Abstract

Communication between processes in a parallel environment can take place either explicitly between processes, or through a shared memory space. Library Global Arrays allows latter means of communication. That is via a shared memory space. In this work we were testing speed of the two modes of communication on several selected tasks, and then the results were compared. In most cases the library Global Arrays proved faster and easier to use. This does not mean that it is better to use this library for solving similar tasks under different conditions like different data size or different number of used processes.



## Klíčová slova

MPI, GA, Global Arrays, paralelní algoritmy, globální adresovací prostor

## Keywords

MPI, GA, Global Arrays, parallel algorithms, global address space


## Citace


KUCHAŘ, Ondřej. Implementace výpočetně náročných algoritmů v paralelním prostředí Global Arrays. Brno, 2016. 34 s. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Jaroš Jiří.

# Implementace výpočetně náročných algoritmů v paralelním prostředí Global Arrays

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Jiřího Jaroše, Ph.D.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Ondřej Kuchař
18. 05. 2016

## Poděkování

*This work was supported by the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070), funded by the European Regional Development Fund and the national budget of the Czech Republic via the Research and Development for Innovations Operational Programme, as well as Czech Ministry of Education, Youth and Sports via the project Large Research, Development and Innovations Infrastructures (LM2011033).*
Chtěl bych poděkovat Ing. Jiřímu Jarošovi Ph.D. za pomoc a rady při psaní této práce a za jeho trpělivost.

# Obsah

# 1 Úvod

Pokud chceme řešit výpočetně náročnou úlohu, tak se paralelizaci nevyhneme. Výkon jednotlivých procesorů sice stále roste, ale zdaleka ne tak rychle, jako poptávka po něm. Při paralelních výpočtech je hlavním problémem komunikace mezi procesy a rozdělení dat mezi ně. V této práci se budeme zabývat 2 způsoby řešení tohoto problému.

Prvním způsobem je zasílání zpráv, kdy jeden proces většinou řídí rozdělování dat potřebných k provedení výpočtů a shromažďování výsledných dat. Rozesílá data ostatním procesům pomocí funkcí poskytnutých danou knihovnou a také se podílí na samotném výpočtu. Tento způsob může vést k velice dlouhému a složitému kódu, protože je potřeba specificky určovat cíle a zdroje komunikace mezi jednotlivými procesy.

Druhým způsobem je využití sdíleného adresového prostoru, kdy každý proces má přístup ke všem datům, která se v tomto prostoru nacházejí. Každý proces si pomocí funkcí dané knihovny zkopíruje data, která potřebuje k výpočtu a po jeho provedení výsledná data uloží zpět do sdíleného prostoru. Výhoda tohoto řešení spočívá v usnadnění řízení komunikace mezi jednotlivými procesy, které v podstatě celé zaštiťují knihovní funkce. Je potřeba pouze řešit situaci, kdy se stejným adresovým prostorem pracuje více procesů najednou.

V této práci se podrobněji podíváme na knihovny řešící oba způsoby komunikace, na náročnost implementace datové komunikace a na výkonový rozdíl. Výsledkem této práce tedy bude, zdali implementace datové komunikace mezi procesory s pomocí sdíleného adresového prostoru bude jednodušší a zdali vyváží případný výkonový rozdíl.

# 2 Distribuovaný systém

Distribuovaný systém je několik samostatných počítačů (jednotek, uzlů) propojených rychlou sítí, pomocí které mohou sdílet systémové zdroje a posílat potřebná data ve formě zpráv. Jednotlivé uzly jsou většinou bez periferií jako klávesnice, myš nebo monitor. Operační systém běžící na těchto uzlech je optimalizován na propustnost dat a rychlost zpracování jednotlivých úloh.

# 3 Pracovní prostředí

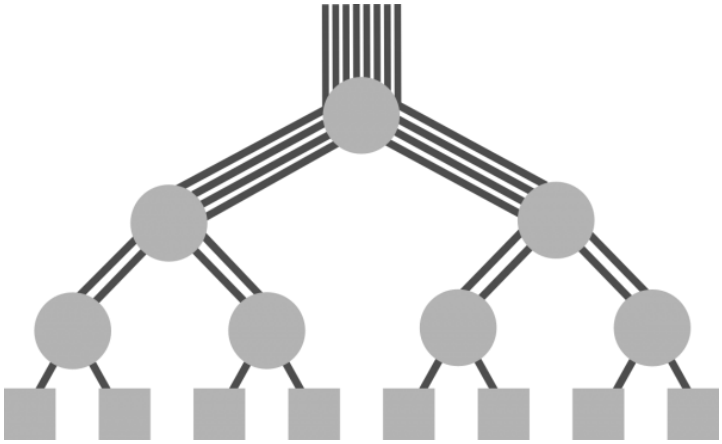
Všechny programy a skripty byly spouštěny na superpočítači Anselm[1]. Tento superpočítač se nachází na Ostravské technické univerzitě.

Pro spuštění programů na jiném linuxovém stroji je potřeba mít nainstalovanou knihovnu Global Arrays[2], OpenMPI[3] a knihovnu HDF5[4] pro paralelní zápis do souboru.

[1] https://docs.it4i.cz/anselm-cluster-documentation
[2] http://hpc.pnl.gov/globalarrays/download/
[3] https://www.open-mpi.org/software/ompi/v1.10/
[4] https://www.hdfgroup.org/HDF5/release/obtain5.html

## 3.1 Specifikace

Anselm má celkem 209 výpočetních serverů (uzlů)[5]. 180 uzlů je bez akcelerace. K 23 uzlům je připojena grafická výpočetní jednotka Nvidia Kepler K20, 4 uzly jsou akcelerovány pomocí výpočetních karet Intel Xeon Phi P5110 a zbylé 2 uzly jsou takzvané „fat-nodes“, které mají k dispozici 512GB operační paměti. Pro naše účely budou stačit uzly bez akcelerace, tedy 2x Intel Sandy Bridge E5-2665, 2.64GHz (2x 8 jader) a 64GB operační paměti na uzel. Teoreticky tedy máme k dispozici celkem 2880 výpočetních jader a 11.25 TB operační paměti. Všechny uzly jsou propojeny vysokorychlostní sítí[6] s nízkou latencí technologie Infiniband s plně neblokující topologií fat-tree schopné přenosové rychlosti až 3600MB/s (při použití Infiniband protokolu).



*Obr 3-1. Ilustrace fat-tree topologie*

## 3.2 Úložiště

Na Anselmu je k dispozici několik různých souborových systémů. Systémy HOME a SCRATCH jsou přístupné jak z přihlašovacích, tak z výpočetních uzlů.

HOME poskytuje každému uživateli 250GB úložného prostoru pro data a teoretická propustnost dosahuje 2GB/s. Tato data zůstávají uložena do vypršení platnosti uživatelského účtu a jsou zálohována.

SCRATCH je souborový systém, který se doporučuje používat pro větší objem dat generovaných nebo zpracovávaných při výpočtu. Uživatelská kvóta je nastavena na 100TB a teoretická propustnost je 6GB/s. Pokud se s těmito daty nemanipuluje více než 90 dní, jsou ze systému smazána. Při testování budeme využívat souborový systém HOME.

## 3.3 Přihlašování a přenos souborů

Přihlášení[7] na Anselm lze provést pouze prostřednictvím SSH protokolu. Uživatel se přihlašuje na jeden ze dvou přihlašovacích serverů. Pro přihlášení lze použít terminál (Linux, Mac) nebo Putty (Windows). Lze také využít VNC server běžící na Anselmu a přihlásit se do grafického režimu (viz. Příloha 1).

[5] https://docs.it4i.cz/anselm-cluster-documentation/compute-nodes
[6] https://docs.it4i.cz/anselm-cluster-documentation/network
[7] https://docs.it4i.cz/anselm-cluster-documentation/accessing-the-cluster/shell-and-data-access

Přenos souborů lze provádět pomocí scp nebo sftp klienta. Lze také vzdálený souborový systém připojit pomocí příkazu `sshfs` (Mac, Linux) nebo pomocí softwaru Sshfs Manager (Windows) jako výměnné úložiště. Maximální teoretická přenosová rychlost je 160 MB/s za předpokladu, že je spojení realizováno minimálně pomocí 10GB linky a že máme dostatečně rychlý procesor.

## 3.4 Spouštění výpočetních úloh

Anselm používá pro plánování, rozdělování a řízení jednotlivých prací mezi uzly software PBS Professional[8]. Všechny uzly jsou dedikované, nemusíme se tedy obávat, že budeme při běhu programu s někým sdílet výpočetní výkon. Alokovat (zabrat) výpočetní zdroje lze dvěma způsoby. Prvním způsobem zadáme přímo program, který chceme na vybraných uzlech spustit. Např.:

```
qsub -q qprod -A OPEN-3-11 -l select=4:ncpus=16:mpiprocs=16:ompthreads=1,
walltime=03:00:00 ./mpi_avg.x
```

Tento příkaz zajistí alokování 4 uzlů (16 jader na uzel), nastaví maximální dobu provádění výpočtu na 3 hodiny a spustí program `mpi_avg.x`. Po uplynutí maximální doby provádění je výpočet zastaven, ať už byl dokončen nebo ne. Část příkazu `mpiprocs=16:ompthreads=1` určuje, že maximální počet MPI procesů na uzel je 16 a že každý proces bude mít pouze jedno vlákno.
Druhý způsob je interaktivní režim. Po přidělení zdrojů jsme přihlášeni přímo na výpočetní uzel a zde můžeme ručně spouštět vybrané úlohy. Tento způsob lze také využít při spouštění grafického rozhraní ladících aplikací.

# 4 Message Passing Interface

V první řadě bych chtěl uvést, že Message Passing Interface[9] (MPI) není knihovna, ale standart. Tento standart zaručuje stejnou syntaxi a sémantiku základních funkcí použitých pro zajištění komunikace [1]. MPI poskytuje rozhraní pro komunikaci mezi procesy prostřednictvím zasílání zpráv. Zasílání zpráv je jednou z hlavních metod programování paralelních aplikací, které se v dnešní době používají. Pro implementaci datové komunikace paralelního algoritmu pomocí standartu MPI však musí programátor manuálně zajistit komunikaci mezi jednotlivými procesy. Realizace programů se může tedy značně zkomplikovat.

Různé implementace uvedeného standartu můžou mít své nadstandartní rozšíření. Na Anselmu jsou k dispozici tyto knihovny implementující standart MPI: Bullx MPI, Intel MPI, Lam MPI, MVAPICH2 a OpenMPI. V našem případě bude využita poslední uvedená knihovna, tedy knihovna OpenMPI.

## 4.1 Komunikátory

Procesy lze rozdělovat do různých skupin, ve kterých mohou mezi sebou komunikovat. Při spuštění MPI programu je automaticky vytvořen komunikátor `MPI_COMM_WORLD`, který obsahuje všechny procesy.

[8] https://docs.it4i.cz/pbspro-documentation
[9] https://www.mpi-forum.org/docs/docs.html

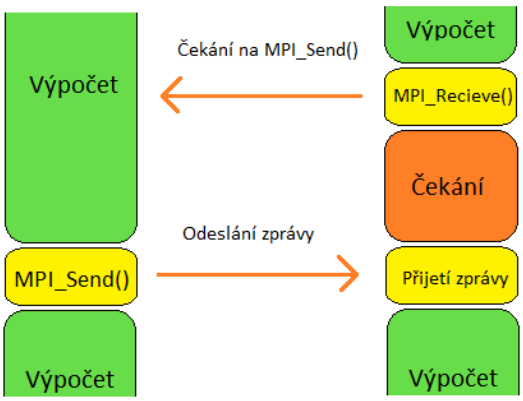
## 4.2 Point-to-point komunikace

Mezi základní a nejčastěji používané funkce v MPI patří dvojice funkcí `MPI_Send()` a `MPI_Recv()`. Těmito funkce lze zajistit přenos dat z jednoho procesu do druhého. Jedná se o takzvané lokální funkce, tedy funkce, které vykonávají jen určené procesy. Při neblokující komunikaci je nutné zajistit, aby nedošlo ke stavu, kdy jeden proces odešle zprávu druhému procesu, ale ten ji už nepřijme. V tomto případě dojde k zablokování běhu programu.

### 4.2.1 Blokující a neblokující komunikace

Point-to-point komunikaci lze rozdělit na blokující a neblokující [1]. Oba způsoby si můžeme vysvětlit na jednoduchém příkladu komunikace 2 procesů *p0* a *p1*.

V případě blokující komunikace posílá proces *p0* zprávu procesu *p1* zavoláním funkce `MPI_Send()`. Proces *p1* musí zprávu přijmout zavoláním funkce `MPI_Recv()`. Pokud však proces *p1* tuto funkci zavolá dříve, než stačí proces *p0* zprávu odeslat, tak bude proces *p1* čekat, dokud proces *p0*, zprávu neodešle.




*Obr 4-1Ukázka problému blokující komunikace*

Při neblokující komunikaci zavolá proces *p0* neblokující variantu funkce pro odeslání zprávy `MPI_Isend()` a uloží si požadavek na odeslání. V tomto případě proces *p0* nečeká, až proces *p1* zprávu přijme, ale okamžitě pokračuje ve vykonávání programu. Pro zajištění správného odeslání dat je pak zavolána funkce `MPI_Wait()` a té je předán dříve uložený požadavek. Funkce pak běh programu přeruší, dokud není požadavek vyřízen. Analogicky, když proces *p1* zavolá neblokující variantu funkce pro přijetí zprávy `MPI_Irecv()`, nečeká na přijetí zprávy, ale pokračuje ve vykonávání programu. Proces *p1* tedy může mezitím provádět výpočty, které nejsou závislé na obsahu očekávané zprávy a to zda byla zpráva přijata, zkontrolovat později pomocí funkce `MPI_Wait()`.

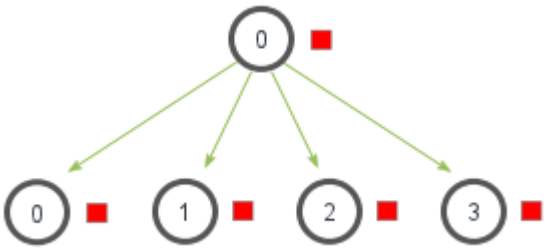
Dále také záleží na tom, jestli systém poskytuje vyrovnávací paměť (buffer). Pokud ano, tak se po odeslání zprávy procesem *p0* tato zpráva uloží do poskytnuté vyrovnávací paměti a proces *p0* pokračuje ve výpočtech [1]. Pokud ne, tak musí proces *p0* čekat, dokud proces *p1* nezavolá funkci pro přijetí zprávy. Pak mluvíme o synchronní komunikaci. Proces *p1* musí dát vědět, že je připraven zprávu přijmout.

# 4.3 Kolektivní komunikace

Kolektivní komunikace je realizována funkcemi, na kterých se podílejí všechny procesy v daném komunikátoru a je tedy nutné, aby každý proces v daném komunikátoru tuto funkci zavolal [1].

## 4.3.1 Broadcast

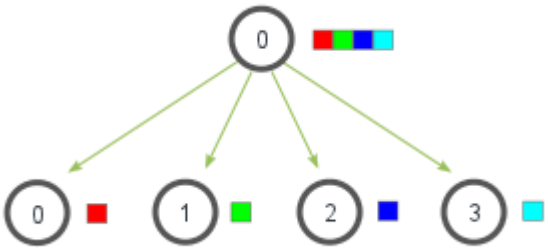
Pokud chceme odeslat stejná data všem procesům, můžeme k tomu použít funkci `MPI_Bcast()`. Tato funkce odešle data všem procesům v daném komunikátoru [1]. Tuto funkci musí zavolat každý proces, který se v komunikátoru nachází, se stejným parametrem pro komunikátor a pro zdrojový proces.




*Obr 4-2. Jednoduchá ukázka principu funkce MPI_Bcast*
*obrázek převzatý z [2]*

## 4.3.2 Scatter a Scatterv

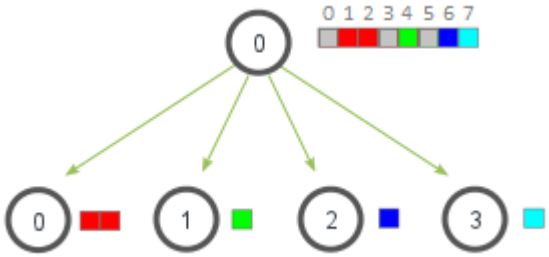
Pokud máme data, která chceme rozdělit a jednotlivé části rozeslat všem procesům, pak použijeme funkci `MPI_Scatter()` [1]. Této funkci se jako parametr zadá velikost bloku dat (počet), která budou přijata jednotlivými procesy. Jako příklad můžeme uvést 4 procesy, mezi kterými je potřeba data rozdělit. Proces *p0* potřebuje rozdělit a rozeslat pole (4) hodnot všem procesům, tak aby každý proces obdržel 1 hodnotu. Proces *p0* tedy zavolá funkci `MPI_Scatter()` s parametrem určujícím počet hodnot, které má každý proces přijmout (v našem příkladu to bude 1).




*Obr 4-3. Jednoduchá ukázka principu funkce MPI_Scatter*
*obrázek převzatý z [2]*

Funkce `MPI_Scatterv()` nám navíc umožňuje nepravidelné rozložení dat. Této funkci se totiž parametr pro určení počtu přijatých hodnot předává jako pole, které určuje, který proces obdrží kolik hodnot. Navíc se této funkci předává nový parametr typu pole, který určuje odsazení hodnot od začátku zdrojového pole odesílajícího procesu. V následujícím příkladu si můžeme ukázat volání

funkce `MPI_Scatterv()`. Parametr pole určující počet přijatých prvků na jednotlivých procesech má velikost 4 (odesíláme data 4 procesům) a jeho hodnoty jsou {2, 1, 1, 1}. Pole, které určuje odsazení

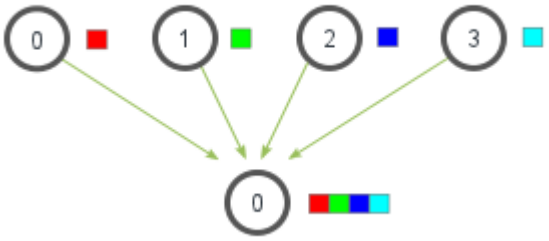



*Obr 4-4Jednoduchá ukázka principu funkce MPI_Scatterv*

hodnot od počátku zdrojového pole má také velikost 4 a jeho hodnoty jsou {1, 4, 6, 7}. Výsledek této operace můžeme vidět na následujícím obrázku.

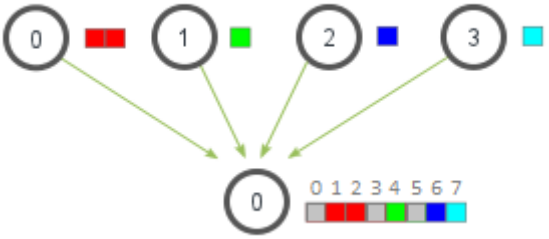
### 4.3.3 Gather a Gatherv

Funkce `MPI_Gather()` je opakem funkce `MPI_Scatter()`. Pokud máme data v každém procesu a chce je získat (sloučit), zavoláme funkci `MPI_Gather()`. Před provedením této akce se musíme ujistit, že máme na procesu, který data přijímá, dostatečně velký alokovaný prostor. [1]




*Obr 4-5. Jednoduchá ukázka principu funkce MPI_Gather*
*obrázek převzatý z [2]*

Obdobně jako funkce `MPI_Scatterv()` umožňuje funkce `MPI_Gatherv()` určit kolik hodnot od kterého procesu přijmout a jaké odsazení mají mít ve výsledném poli na přijímajícím



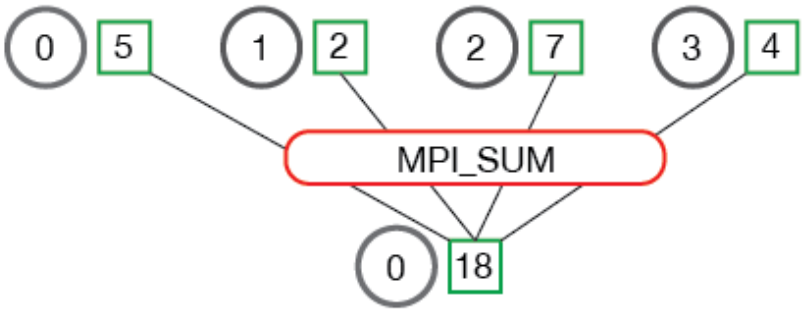

procesu. Opět si ukážeme na příkladu. Pole určující počet přijatých prvků má velikost 4 a jeho hodnota je {2, 1, 1, 1}. Pole určující odsazení má také velikost 4 a jeho hodnota je {1, 4, 6, 7}. Následující

*Obr 4-6Jednoduchá ukázka principu funkce MPI_Gatherv*

obrázek znázorňuje volání funkce `MPI_Gatherv()` s uvedenými parametry.

### 4.3.4 Reduce

Funkce `MPI_Reduce()` je stejná jako `MPI_Gather()` s tím rozdílem, že jako další parametr je uvedena operace, která se má nad přenášenými daty provést a výsledek je zapsán do jediné hodnoty [1]. Například sečtení všech přenášených hodnot se provede funkcí `MPI_Reduce()` se zadaným parametrem `MPI_SUM`.




*Obr 4-7. Ukázka použití funkce MPI_Reduce s parametrem MPI_SUM obrázek převzatý z [2]*

## 4.4 Datové typy

Každé funkci, která nějakým způsobem pracuje s daty, se předává parametr určující datový typ přenášených dat. V následující tabulce jsou uvedeny datové typy, které rozhraní MPI podporuje a jejich odpovídající datové typy v jazyku C [2].

| MPI datový typ | C datový typ |
|---|---|
| MPI_SHORT | short int |
| MPI_INT | int |
| MPI_LONG | long int |
| MPI_LONG_LONG | long long int |
| MPI_UNSIGNED_CHAR | unsigned char |
| MPI_UNSIGNED_SHORT | unsigned short int |
| MPI_UNSIGNED | unsigned int |
| MPI_UNSIGNED_LONG | unsigned long int |
| MPI_UNSIGNED_LONG_LONG | unsigned long long int |
| MPI_FLOAT | float |
| MPI_DOUBLE | double |
| MPI_LONG_DOUBLE | long double |
| MPI_BYTE | char |

Pokud chceme využít funkce pro odesílání dat, odesílaná data musí být v paměti uložena kontinuálně. MPI umožňuje vytváření vlastních odvozených datových typů.

### 4.4.1 MPI_Type_contiguous

Tato funkce umožňuje vytvořit datový typ, který určuje násobek jiného datového typu, kterého hodnoty jsou v paměti uloženy vedle sebe [3]. Lze tak například jednoduše vytvořit datový typ celé matice nebo jen jednoho řádku.

# 5 Global Arrays

Knihovna Global Arrays[10] (GA) vznikla v roce 1994 v Environmental Molecular Sciences Laboratory (EMSL) v Pacific Northwest National Laboratory (PNNL). Jejími hlavními tvůrci jsou Jarek Nieplocha, Manojkumar Krishnan, Bruce Palmer a Vinod Tipparaju.

Tato knihovna umožňuje vytvářet globální pole (prostor) distribuované přes několik výpočetních uzlů [4]. Zachovává však lokalitu dat a to pomocí jednoduchých funkcí pro zjišťování, kterému procesu/procesům patří konkrétní část globálního prostoru.

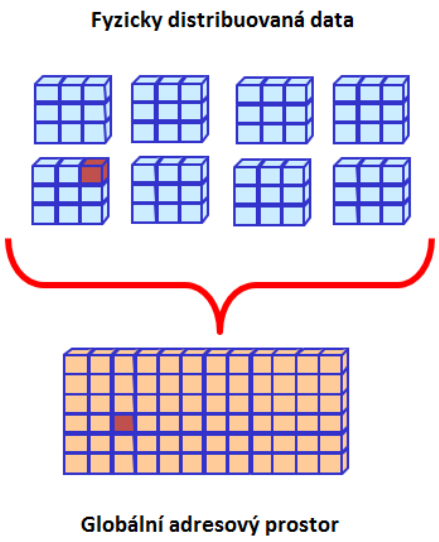
GA má rozhraní pro Fortran, C, C++ a Python. Každé vytvořené pole podporuje až 7 dimenzí ve Fortranu a ještě více v C. V našem případě budeme používat rozhraní pro jazyk C.

Všechny operace se v GA dělí na kolektivní a lokální.

- Kolektivní operace vyžadují „spolupráci“ všech zúčastněných procesů. K těmto operacím patří vytváření, mazání a kopírování globálních polí. Patří sem i data-paralelní operace jako například násobení matic.
- Lokální operace mohou být prováděny nezávisle na ostatních procesech. Mezi tyto operace patří lokální přístup k prvkům pole a získávání a zápis dat do vzdálených lokalit.

Základním cílem GA je vytvořit globální adresový prostor v distribuovaném systému se zachováním lokality dat a poskytnutí jednoduchého rozhraní pro manipulaci s nimi. Zachování lokality dat je umožněno například funkcí, která zjišťuje, jaký úsek globálního adresového prostoru patří danému procesu.

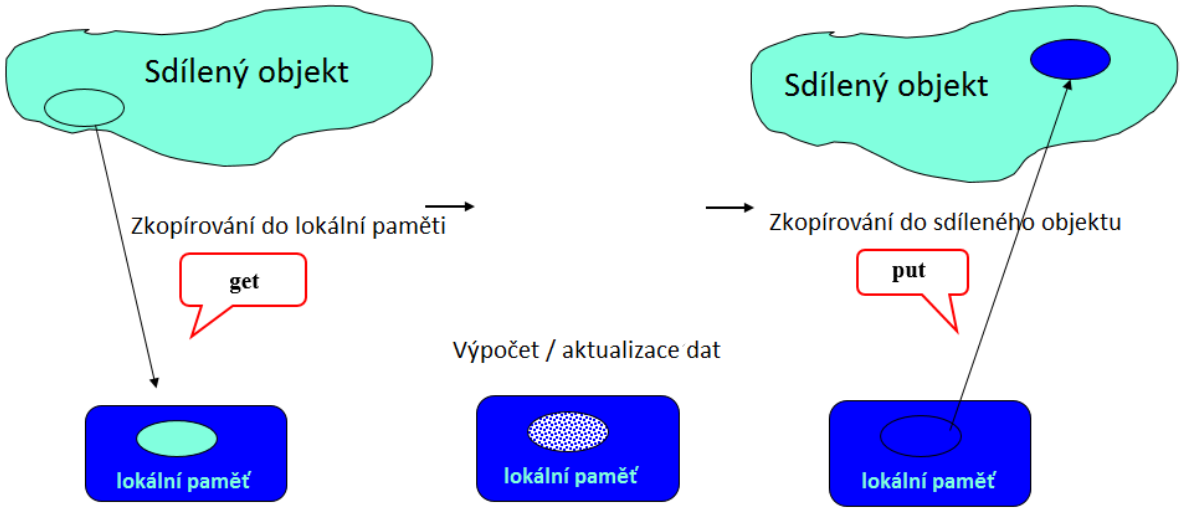
[10] http://hpc.pnl.gov/globalarrays




*Obr 5-1Rozdělení globálního adresového prostoru*
*obrázek převzatý z [4]*

# 5.1 Výpočetní model a přístup k datům

Základní přístup k programování aplikací určených pro systém se sdílenou pamětí je získání dat z globálního adresového prostoru, provedení lokálního výpočtu a poté vrácení vypočtených dat. Pro



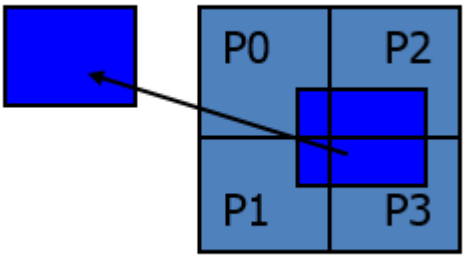

*Obr 5-2 Ilustrace výpočetního modelu GA*

tyto účely jsou v GA určeny funkce `NGA_Get()` a `NGA_Put()`.

Funkci `NGA_Get()` se předávají parametry, které určují, ze kterého globálního pole se data budou kopírovat, horní a dolní mez pro každou dimenzi, ukazatel na cílové pole, kam se data mají zapsat a velikost bloku dat. Pokud zadanou část globálního pole vlastní více procesů, GA zajistí veškerou komunikaci na pozadí. Tato skutečnost je velikým usnadněním pro programátora. Při získávání dat se nemusí starat o každý přenos mezi procesy.

Funkci `NGA_Put()` se předávají stejné parametry jako předchozí funkci. Jen tok dat je opačný. Kopírujeme data z lokálního pole do pole globálního. Program se z této funkce může vrátit

ještě předtím, než do globálního pole dorazí všechna data. Pokud toto chování vytváří nějaký problém, lze ho vyřešit dvojicí funkcí `GA_Init_fence()` a `GA_Fence()`.




*Obr 5-3 Příklad použití funkce NGA_Get() obrázek převzatý z [4]*

`GA_Init_fence()` zajistí, že při veškeré následné komunikaci je proces, který komunikaci provádí zablokován, do té doby, než jsou skutečně všechna data doručena do cíle. Toto chování platí pro veškerou komunikaci (směrem do globálního pole) dokud není zavolána funkce `GA_Fence()`. Před voláním této funkce musí být vždy zavolána funkce `GA_Init_fence()`. Tyto funkce mohou být i vnořené. Počet volání `GA_Fence()` musí odpovídat počtu volání `GA_Init_fence()`.

Další užitečnou funkcí je funkce `NGA_Acc()`, která provádí atomickou operaci sčítání, kdy obsah lokálního pole sečte s určeným úsekem pole globálního. Je možné také přičítanou hodnotu vynásobit koeficientem *alfa*, který se zadává jako parametr funkce. Takto lze jednoduše napodobit chování funkce `MPI_Gather()` se zadanou operací MPI_SUM.

## 5.2 Využití lokality dat

Pro využití lokality dat máme k dispozici funkci `NGA_Distribution()`, která vrací informace o tom, jakou část zadaného globálního pole zadaný proces vlastní [5]. Tato informace je uložena ve formě dvou polí, ve kterých jsou uloženy hodnoty dolní a horní meze pro každou dimenzi globálního pole. Pokud zadaný proces v globálním poli nevlastní žádná data, je jako dolní mez vrácena hodnota `-1` a jako horní mez hodnota `-2`, pro všechny dimenze.

Pokud víme, jakou část globálního pole proces vlastí, můžeme použít funkci `NGA_Access()`, která vrací ukazatel na data umístěná v globálním poli a lze tedy vynechat kopírování dat do lokálního pole. Poté můžeme s daty normálně manipulovat a nemusíme se starat o žádný přenos. Je ovšem nutné zajistit výlučný přístup k těmto datům, jelikož jsou přístupná všem procesům. Funkci `NGA_Access()` musí následovat funkce `NGA_Release()`, pokud byla data jen pro čtení nebo `NGA_Release_update()`, pokud byla data zpřístupněna i pro zápis. Tímto způsobem lze přistupovat pouze k datům, která jsou vlastněna zadaným procesem. Je tedy doporučeno vždy před voláním `NGA_Access()` zavolat funkci `NGA_Distribution()`.

## 5.3 Vytváření globálních polí

K vytvoření globálního pole s rovnoměrnou distribucí dat slouží funkce `NGA_Create()`. Při vytváření musíme funkci zadat datový typ vytvářeného pole, počet dimenzí, počet prvků v každé dimenzi a minimální velikost celků, do kterých má být každá dimenze rozdělena [4]. Pokud chceme dvojrozměrné globální pole rozdělit například na vodorovné pruhy, stačí nastavit velikost bloku pro

*Obr 5-4 Rovnoměrné rozložení dat v globálním poli*

druhou dimenzi rovnu počtu prvků v druhé dimenzi a velikost bloku pro první dimenzi na `-1`. Pokud chceme rozdělit pole vertikálně, tak stačí velikost bloku pro první dimenzi nastavit rovnu počtu prvků v první dimenzi. Každé globální pole je reprezentováno celým číslem a začíná od `-1000`.

# 6 Porovnání výkonu

Pro porovnání výkonu obou knihoven budeme u každého problému měřit pouze čas datové komunikace, čas výpočtu a čas celkového běhu programu. Řešení každého problému se v obou knihovnách bude lišit pouze minimálně a změny nastanou jen ve způsobu distribuce dat. Získáme tak přehled o časové náročnosti režie dat a budeme schopni jednoduše vyhodnotit výkonový rozdíl mezi GA a MPI.

Očekávané výsledky jsou takové, že řešení pomocí GA bude pomalejší než řešení pomocí MPI z důvodu větší režie probíhající na pozadí komunikace. V MPI explicitně určujeme, jakým způsobem a jakému procesu se mají daná data zaslat. V případě GA si však knihovna musí tyto informace zjistit sama a může tedy vznikat zpoždění před provedením samotné komunikace.

## 6.1 Spouštění úloh

Před spuštěním každé úlohy vytvoříme složku s názvem `run_<název úlohy>_<řešení>`. Přičemž `<řešení>` je buď `ga` nebo `mpi`. Rozdělení složek podle řešení nám umožní naráz spustit jak úlohu řešenou pomocí GA, tak úlohu řešenou pomocí MPI. Do této složky přeneseme skript zajišťující správnou alokaci zdrojů na superpočítači Anselm, který nahraje potřebné moduly a spustí skript, který se stará o spuštění samotné úlohy.

Skript pro spuštění úlohy je napsán v jazyce Python a jako parametry se mu předává soubor, určující parametry úlohy a relativní cesta ke spustitelnému souboru samotné úlohy. Přestože se jedná o relativní cestu, je nutné ji zadat se zpětným lomítkem na začátku.

Soubor s parametry pro spuštění úlohy obsahuje informace o počtu procesů, které má spouštěná úloha využít při výpočtu, velikosti vstupních dat, které má úloha použít a počet iterací, které se mají provést.

Po dokončení každé úlohy jsou do příslušných souborů uloženy výsledky měření. Jedná se o tyto soubory:

- res – obsahuje naměřený celkový čas
- res_comm – obsahuje průměrný čas doby strávené komunikací
- res_eval – obsahuje průměrný čas doby, který úloha strávila samotným výpočtem

- res_it – obsahuje průměrný čas jedné iterace

Skript, tyto soubory otevře a uloží si jejich obsah, jelikož je jejich obsah přepsán při každém dalším spuštění úlohy. Informace ze souboru určujícím parametry spouštěné úlohy se použijí pro vytvoření názvů sloupců a řádků výsledného souboru vygenerovaným spouštějícím skriptem. Tento soubor je vygenerován pro každé měření a je poté naplněn daty získanými z příslušných výstupních souborů daných úloh a je uložen ve formátu `csv`. Jednotlivá data jsou v souboru oddělena středníkem. Jako název souboru se použije `<název spustitelného souboru úlohy>_<název souboru s výsledky úlohy>_csv.csv`. Získáme tak výsledky dané úlohy v uceleném formátu, se kterým se dá dále snadno pracovat.

## 6.2 Vyhodnocování výsledků

Při vyhodnocování výsledků použijeme výše zmíněné `csv` soubory. Sestrojíme graf, který bude obsahovat výsledky měření celkové doby běhu programu pro všechny parametry a zobrazíme si tak přínos běhu programu na více procesech.

Nakonec sestrojíme graf, který zobrazí porovnání výsledných časů programů používajících GA a MPI. Pro tento graf zvolíme výsledky pouze pro největší velikost vstupních dat.

Pro přesnější porovnání vypočítáme procentuální rozdíl celkové doby běhu programu v každém bodě prvního grafu mezi celkovou dobou běhu GA a celkovou dobou běhu MPI. Procentuální rozdíl získáme následovně

$$P = (T_{ga} - T_{mpi})/ABS(T_{ga}) * 100$$

kde $T_{ga}$ je celkový čas běhu programu, ve kterém byla daná úloha řešena pomocí GA a $T_{mpi}$ je celkový čas běhu programu, ve kterém byla úloha řešena pomocí MPI. Pak si lze jednoduše zobrazíme procentuální rozdíl celkové doby běhu obou implementací. Poté provedeme sumu procentuálních rozdílů ve všech měřených bodech a vydělíme počtem bodů. Získáme tak průměrný procentuální rozdíl celkové doby trvání běhu obou programů.

Pokud budeme potřebovat vysvětlit nějaké nečekané úkazy, použijeme i výsledky z jiných měření.

## 6.3 Počítání průměru vektorů

Malá úloha, na které si ukážeme základní použití jednoduchých funkcí obou knihoven. Vstupem této úlohy je jednorozměrné pole dat, které je rozděleno a rozesláno jednotlivým procesům. Řídící proces vygeneruje pole náhodných hodnot $R$ a rozešle části pole jednotlivým procesům. Tyto části pole označíme $SR$. Každý proces provede dílčí sumu získaných dat a řídící proces poté z těchto dílčích sum provede celkovou sumu a výsledek vydělí celkovým počtem elementů.

Pokud si uvedený výpočet rozepíšeme, tak bude vypadat následovně.

1) Dílčí sumu $S_p$ na zadaných procesech získáme pomocí vzorce

$$S_p = \sum_{i=0}^{r} SR_p[i] = SR_p[0] + SR_p[1] + \cdots + SR_p[r]$$

Kde $p$ je číslo procesu, $SR_p$ je pole dílčích hodnot procesu $p$, $r$ je počet hodnot, které bylo procesu $p$ přiřazeno a $i$ označuje index pole $SR$.

2) Celkovou sumu $S$ ze všech procesů pak získáme následovně

$$S = \sum_{i=0}^{p} S_i = S_0 + S_1 + \cdots + S_p$$

kde $p$ je celkový počet procesů a $S_p$ je dílčí suma procesu $p$.

3) A nakonec řídící proces vypočítá průměr celého vektoru takto

$$\bar{S} = \frac{1}{n} S$$

kde $n$ označuje celkový počet hodnot.

### 6.3.1 MPI

V MPI budeme jako základní funkce používat `MPI_Scatterv()` a `MPI_Reduce()`. Jak již bylo popsáno, řídící proces vytvoří pole, které naplní náhodnými hodnotami. Jako maximální hodnota bylo zvoleno číslo 1.0, aby se zamezilo problému přetečení při počítání velkých polí.
Dále potřebujeme zjistit, kolik elementů pole bude kterému procesoru zasláno. Jednoduchým výpočtem zajistíme rovnoměrné rozložení elementů mezi procesy. Jeho průběh si nyní ukážeme.

1) Vydělíme zadanou velikost pole počtem procesů. Výsledek dělení si uložíme do proměnné `sub_array_size` a zbytek po dělení do proměnné `remainder`. Hodnotu odsazení od počátku pole nastavíme na 0 a uložíme do proměnné `displ`.
2) Dále budeme iterovat přes proměnnou `i` od 0 do `p`, kde `p` udává počet procesů.
   - pro proces `i` nastavíme počet elementů roven proměnné `sub_array_size`
   - pokud je `i` menší než hodnota proměnné `remainder`, tak k počtu elementů přičteme 1
   - odsazení od počátku pole pro proces `i` nastavíme na hodnotu proměnné `displ`
   - k proměnné `displ` přičteme počet elementů pro proces `i`

Po provedení tohoto výpočtu tak máme k dispozici informace o rovnoměrném rozdělení pole mezi všemi procesy i s hodnotami odsazení od počátku pole pro každý proces.

Poté odešle všem procesů příslušné části pole pomocí funkce `MPI_Scatterv()` a jako parametry použijeme informace o rozdělení mezi procesy získané výpočtem popsaným výše. Jednotlivé procesy poté provedou výpočet dílčích sum nad polem, které jim bylo zasláno.

Po vypočtení dílčích sum jednotlivými procesy využijeme funkce `MPI_Reduce()` a jako akci zvolíme `MPI_SUM`. Na řídící proces tak hodnoty dorazí již sečtené a zbývá již jen vydělit tento součet celkovým počtem elementů.

### 6.3.2 GA

Při použití GA budeme pracovat s 2 globálními poli. Nazveme je *a* a *b*. Pole *a* naplníme náhodně vygenerovanými hodnotami a pole *b* použijeme pro uložení celkové sumy. Obě pole budou mít jednu dimenzi a pole *b* bude mít pouze 1 element.

V případě řešení úlohy pomocí GA nastává změna v logice datové komunikace. S vytvořením globálních polí vzniká mezikrok, jelikož máme nyní centralizované úložiště.

Globální pole se vytvoří funkcí `NGA_Create()` a jako datový typ použijeme float. GA pole rovnoměrně rozdělí a není tedy nutné provádět výpočet určující rozdělení pole. Funkci na vytvoření globálního pole volají všechny zúčastněné procesy.

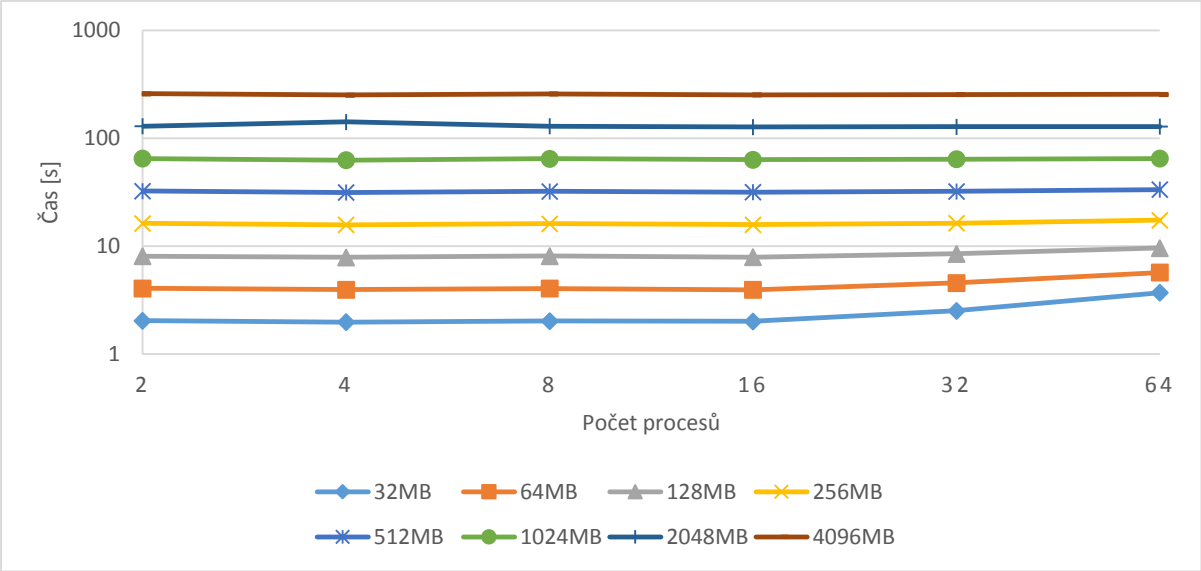
Řídící proces vygeneruje náhodné hodnoty a tyto hodnoty nahraje do pole *a*. K tomu použije funkci `NGA_Put()` a jako horní hraniční parametr použije celkovou velikost pole *a*.

Jednotlivé procesy si potom z pole *a* zkopírují hodnoty, které jim patří. Pro získání informace o tom, jakou část globálního pole si mají dané procesy zkopírovat, musí zavolat funkci `NGA_Distribution()`. Tak získají informace o tom, jaká část globálního pole jim patří. Poté si zkopírují svou část pole do lokálního úložiště pomocí funkce `NGA_Get()` a vypočítají dílčí sumu. Tuto sumu poté pomocí funkce `NGA_Acc()` přičtou do jediného elementu pole *b*. Tím se do pole *b* dostane celková suma. Uvedená operace je atomická a není tedy třeba řešit výlučný přístup.

Řídící proces si zkopíruje výslednou sumu z pole *b* a vydělí ji celkovým počtem hodnot.

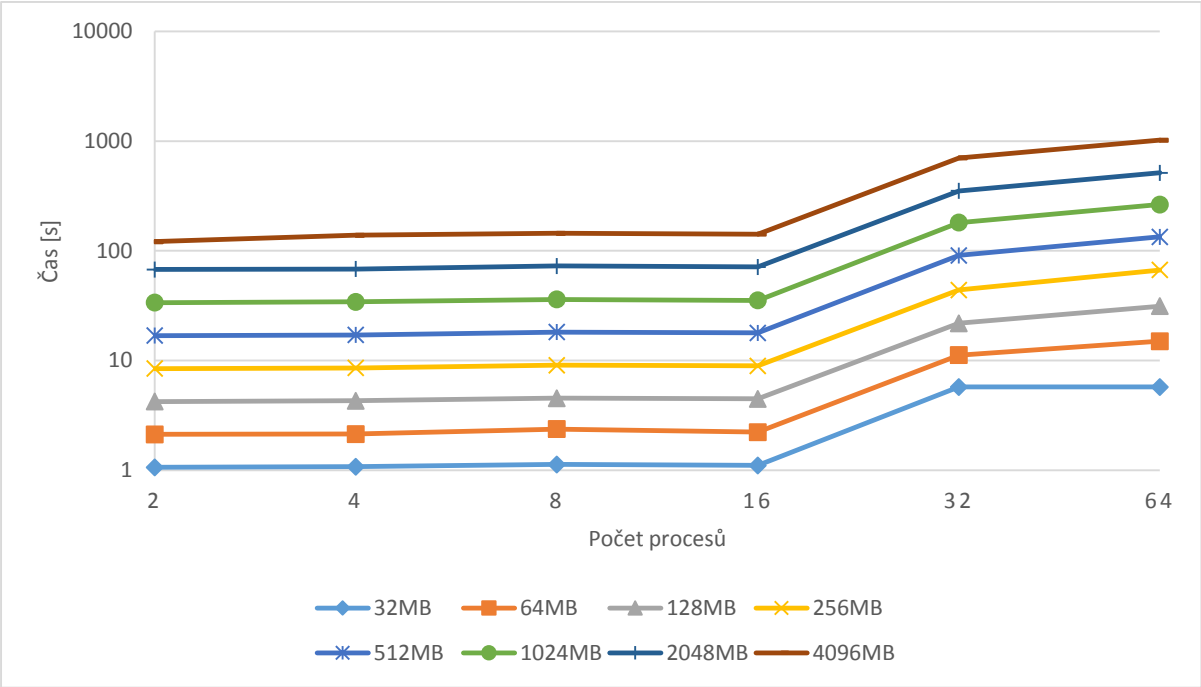
### 6.3.3 Spuštění a výsledky

Jako parametry spouštěné úlohy byly zadávány velikost dat [MB], počet řešících procesů a počet iterací výpočtu. Na následujících grafech se můžeme podívat na výsledky měření celkového času úlohy řešené prostřednictvím MPI a prostřednictvím GA.




*Obr 6-1Graf reprezentující naměřené výsledky programu používajícího MPI*

Jak můžeme vidět, tak tento typ úlohy není vhodným příkladem efektivní implementace paralelního výpočtu. Výsledky jsou pro různé počty procesů stejné a ve větším počtu procesů spojeným s větším množstvím dat, je dokonce řešení pomalejší, z důvodu vysoké časové náročnosti komunikace. To může být způsobeno faktem, že při použití více jak 16 procesů je komunikace prováděna i mezi více




*Obr 6-2Graf reprezentující naměřených údajů programu používajícího GA*

uzly superpočítače Anselm.

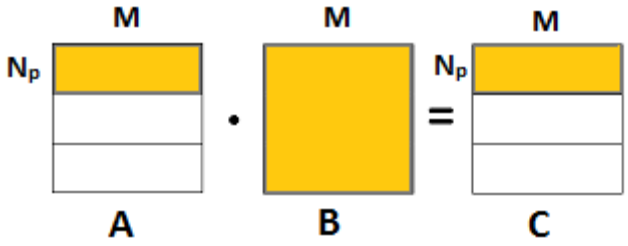
V případě řešení úlohy pomocí GA je situace při větším počtu procesů znatelně horší. Převážení času komunikace mezi procesy nad časem samotného výpočtu se znovu projeví při vyšším počtu procesů. Projeví se ovšem již při práci s menšími daty. Už v případě velikosti dat 32MB je řešení na více uzlech pomalejší, než řešení na více procesech jednoho uzlu.

### 6.3.4 Náročnost implementace

Co se týče rozdílu náročnosti implementace mezi MPI a GA, tak v obou případech je velice podobná. V GA je ovšem třeba si dávat pozor na aktuálnost dat a je tak potřeba procesy synchronizovat, jelikož mají přístup k datům v globálním poli nehledě na to, zda již jiný proces ukončil zapisování do daného globálního pole. Je tak někdy potřeba zaručit exkluzivní přístup k datům. Ovšem v této úloze je to potřeba pouze na začátku kdy řídící proces nahrává všechna data do celého globálního pole.

## 6.4 Násobení matic

V této úloze budeme řešit distribuované násobení matic. Budeme uvažovat pouze násobení čtvercových matic. Pro řešení této úlohy rozdělíme matici A podle mřížky o velikosti $M * N_p$, přičemž $M$ je šířka matice $A$ a $N_p$ je počet řádků matice $A$, které bude proces $p$ počítat. Vznikne tak rozdělení




*Obr 6-3Ukázka principu distribuovaného násobení matic*

matice $A$ podél horizontální osy na vodorovné pruhy. Pro zjednodušení výpočetního algoritmu bude mít každý proces k dispozici celou matici $B$.

Každý proces provede maticové násobení nad daným úsekem podle všeobecně známého vzorce pro násobení matic, který vypadá následovně. [6]

$$(A \cdot B)_{ij} = \sum_{r=1}^{n} a_{ir} b_{rj} = a_{i1} b_{1j} + a_{i2} b_{r2} \dots a_{in} b_{nj}.$$

Řídící proces generuje nové náhodné hodnoty matic při každé iteraci. Výsledná data poté shromáždí a po provedení všech iterací zapíše výslednou matici C do souboru pomocí knihovny HDF5. Matici zapisujeme pouze po provedení všech iterací, aby samotný zápis na disk zbytečně nezdržoval běh celého programu.

Měřit budeme časové úseky komunikace, samotného násobení matic, každé iterace a celkový čas běhu programu. Kromě celkového času běhu programu se z provedených měření vypočítá průměr vzhledem k počtu iterací. Pro porovnání rozdílu rychlostí jednotlivých implementací budeme používat celkový čas běhu programu.

### 6.4.1 MPI

Řídící proces vygeneruje dvě matice s náhodnými čísly. Obdobně jako v úloze počítání průměru vektorů určí, který proces bude počítat kolik řádků a tuto informaci rozešle, aby měl každý proces tyto informace k dispozici. Následně pomocí funkce `MPI_Bcast()` odešle všem procesů matici $B$. Dalším krokem je odeslání vybraných částí matice $A$ ostatním procesům, podle dříve určených počtů řádků. K tomu je použita funkce `MPI_Scatterv()` kterou jsme si popsali dříve.

Po přijetí potřebných dat začíná každý proces s vlastním násobením úseku matice, který mu byl přidělen. Jakmile je výpočet dokončen, jsou dílčí výsledky odeslány zpět řídícímu procesu. Tento přenos je zprostředkován funkcí `MPI_Gatherv()`, která je opakem funkce `MPI_Scatterv()` a její chování bylo taktéž popsáno dříve. Konečný výsledek je pak pomocí knihovny HD5F zapsán do souboru s příponou h5.

Při řešení tohoto problému přišlo vhod vytvoření vlastního datového typu pro MPI, který určuje celý řádek matice. Dosáhlo se tak mírného zjednodušení části kódu, která obstarává komunikaci mezi procesy.

### 6.4.2 GA

Při použití GA je řešení úlohy podobné. Opět však vzniká prostředník v komunikaci v podobě globálního pole.

Všechny procesy kolektivně vytvoří tři globální pole, které odpovídají příslušným maticím. Pro matici *A* tedy globální pole *g_a*, pro matici *B* *g_b* a pro matici *C* globální pole *g_C*. Řídící proces poté vygeneruje náhodné hodnoty pro matici *A* a *B* a tyto hodnoty nahraje do příslušných globálních polí. Veškerá komunikace s ostatními procesy v tomto případě probíhá na pozadí.

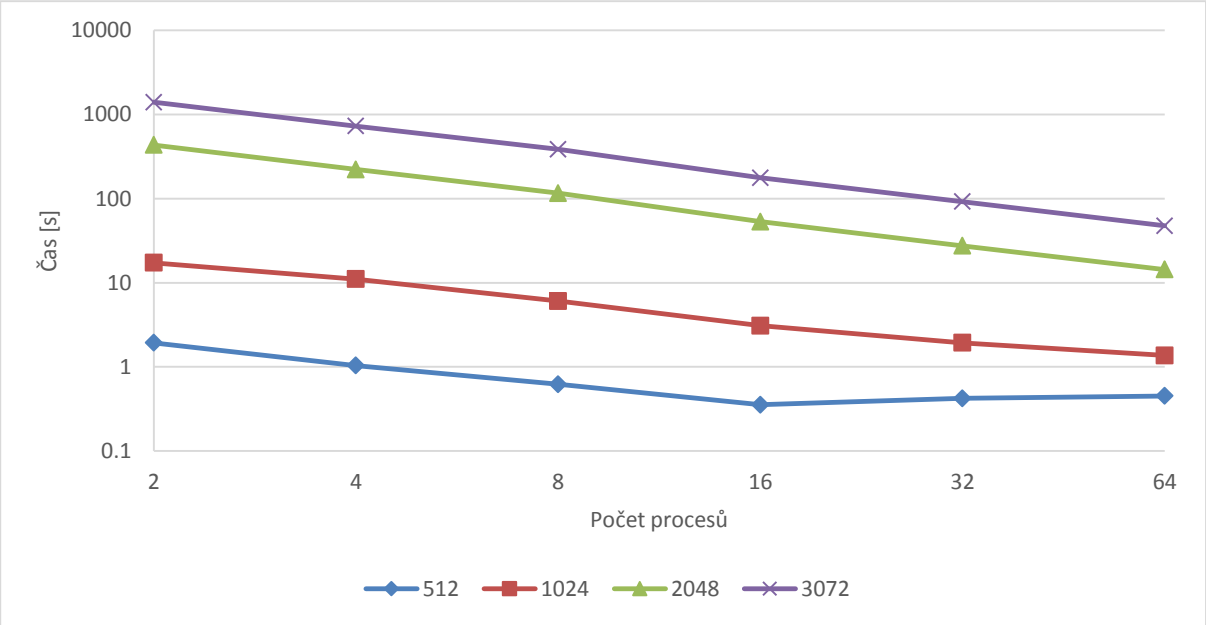
Dílčí procesy poté pomocí funkce `NGA_Distribution()` zjistí, která část globálního pole *g_a* jim patří a poté zavolají funkci `NGA_Access()`. Této funkci se jako parametr předává adresa ukazatele. Funkce hodnotu ukazatele změní tak, aby nyní ukazatel ukazoval na začátek úseku v paměti vlastněného daným procesem v globálním adresovém prostoru. V podstatě tak máme k dispozici potřebná data z globálního pole *g_a* a nemusíme žádná data kopírovat. Po dokončení práce s daty z globálního pole *g_a* je nutné zavolat funkci `NGA_Release()`.

Po dokončení násobení svého úseku matic nahrají dílčí procesy výsledek do globálního pole *g_c*. V tomto poli je tak nyní k dispozici výsledek násobení matic *A* a *B*.

### 6.4.3 Spuštění a výsledky

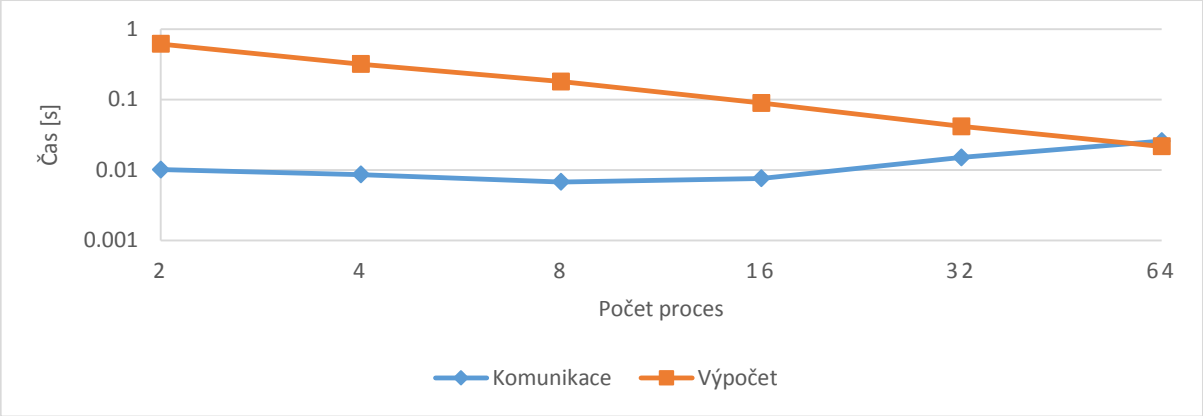
Úloha byla spuštěna na 2, 4, 8, 16, 32 a 64 procesorových jádrech s maticemi o velikost hrany 512, 1024, 2048 a 3072.

Na následujícím grafu můžeme vidět výsledek měření celkového času programu, který řešil úlohu pomocí MPI.




*Obr 6-4Graf reprezentující naměřené výsledky programu realizujícího násobení matic pomocí MPI*

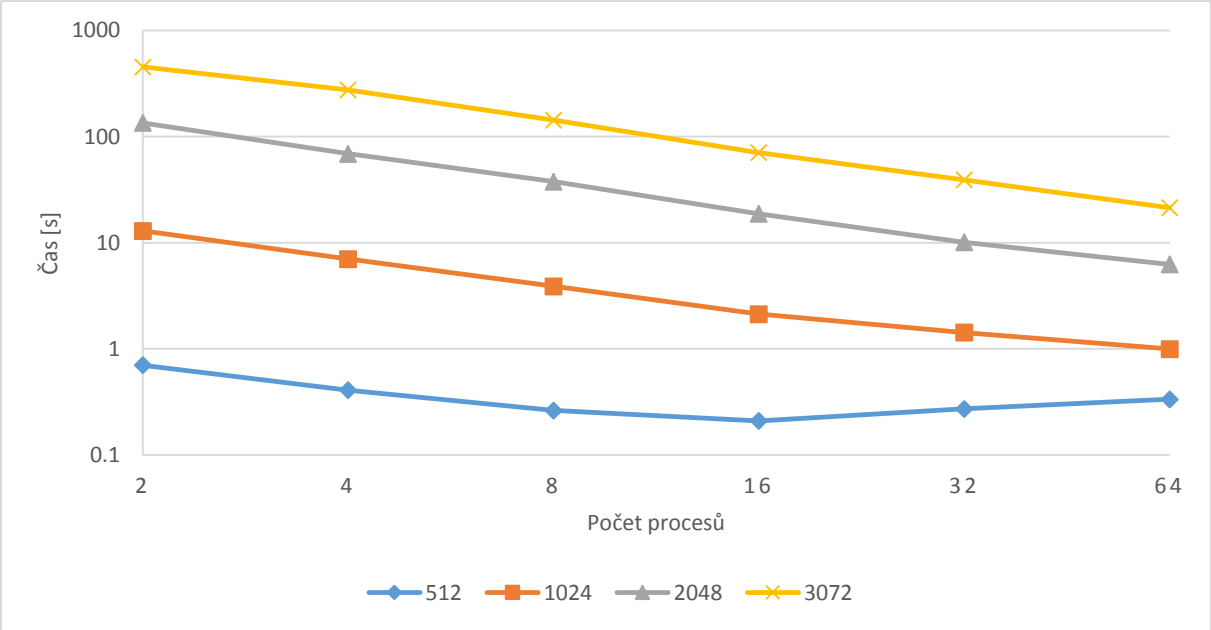
Zde můžeme vidět, že lepší škálovatelnost řešení se projevuje až v případě větších rozměrů matic. Při malém zadaném rozměru je toto řešení dokonce pomalejší při použití více výpočetních uzlů, než při využití všech procesorových jader uzlu jednoho. Je to způsobeno tím, že čas komunikace převyšuje čas výpočtu. Na následujícím grafu si můžeme tuto skutečnost pro daný případ ukázat.




*Obr 6-5Graf porovnávající čas komunikace a čas výpočtu*

Jak vidíme, tak je skutečně při použití 64 procesů čas komunikace větší než čas výpočtu. Rozdíl není sice tak patrný, ale stačí k tomu, aby nebylo možné plně využít výkonový potenciál všech použitých procesorových jader.

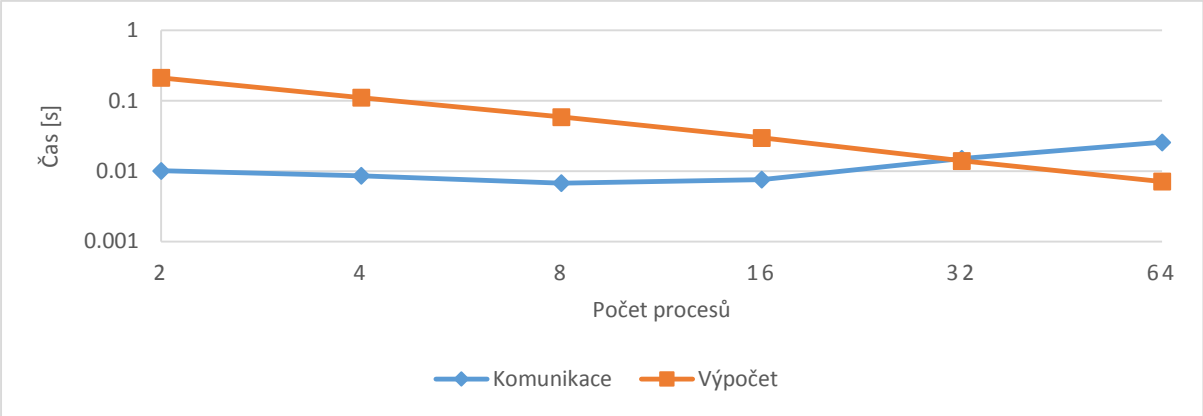
Dále se podíváme, jak si při řešení úlohy počínala knihovna GA.




*Obr 6-6Graf představující naměřené výsledky násobení matic pomocí GA*

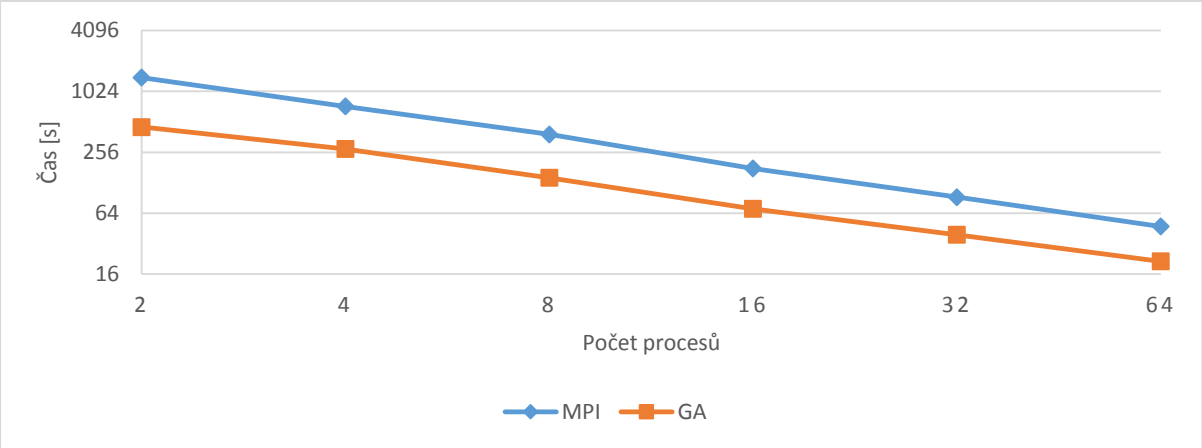
I v tomto případě můžeme vidět zpomalení výpočtu při malé velikosti matic a velkému počtu procesů. Na následujícím grafu si znovu porovnáme čas výpočtu a čas komunikace.

Opět se nám potvrdilo, že komunikace trvá delší dobu než výpočet při použití více procesů. Oproti MPI se však tento problém projevil již při použití 32 procesorových jader.




*Obr 6-8Graf porovnávající dobu komunikace a dobu výpočtu*

Co nás ale opravdu zajímá je porovnání časů běhu programů obou knihoven. Pro následující graf zvolíme největší velikost matice, tedy 3072*3072 elementů a porovnáme změřené časy.




*Obr 6-7Porovnání celkových výsledků mezi MPI a GA*

Překvapivě je řešení úlohy pomocí GA rychlejší než řešení pomocí MPI. Celkový průměrný procentuální rozdíl celkové doby běhu programu mezi GA a MPI je 51.5% což není málo.

### 6.4.4 Náročnost implementace

Při použití knihovny GA je znatelná výhoda globálního prostoru z důvodu jednoduššího přístupu k jednotlivým úsekům dat.

## 6.5 Šíření tepla na 2D desce

V této úloze budeme řešit numerickou simulaci problému šíření tepla. Budeme uvažovat 2D řez CPU chladičem věžovité konstrukce. Tento chladič má měděnou základnu a několik hliníkových žeber, které jsou připojeny k nosné konstrukci. K přenosu tepla dochází na styčné ploše chladiče. Teplo poté přechází do žeber chladiče a odtud do chladícího média, což je v tomto případě vzduch. Aby bylo možné simulovat odvádění tepla, je při výpočtu teploty vzduchu započítána tepelná ztráta. Tím můžeme simulovat proudění vzduchu okolo chladiče, jelikož vzduch proudí kolmo k řezu chladičem.

Průběh celé simulace je zapisován na disk pomocí paralelního zápisu, který nám umožní použitá knihovna HDF5 pro paralelní zápis.

Problém šíření tepla budeme řešit nejjednodušší numerickou metodou konečných diferencí v čase, tedy FDTD (*Finite Difference Time Domain*). Tato metoda je sice nejméně přesná a nejpomalejší, ale pro účely naší simulace je dostačující.

Podobně jako při řešení úlohy násobení matic si plochu řešeného problému rozdělíme na horizontální pruhy. Provedeme tedy 1D dekompozici řešeného prostoru. Pro paralelizaci výpočtu je tato dekompozice dostačující a stále si budeme moct ukázat překrývání komunikace s výpočtem.

Pro každý bod řešené plochy si definujeme tepelnou vodivost. Budou potřebné hodnoty pro měď, hliník a vzduch. Pokud je daný bod vzduch, tak přidáme koeficient, který určuje, jak rychle vzduch proudí okolo chladiče. V každém bodě si pak samozřejmě udržujeme i hodnotu aktuální teploty.

Jelikož provádíme simulaci v čase, je potřeba diskretizovat i časovou osu. Simulace bude prováděna v počtu kroků zadaného parametrem programu.

Pro výpočet bodu v čase $t + 1$ budeme potřebovat znát historický vývoj teploty v čase a aktuální teplotu v okolí aktuálního zkoumaného bodu. S těmito údaji může spočítat gradient teploty v prostoru a provést dopřednou integraci v čase. Pro výpočet gradientu zvolíme 4-okolí bodu a vypočítáme pomocí metody FDTD 2. řádu v prostoru a 1. řádu v čase. Tuto metodu lze popsat pomocí následující rovnice:

$$T_{t+1}[i][j] = \frac{T_t[i][j] + T_t[i+1][j] + T_t[i-1][j] + T_t[i][j+1] + T_t[i][j-1]}{5}$$

V podstatě teplotu dobu v čase $t + 1$ vypočítáme tak, že vypočítáme průměr teploty aktuálního bodu a jeho okolních sousedů v čase $t$. Abychom však mohli simulovat šíření tepla v heterogenním prostředím je potřeba zavést koeficient difuze. Musíme tedy každý člen vynásobit normalizovanou hodnotou tepelné vodivosti v daném bodě.

Pro simulaci proudění vzduchu okolo chladiče je potřeba každý bod, kterého médium je vzduch přepočítat pomocí následujícího vztahu.

$$T_{t+1}[i][j] = (\alpha * T_0) + (1 - \alpha * T_{t+1}[i][j])$$

kde $\alpha$ je relativní rychlost proudění vzduchu okolo chladiče vzhledem k délce časového kroku a $T_0$ je teplota nového vzduchu.

Samotnou simulaci pak provedeme procházením jednotlivých bodů na desce a aktualizováním jejich teploty. Při výpočtu je potřeba si uchovávat jak pole s teplotami v čase $t$, tak pole s teplotami v čase $t + 1$, aby nedocházelo k překrývání nových a starých hodnot.

Při řešení této úlohy bude potřeba mít řídící proces pouze v případě použití MPI, jelikož je potřeba vypočítat a rozeslat informace o rozdělení zadané desky. Data potřebná pro běh simulace si totiž každý proces načte ze vstupního h5 souboru pomocí paralelního čtení, které nám poskytne knihovna HDF5. Není tak již třeba načítat data pouze sekvenčně a následně je rozesílat. Při zápisu jednotlivých kroků simulace rovněž využijeme paralelní zápis do souboru. Tím odpadá nutnost shromažďovat výsledky jednotlivých kroků pro celou desku na jednom procesu. Urychlíme tím tak celkový běh simulace. Samotný zápis na soubor se neprovádí v každém kroku simulace, ale podle nastaveného parametru pro hustotu zápisu. Tento parametr zajistí, aby zápis na disk zbytečně nebrzdil průběh simulace a aby výsledný soubor s jejím průběhem nebyl moc velký. Do výsledného souboru pak budeme zapisovat data, každých $n$ kroků simulace. Pokud je tedy například počet kroků simulace $10000$ a zadaná hodnota parametru je $100$, tak do výsledného souboru zapíšeme celkem $100$ kroků simulace.

Pro vygenerování vstupního souboru simulace je použit malý program, který soubor vyplní potřebnými parametry pro jednotlivá média a počátečními teplotními hodnotami. Program desku generuje podle masky rozložení jednotlivých médií a tuto masku škáluje podle zadaného parametru velikosti desky. Dále do souboru uloží parametry simulace, jako je počáteční teplota, teplota procesoru, ze které se následně odvíjí šíření tepla, velikost desky a masku médií.

Při řešení tohoto problému využijeme překrývání komunikace se sousedními procesy a výpočtu vnitřní části desky. Dosáhneme tak dalšího zrychlení samotné simulace.

Následně si v bodech popíšeme průběh samotné simulace, který je společný pro řešení pomocí MPI a pro řešení pomocí GA

1) Načteme si jednotlivé parametry pro běh simulace. Těmito parametry jsou:
    - název vstupního souboru, ze kterého budeme načítat data
    - název výstupního souboru, do kterého budeme ukládat průběh simulace
    - počet iterací výpočtu
    - intenzitu zápisu do výstupního souboru
    - relativní rychlost proudění vzduchu
2) Po načtení parametrů programu můžeme přejít k načítání hodnot vstupního souboru. Načítání těchto hodnot provádí každý proces a načítá pouze hodnoty, které jsou v úseku, který bude skutečně počítat. Vybere si tedy tento úsek ve vstupním souboru a načte hodnoty popsané výše.
3) Vytvoříme si pole `old_temp` a `new_temp` do kterých načteme počáteční teplotní hodnoty bodů.
4) Následně vytvoříme výstupní soubor pro paralelní zápis a zapíšeme počáteční stav simulace v čase $T_0$.
5) Dalším krokem je získání parametrů materiálů od sousedního procesu.
6) Poté přecházíme do výpočetní fáze, která se provádí v cyklu. V každé iteraci se poté provádí tyto kroky.
    - Provedeme neblokující komunikaci se sousedními procesy, při které od nich získáme potřebné hraniční hodnoty.
    - Normalizujeme hodnoty doménových parametrů, aby byly v intervalu $\langle 0; 1 \rangle$. Tím opět dosáhneme zrychlení běhu simulace. Tentokrát ovšem způsobem, že pro stejný výsledek budeme potřebovat menší počet iterací.
    - Pokračujeme provedením výpočtu jednoho kroku simulace pomocí hodnot z okolí a výsledek uložíme do pole `new_temp`.
    - Pokud je aktuálním bodem vzduch, tak provedeme snížení jeho teploty podle zadaného parametru proudění vzduchu.
    - Po provedení tohoto výpočtu počkáme na dokončení neblokující komunikace započaté na začátku výpočetního cyklu.
    - Poté můžeme vypočítat krok simulace pro hraniční body.
    - Nyní můžeme vypočítané hodnoty v poli `new_temp` zaměnit s hodnotami v poli `old_temp`. V poli `old_temp` se tak nyní nachází vypočtený krok simulace. Tato záměna je provedena prohozením ukazatelů jednotlivých polí. Vyhneme se tak zdlouhavému kopírování polí.
    - Pokud je krok simulace dělitelný parametrem intenzity zápisu na disk, tak provedeme paralelní zápis výsledků daného kroku.

Při řešení tohoto problému již komunikace aktivně vstupuje do výpočtu. Při každé iteraci je nutné si vyměňovat data se sousedními procesy. Nestačí pouze potřebná data rozeslat před začátkem samotného výpočtu, jako tomu bylo u předchozích úloh. Zde se tedy ukáže, jestli režie komunikace v GA bude nějakým způsobem zpomalovat průběh celého výpočtu.

Rozdíl v řešení tohoto problému mezi GA a MPI, je opět ve způsobu komunikace a v manipulaci s daty. Obě knihovny poskytují možnost neblokující komunikace a neměl by tedy být problém popsané řešení problému implementovat. Pojďme se tedy podívat na jednotlivé rozdíly v implementaci.

### 6.5.1 MPI

Jako hlavní funkce pro komunikaci se sousedními procesy budeme využívat funkce `MPI_Isend()` a `MPI_Irecv()`. Tyto funkce bylo nutné použít i pro počáteční rozeslání dat doménových parametrů šíření tepla mezi sousedními procesy. Kdybychom použili obyčejné funkce `MPI_Send()` a `MPI_Recv()`, tak by došlo k zablokování procesů a nebylo by možné pokračovat v provádění programu.

Pro lepší přehlednost a usnadnění práce se zasílanými daty si vytvoříme nový MPI datový typ `row_type`, který bude představovat jeden řádek v desce s jednotlivými body. Tento nový datový typ vychází z datového typu `MPI_FLOAT`, počet obsažených hodnot je roven velikosti hrany desky ze vstupního souboru a je zároveň typem s kontinuálním uložením dat. Data jsou tedy v paměti uložena za sebou bez mezer. Nový typ je nutné systému potvrdit zavoláním funkce `MPI_Type_commit()`. Jelikož vstupní desku rozdělujeme mezi jednotlivé procesy pouze po řádcích, tak není nutné vytvářet datový typ reprezentující její sloupec.

### 6.5.2 GA

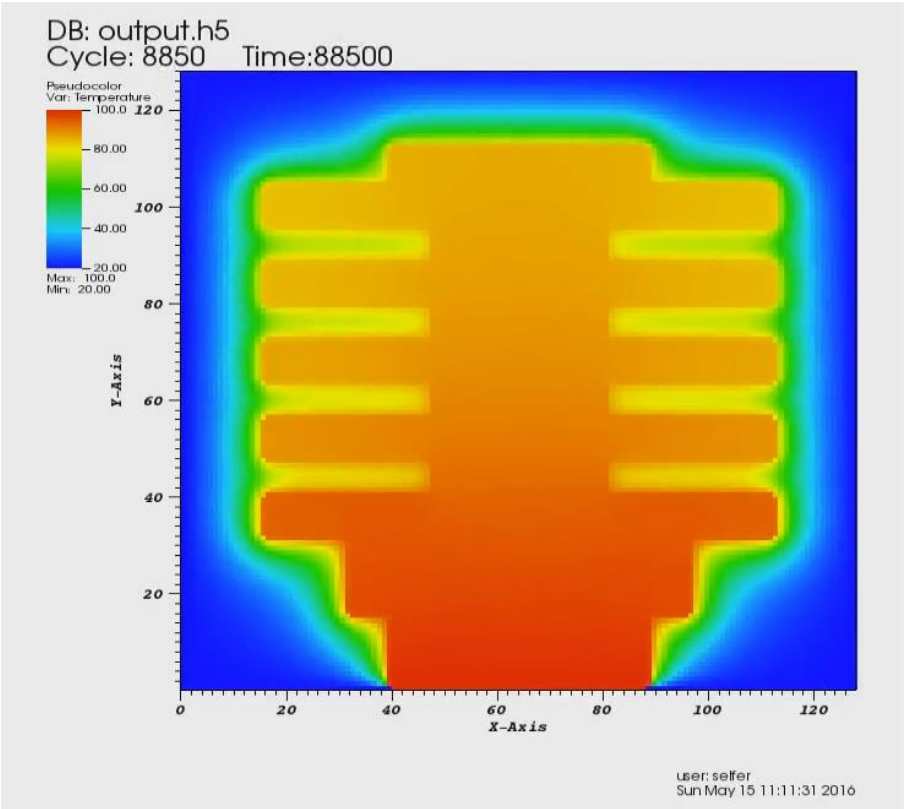
Při řešení tohoto problému pomocí GA budeme využívat dvě globální pole. Jedno pole použijeme pro data doménových parametrů a druhé pole pro samotné výsledky jednotlivých kroků simulace, do kterého budeme postupně tyto výsledky přeukládat.

Zmíněná pole vytvoříme při načítání parametrů simulace a máme tedy pak možnost získat informace o tom, kolik bude který proces počítat řádků desky. Rozdělení mezi procesy zajišťuje GA automaticky při vytváření globálního pole. Globální pole je rozděleno rovnoměrně a při správném nastavení bude rozděleno na vodorovné úseky. Pro získání informací o rozdělení mezi jednotlivými procesy však nemusíme provádět žádný výpočet, ale stačí nám zavolání funkce `NGA_Distribution()`, která nám vrátí horní a dolní hranice v každé dimenzi úseku vlastněného volajícím procesem. Počet řádků, pro které bude daný proces výpočet provádět pak lze získat velice jednoduše odečtením hodnoty dolní hranice od hodnoty horní hranice vlastněného úseku.

Pro neblokující komunikaci pak budeme používat funkce `NGA_NbGet()` a `NGA_NbPut()`. Těmto funkcím předáme jako jeden z parametrů odkaz na hodnotu proměnné, do které uloží požadavek, podle kterého můžeme později tuto komunikaci identifikovat při volání blokující funkce, která běh programu pozastaví, dokud nebudou všechna data přijata, potažmo odeslána.

### 6.5.3 Spuštění a výsledky

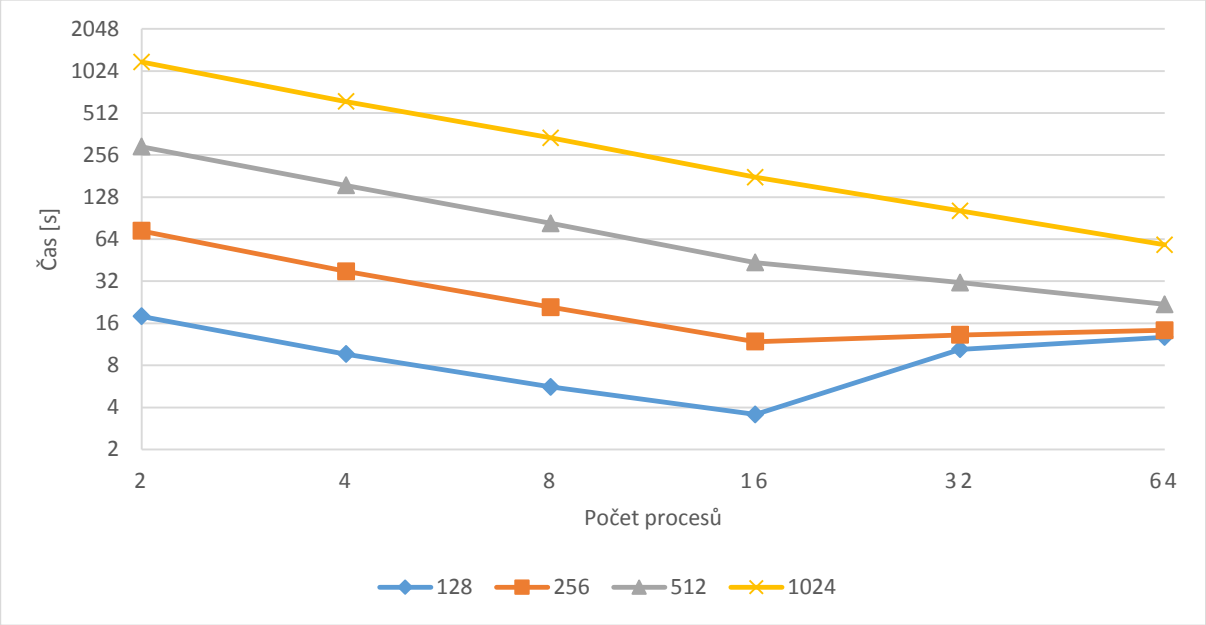
Simulace byla spuštěna na 2, 4, 8, 16, 32 a 64 procesorových jádrech a jako vstupní soubory jí byly předávány soubory s deskou o velikosti 128, 256, 512 a 1024. Počet kroků simulace pak byl při každém spuštění nastaven na 100000, hodnota intenzity zápisu na disk na 500 a relativní rychlost



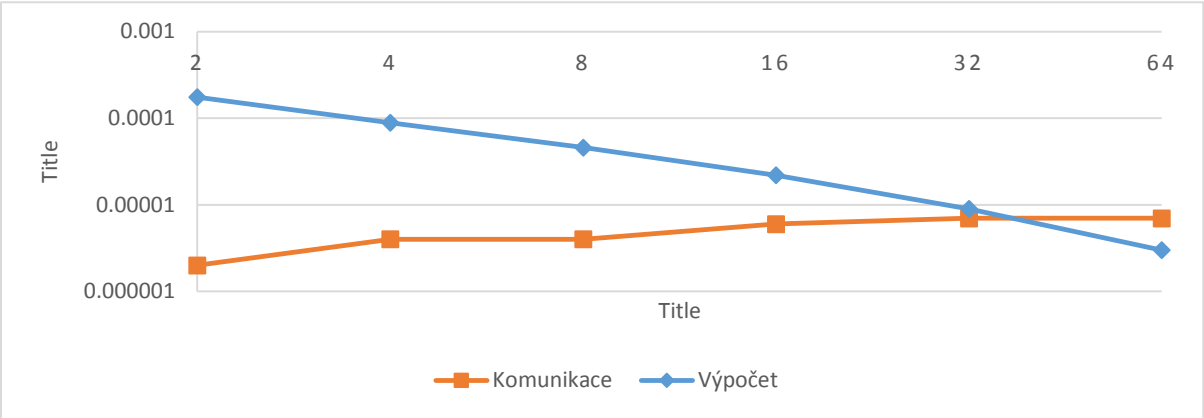

*Obr 6-9Ukázka kroku simulace ilustrovaného pomocí nástroje VisIt*

proudění vzduchu na 0.001, tedy 0.1% aby bylo možné vidět rychlejší šíření tepla. Každý ze vstupních souborů byl vygenerován s počáteční teplotou 20°C a teplotou procesoru nastavenou na 100°C.

Stejně jako v předchozích úlohách si nyní ukážeme časový průběh simulací. Nejdříve si ukážeme výsledek řešení používající MPI.




*Obr 6-10 Graf reprezentující naměřené výsledky programu realizujícího simulaci šíření tepla pomocí MPI*



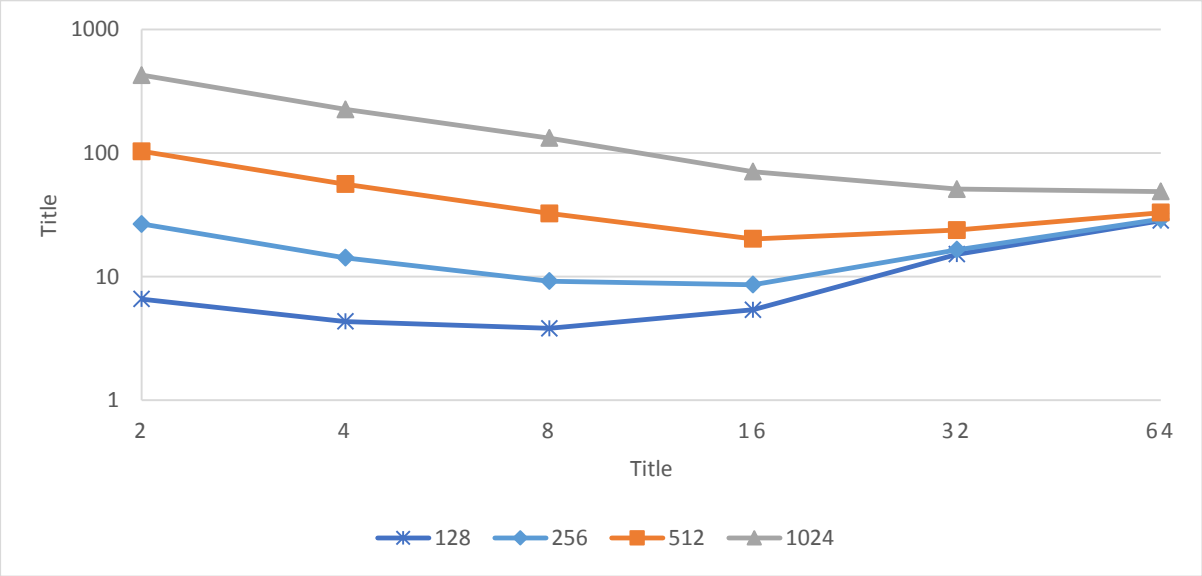

*Obr 6-11Porovnání časů komunikace a časů výpočtu - MPI*

Jak jsme mohli předpokládat, tak stejně jako v úloze násobení matic je při malé velikosti vstupních dat a velkém počtu řešících procesů čas komunikace větší než čas samotného výpočtu. Tento fakt zamezuje lepšímu škálování výkonu tohoto řešení při výpočtu na malých vstupních datech.

Stejně jako v předchozí úloze si můžeme ukázat porovnání časů komunikace s časy provádění výpočtů pro desku o velikosti hrany 128 bodů.

Výsledný graf už nás nepřekvapí. Lze jen podotknout, že čas komunikace se s přibývajícím počtem procesů nijak rapidně nezvyšuje. Můžeme tak vidět, že propojení jednotlivých výpočetních uzlů na Anselmu je velice kvalitní.

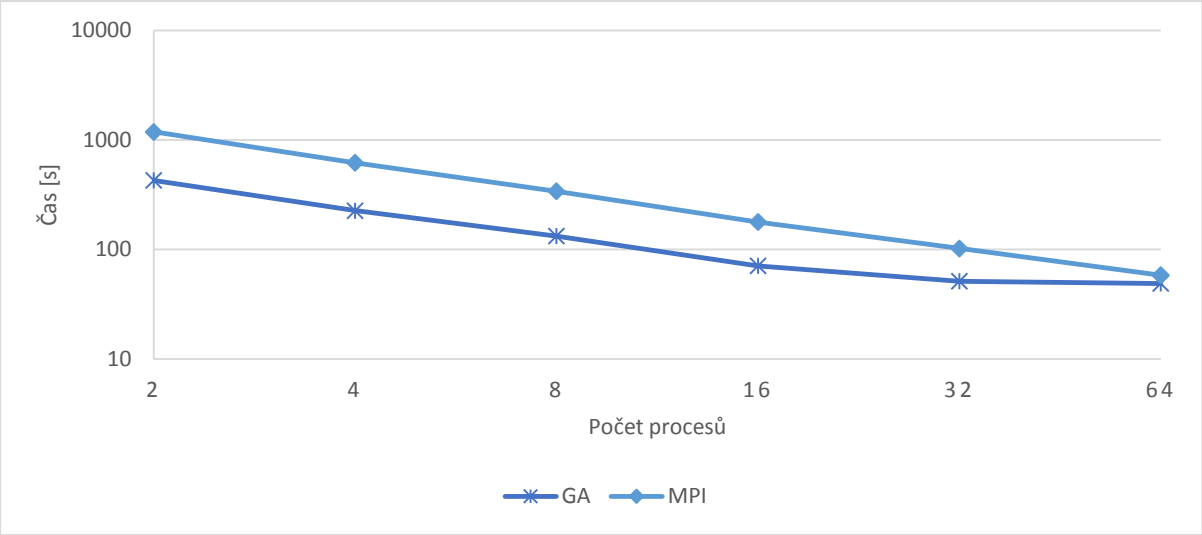
Nyní si můžeme zobrazit graf s výsledky celkového času provádění simulace, které byly naměřeny spuštěním výsledné implementace řešení zadaného problému.




*Obr 6-12 Graf reprezentující naměřené výsledky programu realizujícího simulaci šíření tepla pomocí GA*

Jak můžeme vidět, tak GA je na tom, co se týče škálovatelnosti řešení, o něco hůře než MPI. Až při velikosti hrany desky čítající 1024 bodů, nepřevýšil čas komunikace čas prováděného výpočtu. Projevuje se tak nyní vyšší režie komunikace, kterou GA provádí na pozadí.

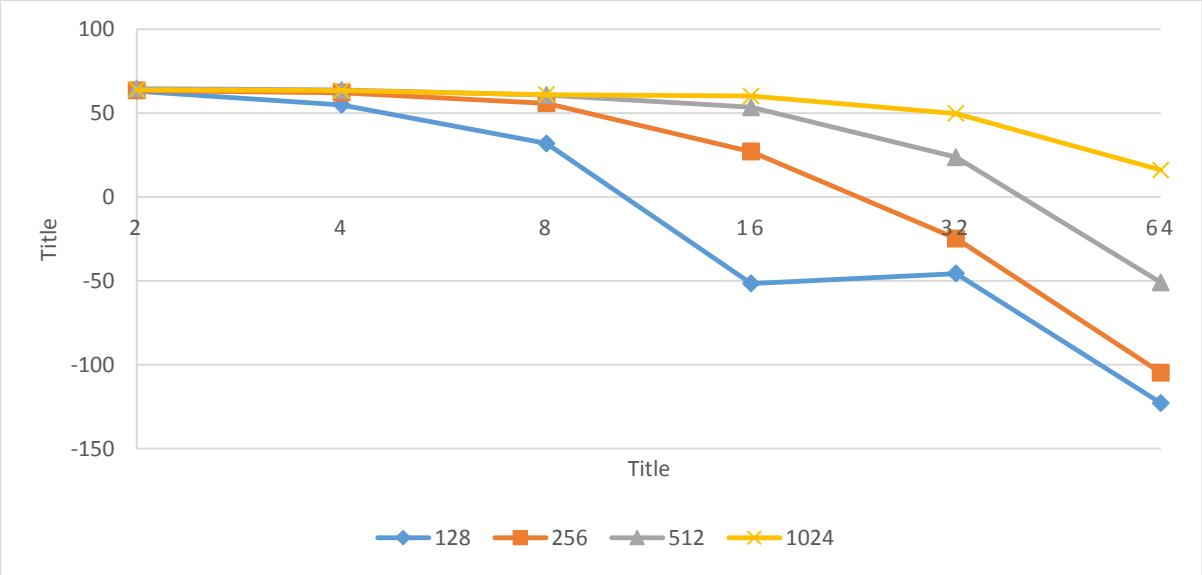
Na následujícím grafu si zobrazíme finální srovnání naměřených celkových časů pro největší vstupní soubor.




*Obr 6-13Porovnání naměřených časů GA a MPI při řešení simulace šíření tepla na desce o velikosti hrany 1024 bodů*

Jak můžeme vidět, tak při stoupajícím počtu procesů, začíná GA trošku pokulhávat. Dá se předpokládat, že při zvolení většího počtu procesů, než 64 by řešení využívající GA bylo pomalejší než řešení využívající MPI.

A nakonec se můžeme podívat na procentuální rozdíl mezi časy naměřenými v jednotlivých případech spuštění simulace.




*Obr 6-14Procentuální rozdíl mezi celkovými časy MPI a GA*

V některých případech se ukázalo být řešení pomocí GA dokonce o více než 100% procent pomalejší, než řešení pomocí MPI. Je tedy patrné, že pro dosažení lepší škálovatelnosti by bylo lepší využít řešení používající MPI. Nicméně s naměřenými výsledky je GA v průměru o 22.5% rychlejší.

### 6.5.4 Náročnost implementace

Opět je náročnost implementace v obou případech velice podobná. Avšak jednodušší bylo získávat data od sousedních procesů přes globální pole. Časová náročnost implementace byla takřka totožná, jakmile byly všechny základní funkce obou knihoven zažité.

# 7 Závěr

V této práci jsme si představili základy používání standartu MPI pro zasílání zpráv mezi procesy. Také jsme se seznámili s možností využití globálního adresového prostoru, který nám poskytuje knihovna GA. Změřili jsme dobu trvání celkového běhu několika programů řešících různé úlohy. Z našich výsledků je patrné, že knihovna GA má potenciál v případě výpočtů na velkých datech. Je však nutné si dát pozor na komunikační režii této knihovny. Při větším počtu procesů se totiž začne projevovat režie zajišťující komunikaci na pozadí, která může značně zpomalit běh celého programu.

Jako další postup by bylo vhodné vyzkoušet implementaci řešení reálného problému pomocí knihovny GA, zkusit využít možnost více rozměrných globálních polí a použití pokročilejších funkcí, jako jsou cyklické závislosti dat nebo vytváření takzvaných polí s takzvanými „ghost“ elementy, které přesahují do části pole patřící sousednímu procesu.

# Použitá literatura

[1] Pacheco, P. S.: Parallel Programming with MPI. Morgan Kaufman Publ., 1997, ISBN 1-55860-339-5

[2] MPI Tutorial. Mpitutorial.com [online]. [cit. 2016-05-10]. Dostupné z: http://mpitutorial.com/tutorials/

[3] MPI Data Types [online prezentace]. Research Computing University of Colorado, [cit. 2016-05-16]. Dostupný z WWW: https://www.rc.colorado.edu/sites/default/files/Datatypes.pdf

[4] PALMER, Bruce; KRISHNAMOORTY, Sriram; Daniel Chavarria, Abhinav Vishnu, Jeff Daily. Parallel Programming Using the Global Arrays Toolkit [online prezentace]. Pacific Northwest National Laboratory, [cit. 2016-05-12]. Dostupný z WWW: http://faculty.washington.edu/rjl/misc/GA_UW_11_v1.pdf

[5] NIEPLOCHA, Jarek, Bruce PALMER, Vinod TIPPARAJDU, Manojkumar KRISHNAN, Harold TREASE a Edo APRA. Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit. International Journal of High Performance Computing Applications [online]. 2006, 20(2), 203-231 [cit. 2016-05-12]. Dostupné z: http://hpc.pnl.gov/globalarrays/papers/ga_acts.pdf

[6] Wikipedie: Otevřená encyklopedie: Násobení matic [online]. c2016 [citováno 18. 05. 2016]. Dostupný z WWW: <https://cs.wikipedia.org/w/index.php?title=N%C3%A1soben%C3%AD_matic&oldid=13326952>