

# IB109 Návrh a implementace paralelních systémů

## Programování aplikací pro prostředí s distribuovanou pamětí

Jiří Barnat

## Principy programování s předáváním zpráv

## **Paradigma – předávání zpráv**

- Nesdílený adresový prostor
- Explicitní paralelismus

## **Pozorování**

- Data explicitně dělena a umístěna do jednotlivých lokálních adresových prostorů.
- Datová lokalita je klíčová vlastnost pro výkon.
- Komunikace vyžaduje aktivní účast komunikujících stran.
- Existují efektivně implementované podpůrné knihovny.
- Programátor je zodpovědný za paralelizaci algoritmu.

## Asynchronní paradigma

- Výpočet začne ve stejný okamžik (synchronně), ale ...
- Probíhá asynchronně (různá vlákna různou rychlostí).
- Možnost synchronizace v jednotlivých bodech výpočtu.
- Neplatí “Trojúhelníková nerovnost” v komunikaci.

## Další vlastnosti

- Vykazuje nedeterministické chování.
- Těžší prokazování korektnosti.
- Možnost provádění zcela odlišného kódu na jednotlivých procesních jednotkách.
- Typicky však “Single program multiple data”.
- Každá procesní jednotka má jednoznačnou identifikaci.

## Send a Receive – Základní stavební kameny

```
send(void *sendbuf, int nelems, int dest)
```

```
receive(void *recvbuf, int nelemns, int src)
```

- `sendbuf` – ukazatel na bafr (blok paměti), kde jsou umístěna data připravená k odeslání.
- `recvbuf` – ukazatel na bafr (blok paměti), kam budou umístěna přijatá data.
- `nelems` – počet datových jednotek, které budou poslány, či přijaty (délka zprávy).
- `dest` – adresát odesílané zprávy, tj. *ID* toho, komu je zpráva určena.
- `src` – odesílatel zprávy, tj. *ID* toho, kdo zprávu poslal, nebo toho, od koho chci zprávu přijmout.

## Příklad

- Paralelní systém s 2 procesy

1	P0	P1
2		
3	a = 100;	receive (&a, 1, 0);
4	send(&a, 1, 1);	printf("%d\n", a);
5	a = 0;	

## Výstup a efektivita

- Co by mělo být na výstupu procesu P1?
- Asynchronní sémantika send() a receive().
- Nutné z důvodu zachování výkonu aplikace.
- Co může být na výstupu procesu P1?

## Blokující operace

- Operace `send()` ukončena až tehdy, je-li to bezpečné vzhledem k sémantice, tj. že příjemce obdrží to, co bylo obsahem odesílaného bafu v okamžiku volání operace `send()`.
- Ukončení operace `send()` nevynucuje a negarantuje, že příjemce již zprávu přijal.
- Operace `receive()` skončí po přijetí dat a jejich umístění na správné místo v paměti.

## Nebafrované operace

- Operace `send()` skončí až po dokončení operace komunikace, tj. až přijímací proces zprávu přijme.
- V rámci operací `send()` a `receive()` před samotným přenosem dat probíhá synchronizace obou participujících stran (handshake).



## Prodlevy

- Způsobeno synchronizací před vlastní komunikací.
- Proces, který dosáhne bodu, kdy je připraven komunikovat čeká, až do stejného bodu dospěje i druhý proces.
- Volání `send()` a `receive()` ve stejný okamžik nelze garantovat na úrovni kódu.
- Nemá velký vliv, pokud dominuje čas komunikace.

## Uvážnutí (deadlock)

1	P0		P1
2			
3	<code>send(&amp;a, 1, 1);</code>		<code>send(&amp;a, 1, 0);</code>
4	<code>receive(&amp;b, 1, 1);</code>		<code>receive(&amp;b, 1, 0);</code>

## Bafr v bafrované komunikaci

- Extra paměť zdánlivě mimo adresový prostor procesů.
- Mezisklad zpráv při komunikaci.

## Bafrované komunikační operace

- Operace `send()` skončí v okamžiku, kdy odesílaná data kompletně překopírována do bafru.
- Případná modifikace posílaných dat po skončení operace `send()`, ale před započítím vlastní komunikace se neprojeví.
- Volání `send()`, vlastní komunikace a následné volání `receive()` na přijímací straně se nemusí časově překrývat.

## Režie související s bafrováním

- Eliminace prodlev za cenu režie bafrování.
- Ve vysoce synchroniích aplikacích může být horší než používání blokujících nebafrovaných operací.

## Velikost bafřů

- Pokud odesílatel generuje zprávy rychleji, než je příjemce schopen zprávy přijímat, velikost bafřů může neúměrně růst (problém producent-konzument).
- Pokud je velikost pro bafry omezená, může docházet (a to nedeterministicky) k situaci, kdy je předem daná velikost bafřů nedostatečná (buffer overflow).
- Případné samovolné blokování odesílatele do té doby, než odesílatel přijme nějaká data a bafry se uvolní, může vést k uváznutí, podobně jako v případě nebafrované blokující komunikace.

## Uvážnutí (i bez blokování odesílatele)

1	P0	P1
2		
3	<code>receive(&amp;b, 1, 1);</code>	<code>receive(&amp;b, 1, 0);</code>
4	<code>send(&amp;a, 1, 1);</code>	<code>send(&amp;a, 1, 0);</code>

## Asymetrický model blokující bafrované komunikace

- Neexistence odpovídajících prostředků pro bafrovanou komunikaci na úrovni komunikační vrstvy.
- Operace `send()` je blokující nebafrovaná.
- Přijímací proces je přerušen v běhu a zpráva je přijata do bafru, kde čeká, dokud přijímací proces nezavolá odpovídající operaci `receive()`.
- Dedikované vlákno pro obsluhu komunikace.

## Motivace

- Komunikace, která nezpůsobuje prodlevy.

## Neblokující operace

- Volání funkce může skončit dříve, než je to sémanticky bezpečné.
- Existuje funkce na zjištění stavu komunikující operace.
- Program nesmí modifikovat odesílaná data, dokud komunikace neskončí.
- Po dobu trvání komunikace program může vykonávat kód.
- Překrývání výpočtu a komunikace.

## HW Realizace

- DMA

# Přehled komunikačních módů

	Blokující	Neblokující
Bafrované	send skončí jakmile jsou data nakopírována do bafru	send skončí jakmile je inicializován DMA přenos
Nebafrované	send skončí až po skončení odpovídajícího receive	send skončí ihned, odešle se pouze požadavek na komunikaci
	Sémantiku operací nelze porušit	Korektní dokončení operací nutné zjišťovat opakovaným dotazováním se

## Message Passing Interface

- Standardizuje syntax a sémantiku komunikačních primitiv
- Přes 120 knihovných funkcí
- Rozhraní pro C, Fortran
- MPI verze 1.2
- MPI verze 2.0 (Paralelní I/O, C++ rozhraní, ...)
- Existují různé implementace standardu
  - mpich
  - LAM/MPI
  - Open MPI
- <http://www.mpi-forum.org/>



MPI funkce	Význam
MPI_Init	Inicializuje MPI
MPI_Finalize	Ukončuje MPI
MPI_Comm_size	Vrací počet participujících procesů
MPI_Comm_rank	Vrací identifikátor volajícího procesu
MPI_Send	Posílá zprávu
MPI_Recv	Přijímá zprávu

- Definice typů a konstant: `#include "mpi.h"`
- Návrátová hodnota při úspěšném volání fce: `MPI_SUCCESS`
- Kompilace: `mpicc`, `mpiCC`, `mpiC++`
- Spuštění programu: `mpirun`

## Inicializace

- Nastavuje MPI prostředí
- Musí být voláno na všech procesorech
- Musí být voláno jako první MPI funkce
- `argv` nesmí být modifikováno před voláním `MPI_Init`
- `MPI_Init(int *argc, char ***argv)`

## Finalizace

- Ukončuje MPI prostředí
- Musí být voláno na všech procesech
- Nesmí být následováno voláním MPI funkce
- Provádí různé úklidové práce
- `int MPI_Finalize()`

## Komunikační domény

- Sdružování participujících procesorů do skupin
- Skupiny se mohou překrývat

## Komunikátory

- Proměnné, které uchovávají komunikační domény
- Typ `MPI_Comm`
- Jsou argumentem všech komunikačních funkcí MPI
- Výchozí komunikátor: `MPI_COMM_WORLD`

## Zjišťování velikosti domény a identifikátoru v rámci domény

- `int MPI_Comm_size(MPI_Comm comm, int *size)`
- `int MPI_Comm_rank(MPI_Comm comm, int *rank)`
- `rank` je identifikátor procesu v dané doméně
- `rank` číslo v intervalu `[0, size-1]`

# Hello World

```
1  #include "mpi.h"
2
3  main (int argc, char *argv[])
4  {
5      int npes, myrank;
6
7      MPI_Init (&argc, &argv);
8      MPI_Comm_size(MPI_COMM_WORLD, &npes);
9      MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
10
11     printf("Hello World!  (%d out of %d)",
12           myrank,npes);
13
14     MPI_Finalize();
15 }
```

```
int MPI_Send(void *buf, int count,  
             MPI_Datatype datatype, int dest,  
             int tag, MPI_Comm comm)
```

- Odešle data odkazovaná ukazatelem `buf`
- Na data se nahlíží jako na sekvenci instancí typu `datatype`
- Odešle se `count` po sobě jdoucích instancí
- `dest` je rank adresáta v komunikační doméně určené komunikátorem `comm`
- `tag`
  - Přiložená informace typu `int` v intervalu `[0, MPI_TAG_UB]`
  - Pro příjemce viditelná bez čtení obsahu zprávy
  - Typicky odlišuje typ zprávy

# Korespondence datových typů MPI a C

MPI datový typ	Odpovídající datový typ v C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	-
MPI_PACKED	-

```
int MPI_Recv(void *buf, int count,  
             MPI_Datatype datatype, int source,  
             int tag, MPI_Comm comm,  
             MPI_Status *status)
```

- Přijme zprávu od odesílatele s rankem source v komunikační doméně comm s tagem tag
- source může být MPI\_ANY\_SOURCE
- tag může být MPI\_ANY\_TAG
- Zpráva uložena na adrese určené ukazatelem buf
- Velikost bafru je určena hodnotami datatype a count
- Pokud je bafr malý, návratová hodnota bude MPI\_ERR\_TRUNCATE

MPI\_Status

```
typedef struct MPI_Status {  
    int MPI_SOURCE;  
    int MPI_TAG;  
    int MPI_ERROR;  
};
```

- Vhodné zejména v případě příjmu v režimu MPI\_ANY\_SOURCE nebo MPI\_ANY\_TAG
- MPI\_Status drží i další informace, například skutečný počet přijatých dat (délka zprávy)

```
int MPI_Get_count(MPI_Status *status,  
                 MPI_Datatype datatype,  
                 int *count)
```



## MPI\_Recv

- Volání skončí až jsou data umístěna v bafru
- Blokující receive operace

## MPI\_Send

- MPI standard připouští 2 různé sémantiky
  - 1) Volání skončí až po dokončení odpovídající receive operace
  - 2) Volání skončí jakmile jsou posílaná data zkopírována do bafru
- Změna odesílaných dat je vždy sémanticky bezpečná
- Blokující send operace

## Možné důvody uvážnutí

- Jiné pořadí zpráv při odesílání a přijímání
- Cyklické posílání a přijímání zpráv (večeřící filozofové)

## MPI\_Sendrecv

- Operace pro současné přijímání i odesílání zpráv
- Nenastává cyklické uváznutí
- Bafry pro odesílaná a přijímaná data musí být různé

```
int MPI_Sendrecv (  
    void *sendbuf, int sendcount,  
    MPI_Datatype senddatatype, int dest, int sendtag,  
    void *recvbuf, int recvcount,  
    MPI_Datatype recvdatatype,  
    int source, int recvtag, MPI_Comm comm,  
    MPI_Status *status)
```

## **Problémy** MPI\_Sendrecv

- Bafry pro odesílaná a přijímaná data zabírají 2x tolik místa
- Složitá manipulace s daty díky 2 různým bafrům

## MPI\_Sendrecv\_replace

- Operace odešle data z bafru a na jejich místo nakopíruje přijatá data

```
int MPI_Sendrecv_replace (  
    void *buf, int count,  
    MPI_Datatype datatype, int dest, int sendtag,  
    int source, int recvtag, MPI_Comm comm,  
    MPI_Status *status)
```

## Neblokující komunikace

```
int MPI_Isend(void *buf, int count,  
             MPI_Datatype datatype, int dest,  
             int tag, MPI_Comm comm,  
             MPI_Request *request)
```

```
int MPI_Irecv(void *buf, int count,  
             MPI_Datatype datatype, int source,  
             int tag, MPI_Comm comm,  
             MPI_Request *request)
```

MPI\_Request

- Identifikátor neblokující komunikační operace
- Potřeba při dotazování se na dokončení operace

# Dokončení neblokujících komunikačních operací

```
int MPI_Test(MPI_Request *request, int *flag,  
             MPI_Status *status)
```

- Nulový flag znamená, že operace ještě probíhá
- Při prvním volání po dokončení operace se naplní status, zničí request a flag nastaví na *true*

```
int MPI_Wait(MPI_Request *request,  
             MPI_Status *status)
```

- Blokující čekání na dokončení operace
- Po dokončení je request zničen a status naplněn

```
int MPI_Request_free(MPI_Request *request)
```

- Explicitní zničení objektu request
- Nemá vliv na probíhající operaci

## Párování

- Neblokující a blokující send a receive se mohou libovolně kombinovat

## Uvážnutí

- Neblokující operace řeší většinu problémů s uvážnutím
- Neblokující operace mají vyšší paměťové nároky
- Později zahájená neblokující operace může skončit dříve

## Kolektivní komunikace



- Množina participujících procesů je určena komunikační doménou (MPI\_Comm).
- **Všechny procesy v doméně musí volat odpovídající MPI funkci.**
- Obecně se kolektivní operace nechovají jako bariéry, tj. jeden proces může dokončit volání funkce dříve, než jiný proces vůbec dosáhne místa volání kolektivní operace.
- Forma virtuální synchronizace.
- Nepoužívají tagy (všichni vědí jaká operace se provádí).
- Pokud je nutné specifikovat zdrojový, či cílový proces, musí tak učinit všechny participující procesy a jejich volba cílového, či zdrojového procesu musí být shodná.
- MPI podporuje dvě varianty kolektivních operací
  - posílají se stejně velká data (např. MPI\_Scatter)
  - posílají se různě velká data (např. MPI\_Scatterv)

```
int MPI_Barrier(MPI_Comm comm)
```

- Základní synchronizační primitivum.
- Volání funkce skončí pokud všechny participující procesy zavolají `MPI_Barrier`.
- Není nutné, aby volaná funkce byla "na stejném místě v programu".

Operace	Jméno MPI funkce
OTA vysílání	MPI_Bcast
ATO redukce	MPI_Reduce
ATA vysílání	MPI_Allgather
ATA redukce	MPI_Reduce_scatter
Kompletní redukce	MPI_Allreduce
Gather	MPI_Gather
Scatter	MPI_Scatter
ATA zosobněná komunikace	MPI_Alltoall

```
int MPI_Bcast(void *buf, int count,  
             MPI_Datatype datatype,  
             int source, MPI_Comm comm)
```

- Rozesílá data uložená v bafru buf procesu source ostatním procesům v doméně comm.
- Kromě buf musí být parametry funkce shodné ve všech participujících procesech.
- Parametr buf na ostatních procesech slouží pro identifikaci bafru pro příjem dat.

```
int MPI_Reduce(void *sendbuf, void *recvbuf,  
              int count, MPI_Datatype datatype,  
              MPI_Op op, int target,  
              MPI_Comm comm)
```

- Data ze sendbuf zkombinována operací op do recvbuf procesu target.
- Všichni participující musí poskytnout recvbuf i když výsledek uložen pouze na procesu target.
- Hodnoty count, datatype, op, target musí být shodné ve všech volajících procesech.
- Možnost definovat vlastní operace typu MPI\_Op.

Operace	Význam	Datové typy
MPI_MAX	Maximum	C integers and floating points
MPI_MIN	Minimum	C integers and floating points
MPI_SUM	Součet	C integers and floating points
MPI_PROD	Součin	C integers and floating points
MPI_BAND	Logické AND	C integers
MPI_BAND	Bitové AND	C integers and byte
MPI_LOR	Logické OR	C integers
MPI_BOR	Bitové OR	C integers and byte
MPI_LXOR	Logické XOR	C integers
MPI_BXOR	Bitové XOR	C integers and byte
MPI_MAXLOC	Maximum a minimální pozice s maximem	Datové dvojice
MPI_MINLOC	Minimum a minimální pozice s minimem	Datové dvojice

MPI datové páry	C datový typ
MPI_2INT	int, int
MPI_SHORT_INT	short, int
MPI_LONG_INT	long, int
MPI_LONG_DOUBLE_INT	long double, int
MPI_FLOAT_INT	float, int
MPI_DOUBLE_INT	double, int

```
int MPI_Allreduce(void *sendbuf, void *recvbuf,  
                 int count, MPI_Datatype datatype,  
                 MPI_Op op, MPI_Comm comm)
```

```
int MPI_Scan(void *sendbuf, void *recvbuf,  
            int count, MPI_Datatype datatype,  
            MPI_Op op, MPI_Comm comm)
```

- Provádí prefixovou redukci.
- Proces s rankem  $i$  má ve výsledku hodnotu vzniklou redukcí hodnot procesů s rankem 0 až  $i$  včetně.
- Jinak shodné s redukcí.



```
int MPI_Gather(void *sendbuf, int sendcount,
              MPI_Datatype senddatatype,
              void *recvbuf, int recvcount,
              MPI_Datatype recvdatatype,
              int target, MPI_Comm comm)
```

- Všichni posílají stejný typ dat.
- Cílový proces obdrží  $p$  bafrů seřazených dle ranku odesílatele.
- `recvbuf`, `recvcount`, `recvdatatype` platné pouze pro proces s rankem `target`.
- `recvcount` je počet odeslaných dat jedním procesem, nikoliv celkový počet přijímaných dat.

```
int MPI_Allgather(void *sendbuf, int sendcount,  
                 MPI_Datatype senddatatype,  
                 void *recvbuf, int recvcount,  
                 MPI_Datatype recvdatatype,  
                 MPI_Comm comm)
```

- Bez určení cílového procesu, výsledek obdrží všichni.
- `recvbuf`, `recvcount`, `recvdatatype` musí být platné pro všechny volající procesy.

```
int MPI_Gatherv(void *sendbuf, int sendcount,  
               MPI_Datatype senddatatype,  
               void *recvbuf, int *recvcounts,  
               int *displs,  
               MPI_Datatype recvdatatype,  
               int target, MPI_Comm comm)
```

- Odesílatelé mohou odesílat různě velká data (různé hodnoty sendcount).
- Pole recvcounts udává, kolik dat bylo přijato od jednotlivých procesů.
- Pole displs udává, kde v bafru recvbuf začínají data od jednotlivých procesů.

```
int MPI_Allgatherv(void *sendbuf, int sendcount,  
                  MPI_Datatype senddatatype,  
                  void *recvbuf, int *recvcounts,  
                  int *displs,  
                  MPI_Datatype recvdatatype,  
                  MPI_Comm comm)
```

```
int MPI_Scatter(void *sendbuf, int sendcount,  
               MPI_Datatype senddatatype,  
               void *recvbuf, int recvcount,  
               MPI_Datatype recvdatatype,  
               int source, MPI_Comm comm)
```

- Posílá různá data **stejné** velikosti všem procesům.

```
int MPI_Scatterv(void *sendbuf, int *sendcounts,  
                 int *displs,  
                 MPI_Datatype senddatatype,  
                 void *recvbuf, int *recvcounts,  
                 MPI_Datatype recvdatatype,  
                 int source, MPI_Comm comm)
```

- Posílá různá data **různé** velikosti všem procesům.

```
int MPI_Alltoall(void *sendbuf, int sendcount,  
                MPI_Datatype senddatatype,  
                void *recvbuf, int recvcount,  
                MPI_Datatype recvdatatype,  
                MPI_Comm comm)
```

- Posílá stejně velké části bafru `sendbuf` jednotlivým procesům v doméně `comm`.
- Proces  $i$  obdrží část velikosti `sendcount`, která začíná na pozici `sendcount * i`.
- Každý proces má v bafru `recvbuf` na pozici `recvcount * i` data velikosti `recvcount` od procesu  $i$ .

```
int MPI_Alltoallv(void *sendbuf, int *sendcounts,
                  int *sdispls,
                  MPI_Datatype senddatatype,
                  void *recvbuf, int *recvcounts,
                  int *rdispls,
                  MPI_Datatype recvdatatype,
                  MPI_Comm comm)
```

- Pole `sdispl` určuje, kde začínají v bafu `sendbuf` data určená jednotlivým procesům.
- Pole `sendcounts` určuje množství dat odesílaných jednotlivým procesům.
- Pole `rdispls` a `recvcounts` udávají stejné informace pro přijatá data.

## Skupiny, komunikátory a topologie



```
int MPI_Comm_split(MPI_Comm comm, int color, int key,  
                  MPI_Comm *newcomm)
```

- Kolektivní operace, musí být volána všemi.
- Parametr `color` určuje výslednou skupinu/doménu.
- Parametr `key` určuje rank ve výsledné skupině.
  - Při shodě `key` rozhoduje původní rank.

## Nevýhody "ručního" mapování

- Pravidla mapování určena v době kompilace programu.
- Nemusí odpovídat optimálnímu mapování.
- Nevhodné zejména v případech nehomogenního prostředí.

## Mapování přes MPI

- Mapování určeno za běhu programu.
- MPI knihovna má k dispozici (alespoň částečnou) informaci o síťovém prostředí (například počet použitých procesorů v jednotlivých participujících uzlech).
- Mapování navrženo s ohledem na minimalizaci ceny komunikace.

```
int MPI_Cart_create (  
    MPI_Comm comm_old, int ndims, int *dims,  
    int *periods, int reorder, MPI_Comm *comm_cart)
```

- Pokud je v původní doméně `comm_old` dostatečný počet procesorů, tak vytvoří novou doménu `comm_cart` s virtuální kartézskou topologií.
- Funkci musí zavolat všechny procesy z domény `comm_old`.
- Parametry kartézské topologie
  - `ndims` – počet dimenzí
  - `dims []` – pole rozměrů jednotlivých dimenzích
  - `periods []` – pole příznaků cyklické uzavřenosti dimenzí
- Příznak `reorder` značí, že ranky procesů se mají v rámci nové domény vhodně přeuspořádat.
- Nepoužité procesy označeny rankem `MPI_COMM_NULL`.

- Komunikační primitiva vyžadují rank adresáta.
- Překlad z koordinátů (`coords[]`) do ranku

```
int MPI_Cart_rank (MPI_Comm comm_cart,  
                  int *coords, int *rank)
```

- Překlad z ranku na koordináty
- `maxdims` je velikost vstupního pole `coords[]`

```
int MPI_Cart_coord(MPI_Comm comm_cart,  
                  int rank, int maxdims,  
                  int *coords)
```

```
int MPI_Cart_sub(MPI_Comm comm_cart, int *keep_dims,  
                MPI_Comm *comm_subcart)
```

- Pro dělení kartézských topologií na topologie s menší dimenzí.
- Pole příznaků `keep_dims` určuje, zda bude odpovídající dimenze zachována v novém dělení.

## Příklad

- Topologie o rozměrech  $2 \times 4 \times 7$ .
- Hodnotou `keep_dims = {true, false, true}`.
- Vzniknou 4 nové domény o rozměrech  $2 \times 7$ .

## Případová studie implementace verifikačního nástroje DiVinE

## DiVinE-Cluster

- Softwarový nástroj pro verifikaci protokolů (LTL MC).
- Problém detekce akceptujícího cyklu v grafu.
- Algoritmy paralelně prochází graf konečného automatu.
- Standardní průzkumová dekompozice.

## Algoritmus MAP

- Detekuje, zda existuje akceptující vrchol, který je svůj vlastní předchůdce.

## Algoritmus OWCTY

- Označuje vrcholy, které nejsou součástí akceptujícího cyklu
  - nemají přímé předchůdce (neleží na cyklu),
  - nemají akceptující předky (neleží na akceptujícím cyklu).

## Obecně

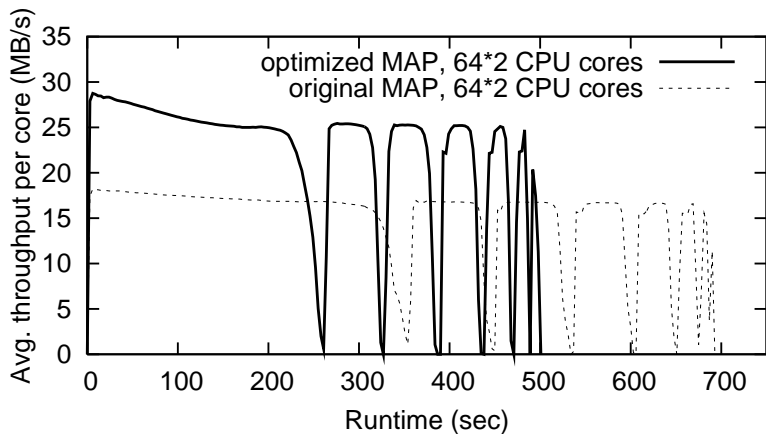
- Při testování na standardním Ethernetu, na malém počtu počítačů, některé nedokonalosti zůstaly neodhaleny.

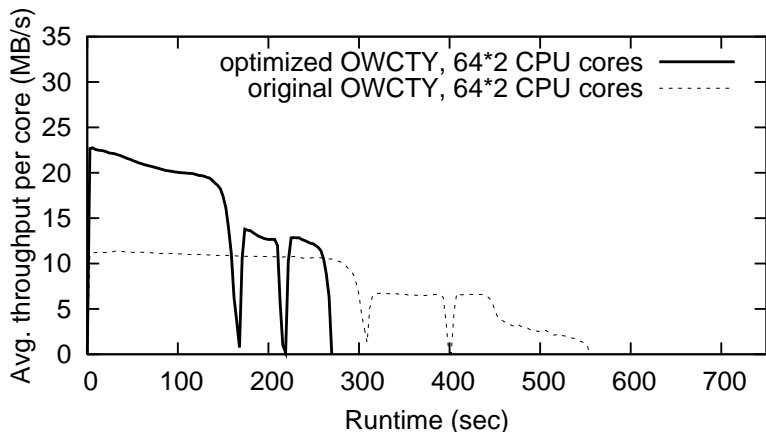
## Konkrétní problémy

- Po inicializaci některých počítačů, aktivita těchto počítačů zabránila inicializaci ostatních počítačů – obrovské prodlevy při startu výpočtu.
- Kombinace posílaných zpráv (bafrování na aplikační úrovni)
  - Příliš časté vylívání – komunikovaly se prázdné bafry.
  - Vylití všech bufferů najednou – náhlá velká zátěž sítě (contention).
  - Používání časových známek a vylívání na základě času – netriviální režie způsobená ověřováním zda uplynulo dané množství času.
- Příliš časté dotazování se na příchozí zprávy.



# Výsledky optimalizací





Nodes	Total cores	Runtime (s)		Efficiency	
		MAP	OWCTY	MAP	OWCTY
1	1	956.8	628.8	100%	100%
16	16	73.9	42.5	81%	92%
16	32	39.4	22.5	76%	87%
16	64	20.6	11.4	73%	86%
64	64	19.5	10.9	77%	90%
64	128	10.8	6.0	69%	82%
64	256	7.4	4.3	51%	57%

Table: Efficiency of MAP and OWCTY

## Zadání

- Napište MPI aplikaci, ve které si participující procesy mezi sebou zvolí jednoho velitele.
- Volba musí být realizována pomocí generování náhodných čísel a jejich výměny mezi jednotlivými MPI procesy.
- Při spuštění tato aplikace vypíše autorovo UČO.
- Kód spustitelný a přeložitelný na nymfe50.

## Odevzdání Odevzdání

- Termín do 11. 5. 2016 23:59.
- Odevzdávárna v ISu, zabaleno programem TAR a komprimováno GZIPem: *IB109\_02\_učo.tar.gz*
- Archiv obsahuje sbalený adresář IB109\_02\_učo.
- Povinně obsahuje Makefile.
- Provedení make uvnitř adresáře přeloží a spustí aplikaci.
- Nesprávné odevzdání jde na vrub studenta.