Otázka č.1: Ciele paralelného programovania paralelných výpoctov a vztah k programovaniu
systémov reálneho casu. Charakteristika a využitie paralelných architektúr. Vyváženost
a dodatocné náklady pri paralelných výpoctoch.

Paralelné programovanie – paralelné algoritmy pre rýchle výpočty a) pseudoparalelné (tlač na viacerých tlačiarňách naraz) b) paralelné (systém reálneho času, výmena pomocou správ)
- paralené programy v systémoch

Nevýhody PP – nie vždy je možné efektívne program paralelizovať, medzi efektívnymi par. problémami a druhom použitej paralelnej architektúry je silný vzťah, pred implementáciou paralelného problému je potrebné ho dekomponovať

Cieľ dekompozície – zabezpečiť , aby paralelný systém mal dostatočné zdroje, treba naplánovať záťaž na jednotlivých procesoch, aby sa minimalizovali náklady na pamäť, komunikáciu ...

Vyvažovanie – program je vyvážený dobre, ak práca je rozložená najefektívnejšie na jednotlivých procesoroch a každý proces trávi väčšinu času výpočtom, čas strávený na komunikáciu je minimálny
- vyváženosť môžeme riešiť staticky (t.j. pred spustením), alebo dynamicky (teda počas behu programu)

Paralelné architektúry
- single instruction (SISD,SIMD)
- multiple instruction (MISD,MIMD)

MIMD - DM (distributive memory)
- VM (visual shared memory)
- SM (shared memory)

SISD – single instruction , single data
SIMD – single instruction, multiple data – každý z velč. lok. procesorov má lokálnu pamäť , vykonáva tú istú operáciu v každom kroku
MIMD – menší počet procesorov, pričom každý procesor je schopný vykonávať samostatný program
- SM - malý počet procesorov, ktoré sú prepojené na globálnu pamäť
- DM – počítačové clustre – rýchlosť závisí od pripojenia počítačov na sieť

Otázka č. 2: Vlastnosti paralelných algoritmov a paralelných problémov, definícia inherentného, ohraniceného a neohraniceného paralelizmu, tried efektívne a optimálne paralelizovatelných problémov a téza paralelného výpoctu.

Paralelné problémy a algoritmy
Inherentný paralelizmus – ak je problém inherentne paralelný nemožno ho riešiť paralelným algoritmom
2 algoritmy tej istej funkcie sa môžu líšiť v skupine inherentného paralelizmu
Neohraničený paralelný algoritmus – je vyjadrený paralelným časom výpočtu bez ohľadu na zdroje počítačovej architektúry
Ohraničený paralelný algoritmus - je vyjadrený paralelným časom výpočtu vo vzťahu na zdroje počítačovej architektúry
Efektívne a optimálne paralelné algoritmy - nech k ϵ Z a n je veľkosť problému, potom efektívny paralelný algoritmus leží v poly logaritmickom čase O(log k n) pri počte procesov n^k . Ak problém nie je efektívne algoritm. – neoplatí sa ho paralelizovať
Optimálny paralelný algoritmus – súčin paralelného času a počtu procesorov je lineárny
- nezávisí na veľkosti problému pT = δ (n)

Súhrn nákladov pri paralelnom výpočte
- Náklady na komunikáciu a synchronizáciu
- Ťažko paralel. časti kódu
- Algoritmické náklady – alg. , ktorý je použitý – namiesto najrýchlejšieho sekvenčného algoritmu, môže obsahovať väčší počet operácií ako sekvenčný
- Softvérové náklady spojené s indexovaním , volaním procedúr, spracovaním cyklov
- Nevyváženosť záťaže

Téza paralelného výpočtu
F- funkcia nejakého problému veľkosti n
- hovorí , že trieda problémov , ktoré možno riešiť neohranič. paral. v čase $F(n)^{\delta(i)}$ je rovná triede problémov, ktoré možno riešiť sekvenčným výpočtom v pamäti veľkosti $F(n)^{\delta(i)}$
δ(i) – ľubovoľná konšt. funkcia

Otázka č.3: Dekompozícia paralelných problémov vo vztahu na jednotlivé druhy paralelizmu, využitie paralelizmu v aplikáciách vysokého výkonu a zodpovedajúce programové modely a paralelné architektúry, ich výhody a nevýhody.

Dekompozícia paralelných problémov
- efektívny paralelný problém možno rozoznať vo vzťahu pre idealizované typy, ktoré sú vlastné tomuto problému.
- Môžu byť aj navzájom kombinované

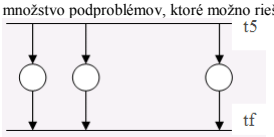
Všeobecné pravidlá dekompozície:
- pre priradenie úlohy pre processor musia byť zdroje dostatočné na to, aby nespôsobili zníženie efektivity, resp. zrútenie výpočtu.

1) Nežiadúce vyradenie procesu z hlavnej pamäte v dôsledku nevhodnej záťaže procesu
- v dôsledku preťaženia spôsobeného nevhodnou dekompozíciou
- preťaženie v dôsledku priradenie času v clusteroch

2) nežiadúce vyradenie procesov v dôsledku nedostatočnej pamäte
3) nevyvážené použitie blokujúcich/neblokujúcich procedúr pri odovzdávaní správ
4) zrútenie v dôsledku zablokovania
5) zrútenie výpočtu v dôsledku nedostatočnej pamäte.

Idealizované typy paralelizmu:
-jednoduchý
-prúdový
-expanzívny
-zmiešaný

Jednoduchý paralelilzmus: problém P je možné rozdeliť na množstvo podproblémov, ktoré možno riešiť nezávisle.



Prúdový paralelizmus: problém P je rozdelený na množinu podproblémov, ktoré sú riešené postupne – v následnosti

t.j. je hrubozrný paralelizmus, problémom je ťažké dekomponovať na veľké množstvo podproblémov.

Otázka č.4: Definícia zrýchlenia a efektívnosti paralelného výpoctu a celkového zrýchlenia a efektívnosti. Odvodenie Amdahlovho pravidla pre neškálovatelné a škálovatelné problémy, porovnanie obidvoch prístupov. Meranie a výpocet podielu paralelného kódu.

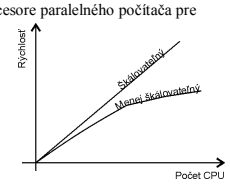
Zrýchlenie a účinnosť
Nech T(u,1) je sekvenčný čas na 1 procese
T(u,p) je paralelný čas na p procesoch

Zrýchlenie $S(u,p)=\frac{T(u,1)}{T(u,p)}$ účinnosť

$$E(u,p)=\frac{S(u,p)}{p}$$

Škálovateľnosť – v ideálnom prípade S(u,p)=p a E(u,p)=100%
- v praxi však S(u,p)<p a E(u,p)<100%
- programy sú škálovateľné, ak sú efektívne vykonávané na veľkom množstve procesov
- škálovateľné problémy – jemnozrnné
- menej škálovateľné problémy – hrubozrnné

Celkové zrýchlenie, celková účinnosť
$T_{BEST}(u)$ – čas na 1 procesore paralelného počítača pre vykonávanie najrýchlejšieho známeho sekvenčného algoritmu



- celkové zrýchlenie

$$\overline{S}(u,p)=\frac{T_{BEST}(u)}{T(u,p)}$$, celková účinnosť

$$\overline{E}(u,p)=\frac{\overline{S}(u,p)}{p}$$

Amdahlovho pravidlo – zrýchlenie, ktoré je dosiahnuté na paralel. počítači môže byť obmedzené existenciou malej časti rekurentne sekvenč. kódu, ktoré nie je možné paralelizovať.
Nech $\alpha \in \langle 0,1 \rangle$ je časť kódu, ktorá musí byť vykonávaná sekvenčne, max. zrýchlenie na paralel. počítači je ohraničená vzťahom.

$$S(u,p)=\frac{1}{\alpha+(1-\alpha)p} \le \frac{1}{\alpha}$$

Otázka č.5: Spôsoby a prostriedky využitia masívneho paralelizmu v programovom modeli údajového paralelizmu. Hlavné metódy využitia expanzívneho paralelizmu v programovom
modeli údajového paralelizmu. Odvodenie redukovaného poctu procesorov pri zachovaní optimálneho paralelného algoritmu.

Expanzívny paralelizmus – založený na rekurzívnom delení problému P
- je buď jemnozrný aleob hrubozrný – záleží od počtu delení problému P
- ak veľkosť problému nie je známa treba to ošetriť – SW náklady
- sú efektívne riešiteľné na SIMD aj MIMD
- pri SIMD môže byť algoritmus optimálny
- pri MIMD nie je tento algoritmus výhodný lebo rastie čas na komunikáciu
- ak m je počet úrovní riešenia, potom veľkosť probl. je n=2^m – 1 a čas výpočtu je O(log n)

Masívny paralelizmus – problém môže byť rieš. na zákl. množ. údajov rovnakým spôsobom
- je to jemnozrný algoritmus – rieši elementárne úlohy vzhľadom na kompletný problém
- SIMD – veľmi vhodná architektúra – jednoduchý programovací jazyk pre paralelizmy
- MIMD – vysoké SW náklady
- Ide o riešenie nezávislých úloh s nezávislými údajmi
- Dekompozícia údajov je jednoduchá
- Ak je algoritmus kombinovaný s inými druzhmi paralelizmu vtedy je algoritmus hrubozrný

Využitie paralelizmu u vysokorýchlostných aplikácií
- môžeme ich vyvinúť na základe masívneho a expanzívneho paralelizmu
- problémy sú inherentné a zväčša jemnozrné
- jednoducjé aprúdové paralel. sú hrubozrné – vysokorýchlostný výpočet je ťažké realizovať

Modely pre vysokorýchlostné aplikácie
- model údajového paralelizmu – podporuje SIMD aj MIMD
- odovzdávanie správ – podporuje MIMD s DM

Model púdajového paralelizmu – je určený pre SIMD avšak je ho možné riešiť aj v MIMD – čas an komunikáciu rastie
- základom je vektorové programovanie s použitím vysokorýchlostných vektorových operácií
- je potrebné nájsť nezávislú množinu údajov a paralelizovať cykly

Redukcia počtu procesov
n – počet čísel, n/2 – počte procesorov, O(log n) – zložitosť
- algoritmy ktoré pracujú v O(log n) sú optimálne
- ak je algoritmtus optimálny, počet procesorov je p=n/log n

1.) majme p<n/2 rozdelíme n čísel na p skupín (p-1) skupín má $\lceil n/p \rceil$, zvyšok má n=(p-1)* $\lceil n/p \rceil$
2.) ku každej skupine priradíme jeden procesor ktorý v rámci svojej skupiny pracuje v čase $\lceil n/p \rceil$-1+log p sekvenčne
3.) nahradíme p=n/log n s $\lceil n/(n/\log n) \rceil$ - + log (n/log n)
4.) p=n/log n je redukovaný počet procesorov

Otázka č.6: Charakteristika programového modelu odovzdávania správ, definícia a dôsledok latentnosti. Charakteristika modelu SPMD a MPMD. Základné druhy operácií pre odovzdávanie správ medzi dvoma procesmi v MPI a ich sémantika z hladiska volania, návratu a ukoncenia procedúry MPI.

Programový model odovzdávania správ
1.) programátor definuje sekv. procesy, kt. na základe odovzdávania správ riešia nejakú úlohu
2.) spôsob ktorým sú procesy synchronizované a ktorým si navzájom vymieňajú údaje je určený programátorom
3.) používa sa sekvenčný programovací jazyk a operácie pre odovzdávanie správ sú realizované prostredníctvom volania interfacu MPI, ktorý zabezpečuje výmenu správ v sieti na základe vzájomnej väzby procesov
4.) najznámejší model je model 1 programu, ktorý spracováva viacnásobné údaje SPMD

Charaktiristika SPMD
1.) v určitom čase môžu rozdielne procesy vykonávať tú istú alebo rôznu časť programu
2.) pamäť ktorú procesy využívajú je lokálna
3.) komunikácia je realizovaná volaniami špeciálnych procedúr

Zníženie komunikačných nákladov
- návrhom vhodnej topológie procesov pri dekompozícii problému – výsledkom je skrátenie ciest medzi procesormi
- maximalizácia dĺžok správ a minimalizácia počtu volaní procedúr pre komunikáciu
- prekrytie komunik. a výpočtu v čase použitím neblok. operácií pre odovzdávanie správ

Komunikčné typy
- proces vysielajúci správu
- proces prijímajúci správu

Komunikácia medzi 2 procesmi môže byť
- blokujúca – po príchode z volanej funkcie môžeme všetky zdroje používať
- neblokujúca – k návratu z volanej funckie môže dôjsť ešte pred ukončením operácií – ešte predtým ako možno opätovne použiť zdroje

Otázka č.7: Blokujúce a neblokujúce operácie MPI, ich porovnanie z hladiska využitia zdrojov
a zablokovania výpoctu. Vztah k asynchrónnemu a synchrónnemu odovzdávaniu správ - rendezvous. Poradie správ a vztah k deterministickému výpoctu.
Typy komunikácie:blokujúca,neblokujúca
Blokujúca:
Štandardný send
-int MPI_Send(void *buff,int count, MPI_DATATYPE,int disp, int tag,MPI_COMM comm)
-môže ale nemusí byť buffrovaný (závisí od inplementácie a dlžky správ)
-nelokálny(lebo musí byť buffrovaný) závisí na inom procese
Buffrovaný send:
-int MPI_Bsend(…)
-môže byť ukončený skôr, než príjemnca zavolá receive
-lokálny
-pripojenie a odpojenie buffra:
1. alokácia miesta v jazyku C
2. alokácia int MPI_Buffer_attach(void * buffer, int size)
3.buffer môžeme použiť rôzne
4.int MPI_Buffer_detach(void * buffer, int *size))
Synchrónny send
-int Mpi_Ssend(…)
-neskončí sa , pokiaľ príjemca nezavolá recevve a kým buffer nemôže byť použitý
-ukončenie inplikuje že receive prebehol
-externá rutina na oneskorenie
Blokujúci receive
-MPI_Status-prímač môže špecifikovať ľubovoľnú hodnotu
-MPI_ANY_SOURCE –zdroj
-MPI_ANY_TAG- značky
-potom údaje vieme získať zo statusu – status.MPI_SOURCE,status.MPI_TAG,status.MPI_ERROR
Neblokujúca:
Štandardný send
-int MPI_Isend(...,MPI_Status *status,$ MPI_Reguest * req)
-môžeme testovať v akom je stave MPI_Test
-MPI_Wait čaká, kým nedôjde k odozve
-req – vnútorná štruktúra nás nezaujíma
Posielanie správ
-správy sa nepredbiehajú
-prosesy príjmajú sprúávy v takom poradí v akom boli odoslané
-podmienka je, že procesy sú jednovláknové a proces nepoužije procedúru na prijatie akejkoľvek správy
-ak sú procesy viacvláknové, nie je možne určiť relatívne usporiadanie operácií v čase v rôznych vláknach

Otázka č.8: Princíp skupinovej komunikácie v MPI, definícia hlavného procesu, skupinové operácie
v MPI, ich syntax, sémantika a aplikácia.

Princíp skupinovej komunikácie v MPI:
Všetky procesy v skupine potrebujú zavolať funkciu pre odovzdávanie správ. Niektorý proces v skupine môže byť root ak je to potrebné. Nie všetky skupinové komunikácie musia mať určený root proces.
Volanie kolektívnej komunikácie:
Napríklad , všetky procesy volajú Bcast funkciu indikujúcu ten istý root proces.
Broadcast
Root zapíše data do buffra a všetci ostatný čítajú čo zapísal root.
int MPI_Bcast(void *buf, int count,MPI_Datatype datatype, int root, MPI_Comm comm)
INOUT buf – začiatočná adresa buffra
IN count počet entít v buffery
IN datatype typ buffru
IN root rank ktorý je root v broadcaste (integer)
IN comm komunikátor (handle)
Reduce
Rootovský proces vykoná danú operáciu (op) nad dátami od ostatných procesov.
int MPI_Reduce (void *sendbuf, void *recvbuf,int count, MPI_Datatype datatype,MPI_Op op, int root, MPI_Comm comm)
IN sendbuf adresa odosielajúceho buffra
OUT recvbuf adresa príjmajúceho buffra (adresa roota)
IN count počet elementov v posielajúcom buffry (integer)
IN datatype typ dát v odosielajúcom buffery (handle)
IN op operácia s dátami (handle)
IN root rank, ktorý proces je root (integer)
IN comm komunikátor (handle**)**
Scatter
Hlavný proces pošle množinu dát, ktoré si ostatné procesy rozdelia.
int MPI_Scatter (void *sendbuf, int sendcount,MPI_Datatype sendtype, void *recvbuf,int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
IN sendbuf adresa odosielajúceho buffra
IN sendcount počet elementov v posielajúcom buffry (iba root)
IN sendtype typ dát v odosielajúcom buffery
IN recvbuf adresa prijmajúceho buffra
IN recvcount počet elementov príjmajúcom buffry
OUT recvtype typ dát v prijmajúcom buffry
IN root rank odosielajúceho procesu
IN comm komunikátor
Gather
Hlavný proces proces poskladá vo svojom buffery všetky podčasti ktoré mu pošlu ostatné procesy.
int MPI_Gather (void *sendbuf, int sendcount,MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
IN sendbuf adresa odosielajúceho buffera
IN sendcount počet elementov v odosielajúcom buffery
IN sendtype typ posielaných dát v buffery
OUT recvbuf adresa príjmajúceho buffera (root)
IN recvcount počet elementov v príjmajúcom buffery
IN recvtype typ dát v príjmajúcom buffey
IN root rank príjmajúceho procesu
IN comm komunikátor
Dalšie funkcie MPI_Allgather a MPI_Alltoall

```
#include <mpi.h>
#define BUFSIZE 10
int main(argc, argv)
int argc; char *argv[];
{ int size, rank;
int buf[BUFSIZE]={0,0,0,0,0,0,0,0,0,0};
int n, value;
float rval;
MPI_Status status;

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
if (size==4) { /* Correct number of processes */
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank==0) { /* Master is root, init send buffer */
buf[0]=5; buf[1]=1; buf[2]=8; buf[3]=7; buf[4]=6;
buf[5]=5; buf[6]=4; buf[7]=2; buf[8]=3; buf[9]=1;
printf("\n Broadcasting {5,1,8,7,6,5,4,2,3,1}");
}
/* broadcasting */
MPI_Bcast(buf,10,MPI_INT,0,MPI_COMM_WORLD);
if (rank==0) { /* Master */
printf("\n Computing by master and slaves");
value=100;
for (n=0;n<BUFSIZE;n++) {
if (value>buf[n]) { value=buf[n]; }
}
printf("\n Minimum %4d by master ",value);

MPI_Recv(&value, 1, MPI_INT, 1, 0, MPI_COMM_WORLD,
&status);
printf("\n Maximum %4d from slave 1",value);
MPI_Recv(&value, 1, MPI_INT, 2, 0, MPI_COMM_WORLD,
&status);
printf("\n Sum %4d from slave 2",value);
MPI_Recv(&rval, 1, MPI_FLOAT, 3, 0,
MPI_COMM_WORLD, &status);
printf("\n Average %4.2f from slave 3\n",rval);
} else if (rank==1) { /* max slave */
value=0;
for (n=0;n<BUFSIZE;n++) { if (value<buf[n]) { value=buf[n]; }
}
MPI_Send(&value, 1, MPI_INT, 0, 0,
MPI_COMM_WORLD);

} else if (rank==2) { /* sum slave */
value=0;
for (n=0;n<BUFSIZE;n++) { value=value+buf[n]; }
/* send sum to master */
MPI_Send(&value, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
/* send sum to slave 3 */
MPI_Send(&value, 1, MPI_INT, 3, 0, MPI_COMM_WORLD);
} else if (rank==3) { /* ave slave */
MPI_Recv(&value, 1, MPI_INT, 2, 0, MPI_COMM_WORLD,
&status);
rval= (float) value / BUFSIZE;
MPI_Send(&rval, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
} } MPI_Finalize(); return(0);
}
```

Otázka č.9.: Základné typy a odvodené typy MPI a spôsob definície nového typu. Zobrazenie
typu, posunutie, rozsah, spodná a horná hranica nového typu. Definícia, porovnanie a využitie odvodených typov. Stlácanie a porovnanie komunikácie na základe odvodeného typu oproti komunikácii prostredníctvom stlácania.

Základné typy: MPI_CHAR, MPI_SHORT, MPI_INT, MPI_FLOAT, MPI_UNSIGNED_LONG, MPI_INSIGNED, MPI_DOUBLE, MPI_BYTE, MPI_PACKED
Definované typy:
- contignous – jedna dĺžka poľa, žiaden posun, jeden údajový typ
- vector - jedna dĺžka poľa, jeden posun, jeden údajový typ
- indexed - viac dĺžok poľa, viac posunov, jeden údajový typ
- structure - viac dĺžok poľa, viac posunov, viac údajových typov

Definícia typu:
1. definujeme meno pre nový údajový typ MPI_Datatype datatype
2. výpočítame argumenty pre inštruktor
3. použijeme konštruktor nového typu na odovzdanie nového typu
4. zavedieme nový údajový typ do MPI MPI_Type_Commit (&datatype)

- rozsah typu MPI
  Typemap = f{(type0, disp0), ... , (type n-1, disp n-1)}
  Lb(Typemap)= min{disp j} ak žiadny typ nie je MPI_LB
- spodná hranica
  Ub(Typemap)= max {(disp j – size of (typ j))}+{ak žiadny typ nie je MPI_UB}
- horná hranica

Ak type i potrebuje zarovnanie extend (Typemap) = Ub(Typemap) – Lb(Typemap)
v MPI: MPI_Type_Lb(), MPI_Type_Ub

Otázka č.10: Dôvody a spôsoby konštrukcie nových komunikátorov v MPI, rozdiel medzi skupinami procesov a komunikátormi. Rozdiel medzi intrakomunikátormi a interkomunikátormi. Druhy topológií procesov a ich uplatnenie v komunikátoroch.

Komunikator – skupina komunikujúcich sekvenčných procesov, tieto tvoria svet, procesy nemôžu komunikovať mimo komunikácie
Interkomunikatory:
- lokálne skupiny procesov
- vzdialená skupina procesov

Infrakomunikaroty – tvorí súčasť skupiny procesov
- komunikácia medzi 2 procesmi
- skupinová komunikácia

MPI_COMM_WORLD – skupina procesov
MPI_COMM_PARENT – interkomunikácia medzi dvoma skupinami
MPI_COMM_SELF - špecifická konštanta, ktorá obsahuje iba volajúci proces
Vytváranie intrakomunikatorov:
- duplikát MPI_COMM_DUP
- zlučovanie MPI_COMM_SPLIT
- vyber skupiny MPI_COMM_GROUP
  - použitie skupiny operácií (prienik, zjednotenie)
  - vytvorenie nového komunikatora

Interkomunikatory:
- vytvárajú sa na komunikáciu medzi procesmi, ktoré sa nachádzajú v rôznych intrakomunikatoroch
- interkomunikator je tvorený – skupinou procesov + kontext + topol=ogia (grafová, karteziánska)