ՀԱՅԱՍՏԱՆԻ ՀԱՆՐԱՊԵՏՈՒԹՅԱՆ
ԿՐԹՈՒԹՅԱՆ ԵՎ ԳԻՏՈՒԹՅԱՆ ՆԱԽԱՐԱՐՈՒԹՅՈՒՆ
ՀԱՅԱՍՏԱՆԻ ՊԵՏԱԿԱՆ ՃԱՐՏԱՐԱԳԻՏԱԿԱՆ ՀԱՄԱԼՍԱՐԱՆ
(ՊՈԼԻՏԵԽՆԻԿ)

Կիրառական մաթեմատիկա և ֆիզիկա
ֆակուլտետի
Մասնագիտական մաթեմատիկական
Կրթության ամբիոն

ԶՈՒԳԱՀԵՌ ԾՐԱԳՐԱՎՈՐՈՒՄ MPI ՄԻՋԱՎԱՅՐՈՒՄ

Ուսումնական ձեռնարկ

ԵՐԵՎԱՆ
ՃԱՐՏԱՐԱԳԵՏ
2014

ՀՏԴ
ԳՄԴ

Հրատարակվում է Հայաստանի պետական ճարտարագիտական համալսարանի 22.12.2011թ. գիտական խորհրդի նիստում հաստատված 2012թ. հրատարակչական պլանի համաձայն

Գրախոսներ՝ ֆ.մ.գ.թ.,դոցենտ **Ի. Հովհաննիսյան**

ֆ.մ.գ.թ., դոցենտ **Վ.Սահակյան**

Գյուրջյան Մ.Ղ.

ԶՈՒԳԱՀԵՌ ԾՐԱԳՐԱՎՈՐՈՒՄ MPI ՄԻՋԱՎԱՅՐՈՒՄ: Ուսումնական ձեռնարկ/ Մ.Ղ.Գյուրջյան; ՀՊՃՀ, Եր.: Ճարտարագետ 2014.- 100 էջ:

Աշխատանքը նվիրված է MPI (Message Passing Interface) միջավայրում զուգահեռ ծրագրավորման գործնական դասընթացի յուրացմանը: Ներկայումս MPI տեխնոլոգիան կլաստերային համակարգերի և տարաբաշխված հիշողությամբ քոմփյութերների համար նախատեսված զուգահեռ ծրագրավորման հիմնական միջոցն է: Դասընթացը ներառում է MPI ստանդարտի բավականին մեծ թվով պրոցեդուրաների նկարագրություն՝ իրենց օգտագործման օրինակներով, ինչպես նաև պրակտիկ տեղեկություններ, որոնք կարող են օգտակար լինել իրական ծրագրերի կազմման ժամանակ: Ենթադրվում է, որ ուսանողն արդեն տիրապետում է ծրագրավորման C լեզվին: Ստանդարտ MPI ծրագրերի մշակման ժամանակ քննարկվում են զուգահեռացման ռազմավարությունները: Ձեռնարկում դիտարկված բոլոր ծրագրերը ռեալ աշխատում են ԱրմԿլաստեր բարձր հաշվողական համակարգի վրա:

Ուսումնական ձեռնարկը նախատեսված է «Ինֆորմատիկա և կիրառական մաթեմատիկա» մասնագիտության ուսանողների, ասպիրանտների և մասնագետների լայն զանգվածի համար, ովքեր ցանկություն ունեն սովորել և օգտագործել զուգահեռ ծրագրավորումը MPI միջավայրում՝ բարդագույն խնդիրներ լուծելու համար:

ՀՏԴ
ԳՄԴ

ISBN

© ՃԱՐՏԱՐԱԳԵՏ 2014
© Գյուրջյան Մ.Ղ. 2014

Բովանդակություն

Ներածություն

Զուգահեռ ծրագրավորումը զուգահեռ պրոցեսների և դրանց փոխազդեցությունների միջև ծրագրավորում է, որն ավանդական հաջորդական ծրագրավորմանն ավելացնում է նոր ձևաչափ և ծրագրավորողին տալիս դեկոմպոզիցիայի գործիք՝ բարդ խնդիրներ իրականացնելու համար:

Զուգահեռ հաշվարկների տեսության մեջ գոյություն ունեն մի շարք բաց խնդիրներ, որոնցից հիմնականը ալգորիթմերի արդյունավետ զուգահեռացման հնարավորությունն է: Զուգահեռ ալգորիթմի արդյունավետությունը էապես կախված է հաշվարկման ժամանակի և համակարգիչների միջև հաղորդակցման ժամանակի (տվյալների փոխանցման ժամանակ) հարաբերակցությունից: Հաղորդակցությունների փոխանցում ունեցող զուգահեռ համակարգերի օպտիմալ հարաբերակցությունը հաշվարկների և հաղորդակցման միջև ապահովում են խոշորահատիկ զուգահեռացման մեթոդները, երբ զուգահեռ ալգորիթմները կառուցվում են խոշոր և հազվադեպ հարաբերակցվող բլոկներից: Գծային հանրահաշվի խնդիրները, ցանցային մեթոդներով լուծվող խնդիրները և մի շարք ուրիշներ բավականին արդյունավետ զուգահեռացվում են խոշորահատիկ մեթոդներով: Արդյունավետ զուգահեռ ծրագիր գրելն ավելի բարդ է, քան հաջորդականը գրելը: Ծրագիրը համարվում է արդյունավետ նաև այն դեպքում, երբ նրա կատարման ժամանակ բեռնված են բոլոր պրոցեսորները: Զուգահեռ ծրագրերի ստեղծման ժամանակ անհրաժեշտ է ընտրել զուգահեռ ճյուղերի օպտիմալ քանակը, ուստի իդեալական զուգահեռ ծրագիրը պետք է օժտված լինի հետևյալ հատկություններով.

- Զուգահեռ կատարվող ճյուղերի երկարությունները պետք է միմյանց հավասար լինեն;

- Լրիվ բացառվեն տվյալների սպասման, ղեկավարման, փոխանցման և ընդհանուր ռեսուրսների օգտագործման կոնֆլիկտների հետևանքով առաջացած դադարները;
- Տվյալների փոխանցումը պետք է լրիվ համատեղելի լինի հաշվարկների հետ:

Արդյունավետ զուգահեռացվող խնդիրների հիմնական դասերն են.

- *Միաչափ զանգվածները*. այսպիսի խնդիրներ հաճախ են հանդիպում: Եթե զանգվածի տարրերի արժեքները որոշվում են բավականին բարդ արտահայտություններով, իսկ դրանք պետք է հաշվարկել բազմաթիվ անգամներ, ապա ցիկլի զուգահեռացումը՝ զանգվածի տարրերի արժեքները հաշվարկելու համար, կարող է շատ արդյունավետ լինել:
- *Երկչափ զանգվածները*. մատրիցների հետ բոլոր գործողությունները գործնականում կարող են կատարվել կլաստերի վրա: Գծային հանրահաշվի բազմաթիվ ալգորիթմեր կարող են արդյունավետ զուգահեռացվել:
- *Դիֆերենցիալ հավասարումների համակարգերը*. դիֆերենցիալ հավասարումների համակարգերի լուծումը հանդիպում է բազմաթիվ ինժեներական և գիտական խնդիրներում: Շատ դեպքերում այդպիսի խնդիրների լուծման ալգորիթմերը կարելի է արդյունավետ զուգահեռացնել՝ կլաստերային համակարգիչների վրա մշակելու համար: Որպես օրինակ կարելի է հիշատակել այնպիսի խնդիրներ, ինչպիսիք են՝ մոլեկուլյար մոդելները, ճարտարագիտական հաշվարկները, N մարմինների մոդելները, գազադինամիկան, էլեկտրադինամիկան և այլն:

Զուգահեռ ծրագրավորման առավել հայտնի տեխնոլոգիաներից է MPI-ը (Message Passing Interface)[11,24]: MPI –ը զուգահեռ կատարվող ծրագրերի ստեղծման տեխնոլոգիա է, որի հիմքում ընկած է պրոցեսների միջև հաղորդագրությունների

փոխանցումը (այդ պրոցեսները կարող են կատարվել ինչպես մեկ, այնպես էլ տարբեր հաշվողական հանգույցների վրա):

Ֆորմալ առումով MPI մոտեցումը հիմնված է ծրագրային մոդուլներում հատուկ գրադարանի ֆունկցիաների կանչերի և հաշվողական հանգույցներում զուգահեռ կատարվող ծրագրի բեռնիչը ներառելու վրա: Նմանատիպ գրադարաններ առկա են գրեթե բոլոր հարթակների համար, ուստի MPI տեխնոլոգիայով գրված կիրառական ծրագրի ճյուղերի փոխազդեցությունը կախված չէ մեքենայի ճարտարապետությունից (կարող են կատարվել և մեկ պրոցեսորանի, և բազմապրոցեսորանի ինչպես տարաբաշխված, այնպես էլ ընդհանուր հիշողությամբ ԷՀՄ-ների վրա), ճյուղերի դասավորվածությունից (կարող է կատարվել մեկ կամ տարբեր պրոցեսորների վրա), կոնկրետ օպերացիոն համակարգի API-ից (Application Program Interface):

MPI–ի պատմությունը սկիզբ է առնում 1992թ.-ից, երբ Oak Ridge National Laboratory (Rice University) լաբորատորիայի կողմից մշակվեց հաղորդագրությունների փոխանցման արդյունավետ և տեղափոխելի մի մոդել: MPI 1.0 ստանդարտի վերջնական տարբերակը ցուցադրվել է 1995թ., իսկ MPI-2 ստանդարտը տպագրվել է 1997թ.: MPI-ի ռեալիզացիաներից է MPICH (բոլոր UNIX համակարգերը և Windows'NT-ն [12]), LAM (UNIX-անման ՕՀ-երը [13,14]), CHIMP/MPI [16] , WMPI [15]. MPI-2 ստանդարտի հիմնական տարբերությունները հետևյալն են. պրոցեսների դինամիկ առաջացումը, զուգահեռ մուտք-ելքը, C++-ի համար ինտերֆեյսը, ընդլայնված կոլեկտիվ ֆունկցիաները, պրոցեսների՝ մեկ ուղղությամբ փոխազդեցությունը [1]:

Նշենք, որ MPI-ը ծրագրավորողի համար բավականին ցածր մակարդակի գործիք է: MPI-ով ստեղծված են թվային մեթոդների համար նախատեսված մասնագիտացված գրադարաններ, մասնավորապես՝ գծային հավասարումների համակարգի

լուծում, մատրիցների օրթոգոնալ ձևափոխություն, սեփական արժեքների որոնում, ScaLAPACK[17] գրադարաններ, AZTEC և այլն:

MPI ֆունկցիաների բազմության մեջ են մտնում հատուկ կանչեր՝ բացառիկ իրավիճակների մշակման և կարգաբերման համար: Դրանց նպատակն է հաղորդագրության միջոցով օգտագործողին հայտնելու այնպիսի սխալների մասին, ինչպիսիք են MPI-ին չվերաբերող գործողությունները, օր.՝ մուտք-ելքի բուֆերների դատարկումը ծրագրից դուրս գալուց առաջ: MPI-ի իրականացումները թույլ տալիս են սխալի առաջացման դեպքում շարունակել կիրառական ծրագիրը:

MPI ծրագրի լայնամասշտաբ կարգաբերումը բավականին բարդ է այն պատճառով, որ միաժամանակ կատարվում են մի քանի ծրագրային ճյուղեր (դրան ավելացրած նաև այն, որ ելքային տեքստում տպման օպերատորների ներգրավման միջոցով ծրագրի լավացման ավանդական եղանակը բարդանում է, քանի որ ինֆորմացիայի դուրսբերման ֆայլում իրար են խառնվում նաև MPI-ծրագրի տարբեր ճյուղերից եկող ինֆորմացիաները): Ամենահզոր կարգաբերիչներից մեկը TotalView-ն է [18]: Նման կարգաբերիչների ֆունկցիաների մեջ է մտնում զուգահեռ ծրագրում փոխանակումների մասին ինֆորմացիայի դուրսբերումը՝ հաճախ գրաֆիկական տեսքով:

Խիստ ձևակերպմամբ MPI տեխնոլոգիան ենթադրում է MPMD (Multiple Program– Multiple Data՝ բազմաթիվ ծրագրեր-բազմաթիվ տվյալներ) մոտեցումը: MPMD մոդելում միևնույն ժամանակ և միմյանցից անկախ տարբեր հաշվողական հանգույցներում կատարվում են մի քանի ծրագրային ճյուղեր, որոնք փոխանակվում են տվյալներով նախապես սահմանված ժամանակային կտրվածքներում: Սակայն այդպիսի ծրագրերը կոդավորման ժամանակ հսկայածավալ են լինում (յուրաքանչ-

յուր հաշվողական հանգույցի համար գրվում է առանձին ելքային տեքստ): Այդ իսկ պատճառով գործնականում հիմնականում կիրառվում է SPMD-մոտեցումը (Single Program – Multiple Data՝ մեկ ծրագիր –բազմաթիվ տվյալներ): SPMD- մոդելում տարբեր հաշվողական համակարգերի վրա աշխատում է նույնատիպի ծրագրային կոդը, ծրագրի բոլորն սկսում են կատարվել միաժամանակ, որպես UNIX պրոցեսներ, ճյուղերի քանակը ֆիքսված է՝ համաձայն MPI 1.1 ստանդարտի, ինչպես նաև աշխատանքի ընթացքում հնարավոր չէ ստեղծել նոր ճյուղեր:

Հետագայում կենթադրենք, որ զուգահեռ ծրագիրը բաղկացած է մի քանի ճյուղերից (կամ պրոցեսներից, կամ խնդի-րներից), որոնք միաժամանակ կատարվում են հաշվողական հանգույցների վրա, իսկ պրոցեսները իրար հետ փոխանակվում են հաղորդագրությունների միջոցով: Ամեն հաղորդագրություն ունի իդենտիֆիկատոր, ինչը հնարավորություն է տալիս ծրագրին ու կապերի գրադարանին տարբերել դրանք: MPI-ում գոյություն ունի «կապման տարածք» հասկացություն: Կապման տարածքներն ունեն միմյանցից անկախ պրոցեսների համարակալում (կոմունիկատորը հենց որոշում է այդ կապման տարածքը):

Ընդհանուր դեպքում զուգահեռ ծրագրի ստեղծումը պարունակում է երկու հիմնական մաս.

- Ելքային հաջորդական ալգորիթմը ենթարկվում է դեկոմպոզիցիայի՝ զուգահեռացման, այսինքն՝ տրոհվում է միմյանցից անկախ աշխատող ճյուղերի: Փոխկապակցման համար ճյուղերում ներառվում են երկու տեսակի լրացուցիչ ոչ մաթեմատիկական օպերացիաներ՝ տվյալների ընդունում (Receive) և փոխանցում (Send):
- Զուգահեռացված ալգորիթմը ձևակերպվում է որպես ծրագիր, որտեղ ընդունման և փոխանցման օպերացիաները

գրանցվում են ճյուղերի միջև որոշակի համակարգի տերմիններով (մեր դեպքում MPI):

Ավելի մանրամասն այս պրոցեսները նկարագրվում են [2,3]-ում, իսկ ովքեր ցանկանում են լրջորեն զբաղվել այս ամենով, խորհուրդ են տրվում [1,4] աշխատանքները:

Ծրագրավորողի տեսակետից գոյություն ունի երկու հիմնական մեթոդ՝ պարադիգմաներ. տվյալները կարող են փոխանցվել տարաբաշխված օպերատիվ հիշողության միջոցով (այդպիսի հիշողությանը ճյուղերի դիմման սինքրոնացումը տեղի է ունենում սեմաֆորների միջոցով) և հաղորդագրությունների միջոցով: Առաջին մեթոդը հիմնական է համարվում բոլոր պրոցեսորների համար ընդհանուր հիշողությամբ ԷՀՄ-ների համար, իսկ երկրորդը՝ ցանցի միջոցով իրար հետ կապված հաշվողական հանգույցների դեպքում: Այս երկու պարադիգմաներից յուրաքանչյուրը կարող է իմիտացվել մյուսով:

Հարկ է նշել, որ MPI-ը բավականին մեծ և բարդ գրադարան է, որը բաղկացած է մոտավորապես 130 ֆունկցիայից, որոնց մեջ մտնում են.

- MPI պրոցեսների ինիցիալիզացման և փակման ֆունկցիաներ,
- Ֆունկցիաներ, որոնք իրականացնում են կետ-կետ տիպի հաղորդակցման օպերացիաներ,
- Ֆունկցիաներ, որոնք իրականացնում են կոլեկտիվ օպերացիաներ,
- Պրոցեսների խմբերի և հաղորդակցման միջոցների հետ աշխատելու ֆունկցիաներ,
- Տվյալների կառուցվածքների հետ աշխատող ֆունկցիաներ,
- Պրոցեսների տոպոլոգիաների ձևավորման ֆունկցիաներ:

MPI-ում սահմանված է երեք դասի ֆունկցիա՝ արգելափակող, լոկալ և կոլեկտիվ:

- Արգելափակող ֆունկցիաները կանգնեցնում են պրոցեսի աշխատանքը մինչ այն պահը, երբ արդեն իրենց կողմից իրականացված օպերացիան կատարվել է: Դրան հակառակ՝ չարգելափակող ֆունկցիաները անմիջապես հետ են վերադարձնում ղեկավարումը իրենց կանչած ծրագրին, իսկ օպերացիայի իրականացումը կատարվում է ֆոնային ռեժիմում: Չարգելափակող ֆունցիաները հետ են վերադարձնում չեկեր (requests), որոք վերանում են ավարտելիս: Մինչև ոչնչանալը չարգելափակող ֆունկցիայի արգումենտներ հանդիսացող փոփոխականների և զանգվածների հետ ոչինչ անել չի կարելի:
- Լոկալ ֆունկցիաները ճյուղերի միջև չեն իրականացնում տվյալների փոխանցումներ: Լոկալ է հանդիսանում տեղեկատվական ֆունկցիաների մեծ մասը, քանի որ համակարգային տվյալների պատճեններն արդեն իսկ պահված են յուրաքանչյուր ճյուղում: MPI_Send փոխանցման ֆունկցիան և MPI_Barrier սինքրոնացման ֆունկցիան լոկալ ֆունկցիա չէ, քանի որ այն իրականացնում է փոխանցում: Միևնույն ժամանակ MPI_Recv ընդունման ֆունկցիան լոկալ ֆունկցիա է. այն միայն պասիվ վիճակով սպասում է տվյալների և ոչինչ չի ուղարկում մյուս ճյուղերին:
- Կոլեկտիվ ֆունկցիաները պետք է կանչվեն կոմունիկատորի բոլոր ճյուղերի կողմից (կոմունիկատորի հասկացությանը կծանոթանանք հաջորդիվ): Այս կանոնին չհետևելու արդյունքում հանգում ենք ծրագրային սխալների՝ հիմնականում ծրագրի <կախմանը> կատարման փուլում:

MPI-ում ճյուղերը միավորված են խմբերի մեջ: Դրանք օգնում են իրարից տարբերակել հաղորդագրությունները: Այն արվում է MPI ֆունկցիաների մեջ <կոմունիկատոր> տիպի

պարամետրի ներմուծմամբ, որը կարելի է դիտարկել նաև որպես խմբի դեսկրիպտոր (համար): Կոմունիկատորը սահմանափակում է տվյալ ֆունկցիայի գործողության տիրույթը համապատասխան խմբում: Խմբի կոմունիկատորը իր մեջ ընդգրկում է ծրագրի բոլոր ճյուղերը: Դա կատարվում է ավտոմատ կերպով MPI_INIT ֆունկցիայի կատարման ժամանակ և կոչվում է MPI_COMM_WORLD: Նշենք, որ հետագայում հնարավորություն կա ստեղծել նոր կոմունիկատորներ MPI_Comm_split ֆունկցիայի միջոցով (տես [2 -4]):

Կոմունիկացիաների ամենապարզ ֆունկցիաներից են <կետ-կետ> փոխանցումները: Այս դեպքում մասնակցում է երկու պրոցես, որոնցից մեկը հաղորդագրություն ընդունող է, իսկ մյուսը՝ ուղարկող: Ուղարկող պրոցեսը կանչում է տվյալների փոխանցման որևէ մի պրոցեդուրա և այնտեղ նշում ստացող պրոցեսի համարը կոմունիկատորում, իսկ ստացող պրոցեսը պետք է կանչի ընդունման պրոցեդուրաներից մեկը և այնտեղ նշի նույն կոմունիկատորը (երբեմն պարտադիր չէ իմանալ ուղարկող պրոցեսի ճիշտ համարը տրված կոմունիկատորում):

Այս խմբի բոլոր պրոցեդուրաները բաժանվում են երկու խմբի՝ արգելափակումով պրոցեդուրաներ (սինխրոնացման) և առանց արգելափակման պրոցեդուրաներ (ասինքրոն): Արգելափակումով փոխանակման պրոցեդուրաները կանգնեցնում են պրոցեսի աշխատանքը մինչ որոշակի պայմանի կատարումը, իսկ ասինխրոն պրոցեդուրաներից դուրս գալը կատարվում է համապատասխան կոմունիկացիոն օպերացիայի ինիցիալիզացիայից անմիջապես հետո: Արգելափակող պրոցեդուրաների ոչ ճիշտ օգտագործումը կարող է հանգեցնել փակուղային իրավիճակի (տես ստորև), այդ պատճառով էլ այդ դեպքերում ենթադրվում է առավել զգուշություն: Ասինք-

րոն օպերացիաների օգտագործման ժամանակ փակուղային իրավիճակներ չեն լինում, սակայն այս օպերացիաներն էլ պահանջում են զգուշություն՝ տվյալների զանգվածները օգտագործելիս:

<Կետ-կետ> տիպի պրոցեդուրաների օգտագործման ժամանակ ծրագրավորողը հնարավորություն ունի ընտրել պրոցեսների փոխադարձ գործողությունների եղանակը և MPI կոմունիկացիոն մոդուլի ու կանչող պրոցեսի փոխադարձ գործողությունների մեթոդը:

Պրոցեսների փոխադարձ գործողությունների եղանակները հետևյալն են.

- Տվյալների բուֆերացումն իրականացվում է ուղարկող մասում (այդ ժամանակ փոխանցող ֆունկցիան իր տրամադրության տակ է վերցնում ժամանակավոր բուֆերը, պատճենում այնտեղ հաղորդագրությունը և ղեկավարումը հանձնում կանչող պրոցեսին, իսկ բուֆերի պարունակությունը ֆոնային ռեժիմում փոխանցվում է ընդունող պրոցեսին):
- Սպասողական ռեժիմ ընդունող պրոցեսի կողմում (այս դեպքում հնարավոր է փոխանցող կողմում սխալի կոդի արտածում և ավարտ):
- Սպասողական վիճակ փոխանցող կողմում (ընդունող կողմում ավարտ՝ սխալի կոդով):
- Վերոհիշյալ երեք տարբերակից մեկի ավտոմատ ընտրություն (հենց այդպես են վարվում MPI_Recv և MPI_Send պարզագույն արգելափակող ֆունկցիաները):

Ըստ MPI կոմունիկացիոն մոդուլի և կանչող պրոցեսի փոխազդեցության մեթոդներն են.

- *Արգելափակող ռեժիմ.* ղեկավարումը կանչող պրոցեսին է վերադառնում միայն այն բանից հետո, երբ տվյալները

ընդունվել կամ փոխանցվել են (կամ ժամանակավոր բուֆերում պատճենահանվել են):

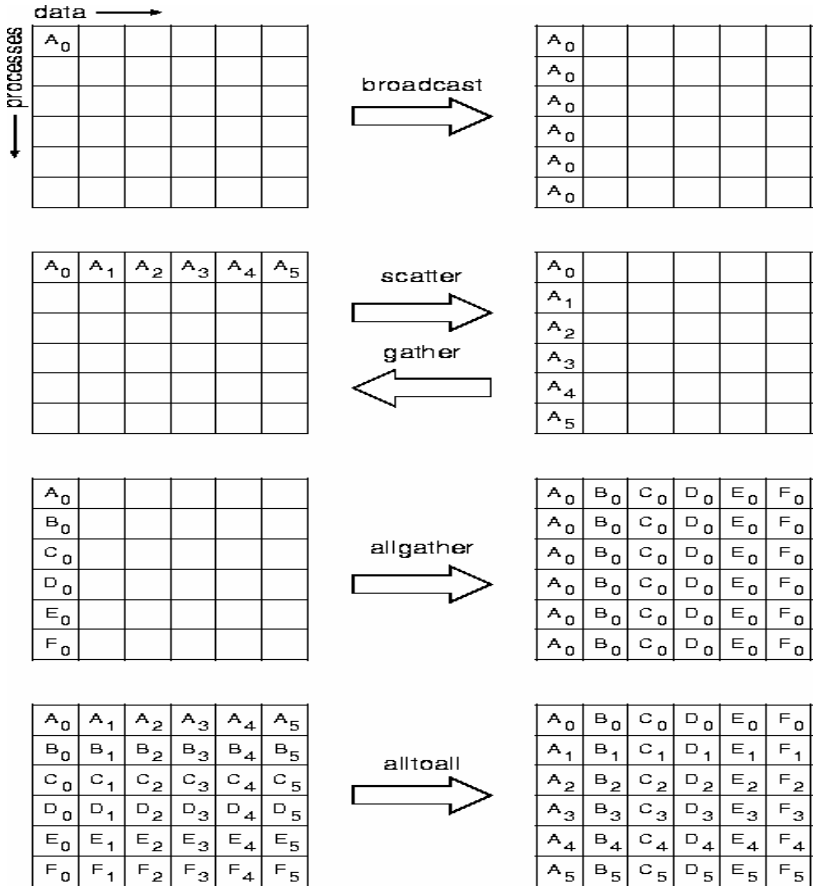
- *Չարգելափակող ռեժիմ.* ղեկավարումը կանչող պրոցեսին է վերադառնում անմիջապես (այսինքն՝ պրոցեսն արգելափակվում է նախքան օպերացիայի ավարտը) և փաստացի փոխանցում-ընդունումը տեղի է ունենում ֆոնային ռեժիմում:

Գոյություն ունեն կոլեկտիվ կոմունիկացիաների մի քանի տեսակներ.

Broadcast. մեկը՝ բոլորին: Հաղորդագրության լայնամասշտաբ փոխանցում է, երբ պրոցեսներից մեկի մի բլոկը (տվյալների բլոկը - ՏԲ) հաղորդվում է MPI_Bcast(buffer, count, datatype, source, comm) խմբի բոլոր պրոցեսներին, որտեղ buffer-ը տվյալների բուֆերի սկզբնական հասցեն է, count-ը՝ բուֆերի տարրերի քանակը, datatype-ը՝ տարրերի տիպը, source-ը՝ փոխանցող պրոցեսի համարը, comm-ը՝ կոմունիկատորը:

Scatter. մեկը՝ յուրաքանչյուրին: ՏԲ-ի բաժանումը MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, source, comm) խմբի մի պրոցեսից մյուս պրոցեսներին, որտեղ sendbuf, sendcount, sendtype-ը դրանք սկզբնական հասցեն, բուֆերի տարրերի քանակն ու տիպն են, recvbuf, recvcount, recvtype-ը՝ նույնը տվյալների հավաքման բուֆերի համարը, source-ը՝ տվյալների հավաքման պրոցեսի համարը, comm-ը՝ կոմունիկատորը:

Gather. Յուրաքանչյուրը՝ մեկին: ՏԲ-ի հավաքում բոլոր պրոցեսներից՝ MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, dest, comm) խմբի մի պրոցեսի մեջ: Խմբի յուրաքանչյուր պրոցես (ներառյալ առանցքայինը) իր բուֆերի պարունակությունն ուղարկում է առանցքային պրոցեսին:



Նկ.1. Կոլեկտիվ փոխանցումներ

Allgather: բոլորը ՝ յուրաքանչյուրին: ՏԲ-ի հավաքում խմբի բոլոր պրոցեսներից և արդյունքի տրամադրում խմբի բոլոր պրոցեսներին:

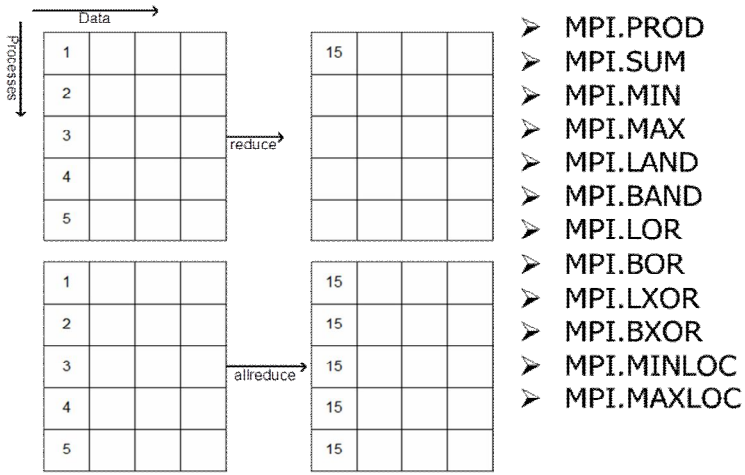
MPI_Allgather (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, source_dest, comm):

Alltoall: բոլորը՝ բոլորին: ՏԲ-ի հավաքում/ուղարկում բոլոր պրոցեսներից դեպի բոլոր պրոցեսներ:

MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm).

MPI_Barrier պրոցեսների խմբի արգելքներով սինքրոնացում: Այս պրոցեդուրան կանչելու ժամանակ ղեկավարումը կանչող ծրագրին չի վերադառնում այնքան ժամանակ, քանի դեռ խմբի բոլոր պրոցեսները չեն իրականացրել MPI_Barrier (comm) կանչերը, որտեղ comm-ը կոմունիկատորն է:

Ռեդուկցիայի գլոբալ (բոլոր պրոցեսներից տվյալներ հավաքող) օպերացիաներ: Գումարում, մաքսիմումի և մինիմումի, ինչպես նաև օգտագործողի կողմից որոշվող գործողության հաշվարկ. արդյունքը փոխանցվում է մեկ կամ բոլոր պրոցեսներին (MPI_Reduce, MPI_Allreduce, MPI_Reduce_Scatter, MPI_Scan):



Նկ. 2. Ռեդուկցիայի օպերացիաներ

Լաբորատոր աշխատանք 1.

Ծանոթացում հաշվողական LINUX-կլաստերի ճարտարապետությանը, ադմինիստրացման հիմունքներին և օգտագործողների աշխատանքի սկզբունքներին:

Աշխատանքի նպատակն է՝ ծանոթանալ կլաստերային հաշվողական համակարգի ճարտարապետությանը, աշխատանքի սկզբունքներին և օգտագործողի կողմից հաշվողական կլաստերի ռեսուրսներին հեռավար հասանելությանը:

Տեսական մաս: Վերջին տարիներին մեծ տարածում են ստացել կլաստերային տեխնոլոգիաները: Կլաստերի գրավչությունը որոշվում է նախ՝ յուրահատուկ ճարտարապետության ստեղծման հնարավորությամբ, որը կարող է օժտված լինել բավական մեծ արտադրողականությամբ, ծրագրային ապահովման և սարքավորման հուսալիության կայունությամբ, միևնույն ժամանակ ստանդարտ բաղադրիչներից հեշտությամբ լրացվող և մոդեռնացվող համապիտանի միջոցներով:

Կլաստերը միմյանց հետ փոխկապակցված պրոցեսորների հավաքածու է, որն օգտագործվում է որպես մեկ միասնական ռեսուրս: Կլաստերի հանգույցները (հանգույցը կլաստերի կազմության մեջ մտնող մեկ քոմփյութերն է) միմյանց հետ միացվում են սովորական ցանցային կապերի կամ ոչ ստանդարտ տեխնոլոգիաների օգնությամբ: Ներկլաստերային կապերը հնարավորություն են տալիս հանգույցներին հաղորդակցվել միմյանց հետ՝ անկախ արտաքին ցանցի յուրահատկությունից:

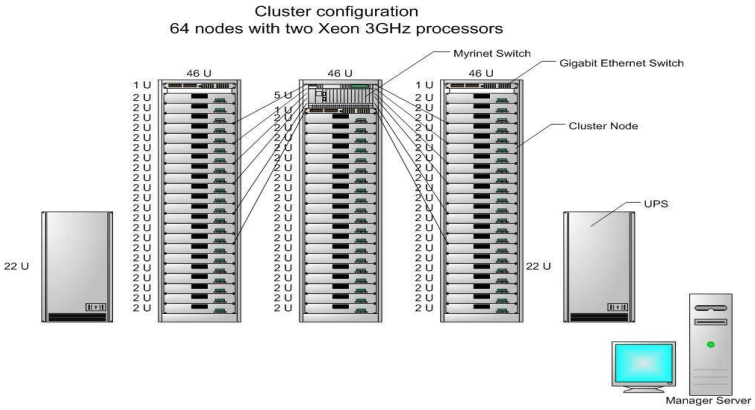
Կլաստերի հանգույց կարող է լինել ինչպես մեկ պրոցեսորային, այնպես էլ բազմապրոցեսորային քոմփյութերը: Ի դեպ, կլաստերի մեջ մտնող քոմփյութերները կարող են ունենալ

տարբեր կոնֆիգուրացիա (պրոցեսների քանակ, օպերատիվ հիշողություն, սկավառակասարքի թողունակություն):

Կլաստերները օժտված են հասանելիության բարձր աստիճանով, քանի որ դրանցում բացակայում են ընդհանուր օպերացիոն համակարգը և համատեղ օգտագործվող օպերատիվ հիշողությունը: Գործնականորեն հանգույցների քանակի ավելացման հնարավորությունը և ընդհանուր օպերացիոն համակարգի բացակայությունը կլաստերային տեխնոլոգիաները դարձնում են լավ մասշտաբավորված: Կլաստերում կարող են կատարվել մի քանի առանձին խնդիրներ, բայց առանձին խնդրի մասշտաբայնության համար պահանջվում է, որպեսզի նրա հանգույցները փոխհաղորդակցվեն միմյանց հետ հաղորդագրությունների փոխանցման միջոցով: Կլաստերի աշխատանքի ապահովման համար օգտագործվում են հատուկ ծրագրային և ապարատային միջոցներ:

Այժմ դիտարկենք կլաստերային համակարգերին բնորոշ ճարտարապետությունը և ծրագրային միջավայրը ԱրմԿլաստեր [19, 20, 21, 22] բարձր արտադրողականությամբ համակարգի օրինակի հիման վրա:

Հաշվողական հանգույցներ: Կլաստերը բաղկացած է 64 հաշվողական հանգույցից՝ կառուցված Intel SE7501CW2 մայրական սարքի հենքի վրա և հավաքված երեք կաղապարի մեջ (նկ. 3):

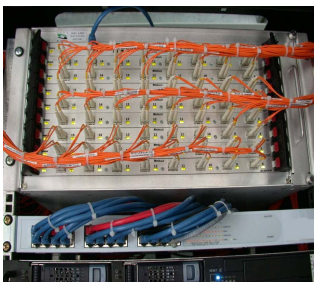


Նկ. 3. Կլաստերի կառուցվածքը

Հաշվողական հագույցները(նկ. 4) հավաքված են 2U չափսի մոդուլների մեջ և պարունակում են՝

- 2 Intel Xeon DP պրոցեսորներ՝ 3000MHz արագագործությամբ և 512Kb քեշ հիշողությամբ,
- Օպերատիվ հիշողություն՝ 4Gb,
- Հիմնական IDE հիշասարք՝ 80GB,
- 2 ցանցային ինտերֆեյս՝ Gigabit Ethernet, Fast Ethernet,
- Յուրաքանչյուր հաշվողական հագույցում տեղադրված է նաև Myrinet 2000

Նկ.4. Կլաստերի հաշվողական հանգույցները



Նկ.5.Հաղորդակցման միջավայրերը

Միջհանգուցային կապեր: Հանգույցների միջև միջհանգուցային կապերը իրականացված են երկու հաղորդակցման միջավայրի օգնությամբ (նկ. 5): Հաշվողական հաղորդակցման միջավայրը նախատեսված է զուգահեռ ծրագրի կատարման ընթացքում առաջացող միջհանգուցային փոխազդեցության, իսկ ղեկավարող հաղորդակցման միջավայրը՝ օպերացիոն համակարգի ցանցային ծառայությունների ապահովման, կլաստերի վերլուծության ենթարկման և ղեկավարման համակարգերի համար: Հաղորդակցման միջավայրերի այսպիսի բաժանումը հնարավորություն է տալիս բացառել ղեկավարող և հաշվողական տվյալների հոսքերի փոխադարձ ազդեցությունը: Հաշվողական հաղորդակցման միջավայրը իրականացված է ցանցային Myrinet 2000, իսկ ղեկավարող հաղորդակցման միջավայրը՝ Gigabit Ethernet տեխնոլոգիաների օգնությամբ: Սովորական ցանցային տեխնոլոգիաներով նույնպես կարելի է կառուցել կլաստերներ, սակայն դրանք չեն ապահովի բարձր արագագործություն և հասանելիություն: Բնութագրիչները, որոնք Myrinet ցանցը տարբերում են այլ ցանցերից, հետևյալն են՝

- 2+2 գիգաբիթ/վայրկյան թողունակություն,
- ցածր լատենտայնություն (Latency),
- ճկուն բաշխիչ սարքերի բազմազանություն, որը հնարավորություն է տալիս կլաստերի հանգույցների թիվը հասցնել մի քանի հազարի:

Մուտքի ապահովման հանգույց: Մուտքի ապահովման հանգույցը լուծում է մի շարք խնդիրներ և հանդիսանում է՝

- Կլաստերի ռեսուրսներին մուտքի ապահովման հանգույց;
- Ֆայլային սերվեր և հիշողության շտեմարան;

- Կլաստերի ղեկավարման և վերլուծության ենթարկման սերվեր;
- Կլաստերի էլեկտրական սնուցման ղեկավարման սերվեր:

ԱրմԿլաստեր համակարգում մուտքի ապահովման հանգույցը կառուցված է Intel SE7501CW2 սերվերային մայրական սարքի հենքի վրա և պարունակում է՝

- Intel Xeon պրոցեսոր՝ 2667MHz արագագործությամբ և 512Kb քեշ հիշողությամբ;
- Օպերատիվ հիշողություն՝ 2Gb;
- Հիշողության շտեմարան՝ 6 SCSI հիշասարքերից՝ յուրաքանչյուրը 36GB ծավալով;
- 2 ցանցային ինտերֆեյս՝ Gigabit Ethernet, Fast Ethernet;
- Տվյալների գրանցման սարք՝ Sony DVD ROM+CD-RW:

Էլեկտրական սնուցման և սառեցման համակարգ: Էլեկտրական սնուցման անխափան աշխատանքի ապահովման համար օգտագործվում են երկու անխափան սնուցման APS Symmera 16kVA Power Array սարքեր: APS Power Array-ը բաղկացած է 4KVA հոսանքի սնուցման (Power Module), մարտկոցային (Battery Module) և ղեկավարման մոդուլներից: Լրացուցիչ մարտկոցները ավելացնում են աշխատանքի տևողությունը:

Աշխատանքի ընթացքում, հատկապես երբ հանգույցները ծանրաբեռնված են աշխատանքով, կլաստերը արտանետում է մեծ քանակի ջերմություն, որը էապես նվազեցնում է համակարգի արտադրողականությունը: Անհրաժեշտ ջերմաստիճանը ապահովելու համար տեղադրվում են սառեցման համակարգեր:

Համակարգային ծրագրային ապահովում: Կլաստերի անբաժանելի մասը նրա համակարգային ծրագրային ապահովումն է, որն ապահովում է կլաստերի արդյունավետ շահագործումը որպես մեկ ամբողջություն և հնարավորություն է տալիս լուծել հետևյալ խնդիրները՝

- MPI զուգահեռ ծրագրերի իրականացում,
- Կլաստերի ղեկավարում,
- Կլաստերի ռեսուրսների և համակարգի վերլուծություն,
- Անվտանգ հեռահար մուտքի ապահովում,
- Ռեսուրսների արդյունավետ բաշխում և համամասնական ծանրաբեռնվածություն:

ԱրմԿալստեր համակարգում ծրագրային ապահովումը կառուցված է Enterprise Linux օպերացիոն համակարգի հենքի վրա և իրականացված է OSCAR ծրագրային փաթեթի օգնությամբ:

Համակարգային ծրագրային ապահովումը պայմանականորեն բաժանվում է մի քանի մասերի:

1. **Զուգահեռ միջավայր, կոմպիլյատորներ, գրադարաններ:** MPI զուգահեռ ծրագրերի կատարումն ապահովում է կլաստերի վրա տեղադրված զուգահեռ միջավայրը, որն ապահովվում է LAM և MPICH ծրագրային փաթեթների օգնությամբ:

LAM (Local Area Multicomputer) - MPI-ի ծրագրային միջավայր է, որը մշակվել է Նոտր-Դամ համալսարանի կողմից: LAM/MPI նախատեսված է կլաստերների հետ աշխատանքի համար, որն ընդգրկում է ինչպես բեռնման և ղեկավարման հնարավորությունները, այնպես էլ մի կլիենտից մյուսին արագ աշխատանքը կազմակերպող հաղորդակցությունների փոխանցման արձանագրությունը: LAM/MPI տվյալների փոխանցման համար կարող է օգտագործել TCP/IP արձանագրությունը կամ

տարաբաշխված հիշողությունը: Այս փաթեթի հիմնական առավելությունը հարուստ կարգավորիչի գոյությունն է:

MPICH (MPI CHameleon) - MPI-ի ծրագրային միջավայր է, որը մշակվել է Արգոնի ազգային լաբորատորիայում (ԱՄՆ): MPICH-ը հնարավորություն է տալիս օգտագործել սոկետների մեխանիզմը, ինչպես նաև TCP/IP արձանագրությունը: Այս միջավայրը իր վրա է վերցնում բոլոր տիպերի ձևափոխութ-յու-ն-ները, ներքին հերթերի հետ աշխատանքը և կոլեկտիվ գործողությունների կազմակերպումը: MPICH-ի վերջին տարբերակները ապահովում են հաղորդակցությունների բազմաարձանագրային փոխանցումներ: MPICH-ը տարածվում է սկզբնական կոդերով և աշխատեցվում է տեղադրման տեղում՝ հաշվի առնելով կոնկրետ պլատֆորմի և հաղորդակցման միջավայրի հատկությունները: Տեղադրումից հետո փաթեթն աշխատում է իր գրադարանների հետ՝ կանչելով C/C++ և FORTRAN 77/90 կոմպիլյատորները:

Կլաստերների վրա տեղադրվում են նաև բազմաթիվ ծրագրային փաթեթներ, որոնք էապես հեշտացնում են օգտագործողների աշխատանքը: Մասնավորապես, ԱրմԿլաստեր վրա տեղադրված են՝ ATLAS (Automatically Tuned Linear Algebra Software), FFTW (Fastest Fourier Transform in the West), LAPACK (Linear Algebra PACKage), SCALAPACK (Scalable Lapack) և BLAS (Basic Linear Algebra Subprograms) գրադարանները:

2. **Կլաստերի ղեկավարումը:** Կլաստերի ղեկավարումը իրականացված է հետևյալ ծրագրային գործիքների օգնությամբ՝

- RPOWER՝ կլաստերի և նրա հանգույցների էլեկտրական սնուցման ղեկավարում (անջատում, բեռնավորում, վերաբեռնավորում),
- C3-4՝ կլաստերի միջավայրում խմբային գործողությունների ապահովում,

- ENV-SWITCHER՝ զուգահեռ MPI ծրագրերի միջավայրի փոփոխություն,
- OPIUM՝ օգտագործողների հետ աշխատանքի ապահովում:

Համակարգի վերլուծության ենթահամակարգեր: Առաջարկված գործիքները ամբողջապես ապահովում են համակարգի վերլուծությունը.

- HMON՝ կլաստերի ֆիզիկական մասի վերլուծության ենթարկման համակարգ. վերլուծության են ենթարկվում ֆիզիկական մասի այն պարամետրերը, որոնք ազդում են կլաստերի աշխատունակության վրա (պրոցեսորների, մայրական սարքերի, հիշասարքերի և այլ համակարգային հանգույցների ջերմաստիճանները, պրոցեսորների օդափոխիչների արագագործությունը, պրոցեսորների ծանրաբեռնվածությունը և այլն):
- MyrinetMON՝ Myrinet 2000 հաշվողական ցանցի հեռավար վերլուծության ենթարկում. հնարավորություն է տալիս հետևել երթուղավորիչի պորտերի պարամետրերին, ցանցային ինտերֆեյսների աշխատանքին, ինչպես նաև հանգույցների միջև ֆիզիկական կապին:
- PowerChute՝ էլեկտրականության անխափան սնուցման սարքերի վերլուծություն. վթարային իրավիճակների դեպքում ապահովում է կլաստերի նորմալ անջատումը:
- Բոլոր գործիքներն ապահովում են նաև անվտանգ WEB ինտերֆեյս:

Կլաստերի ռեսուրսների վերլուծություն: Կլաստերի ռեսուրսների վերլուծության համար գոյություն ունեն բազմաթիվ ծրագրային փաթեթներ, որոնցից առավել կիրառելիներից է Ganglia ծրագրային փաթեթը, որն ստեղծվել է Կալիֆորնիայի համալսարանի կողմից: Ganglia-ն տարաբաշխված վերլուծության ենթարկման համակարգ է՝ բարձր արտադրողականութ-

յամբ հաշվողական համակարգերի համար: Տվյալների ներկայացման համար օգտագործվում է XML տեխնոլոգիան: Ծրագրային փաթեթը կլաստերի հանգույցներից շատ քիչ ռեսուրսներ է խլում և չի ազդում համակարգի ծանրաբեռնվածության վրա: Ծրագրային փաթեթն օժտված է WEB ինտերֆեյսով և հնարավորություն է տալիս հետևել կլաստերի ռեսուրսների ընդհանուր վիճակին: Այն պարունակում է մի քանի WEB էջեր, որը ցույց է տալիս կլաստերի ռեսուրսների մասին տարբեր մանրամասնությամբ ինֆորմացիա:

Կլաստերի մուտքի ապահովումը: Կլաստերային համակարգերը պետք է անվտանգ մուտք ապահովեն դեպի կլաստերի ռեսուրսները: ԱրմԿլաստերի անվտանգ աշխատանքը ապահովելու համար տեղադրված է երկու լրացուցիչ մուտքային հանգույց, որոնք միացված են կլաստերի գլխավոր հանգույցին: Օգտագործողները գրանցվում են այդ հանգույցների վրա, և նրանց աշխատանքի ապահովումը իրականացվում է SSH (Secure Shell) արձանագրության (պրոտոկոլ) միջոցով: Առկա է նաև ճկուն և հարուստ Web ինտերֆեյս, որը հնարավորություն է տալիս աշխատեցնել օգտագործողների խնդիրները անմիջապես իրենց լոկալ քոմփյութերներից:

Խնդիրների փաթեթային մշակման համակարգ: Կլաստերի արդյունավետ օգտագործումը ենթադրում է այնպիսի միջավայրի օգտագործում, որը հնարավորություն կտա օգտագործողների կողմից հարցումների ժամանակ լուծել ռեսուրսների տրամադրման հետ կապված խնդիրները, խնդիրների ավարտից հետո ռեսուրսների ազատումը, տարբեր օգտագործողների համար գերակայությունների համակարգի ապահովումը, ինչպես նաև հաշվողական համակարգի հավասարաչափ ծանրաբեռնվածության ապահովումը: Դիտարկված խնդիրները ԱրմԿլաստեր համակարգում լուծվել են՝ նրա վրա տե-

ղադրելով խնդիրների փաթեթային մշակման PBS համակարգը և MAUI պլանավորողը, ինչպես նաև հատուկ մշակած ծրագրային ապահովումը[22, 27]:

Համակարգի թեստավորում: Կլաստերային համակարգի հիմնական բնութագրիչները՝ արդյունավետությունը և արտադրողականությունը, որոշելու համար իրականացվում են թեստավորումներ: Արտադրողականությունը սահող կետով թվերի հետ գործողությունների քանակն է, որը չափվում է GFlops-երով, իսկ արդյունավետությունը՝ ստացված առավելագույն արտադրողականության և տեսական արտադրողականության հարաբերություն: Տեսական արտադրողականությունը որոշվում է Rpic=2*P*H բանաձևով, որտեղ P-ն կլաստերի հաշվողական հանգույցների քանակն է, իսկ H-ը՝ հաշվողական հանգույցի արագագործությունը: Հետևաբար ԱրմԿլաստերի տեսական արտադրողականությունը կազմում է Rpic=783.36Gflops (2*128*3.06): Առավելագույն արտադրողականությունը գտնելու համար աշխարհում ընդունված է օգտագործել HPL (High Performance Linpack) թեստավորման ծրագրային փաթեթը, որը հիմնօրինակն է: Ինչքան թեստի արդյունքը մոտ է տեսական արտադրողականությանը, այնքան կլաստերը համարվում է լավ նախագծված: Նշենք, որ HPL-ը խնդիր է, որը գեներացնում է գծային հավասարումների համակարգ և լուծում է այն Գաուսի մեթոդով: Թեստի օգտագործման համար անհրաժեշտ է MPI-ի գոյությունը, ինչպես նաև գծային հանրահաշվի բազային պրոցեդուրաների BLAS գրադարանը: Թեստի արդյունքում Արմկլաստեր համակարգի առավելագույն արտադրողականութունը ստացվել է 483.6Gfl-ops, որը կազմում է տեսականի 62%:

ԱրմԿլաստերի միջավայրում կատարվել են նաև այլ թեստավորումներ (Pallas MPI Benchmark, Netperf Benchmark,

LMbench Benchmark)՝ պրոցեսորների, հիշասարքերի, օպերատիվ հիշողությունների, կապուղիների (Myrinet 2000, Gigabit Ethernet), ծրագրային միջավայրերի (MPI) ծանրաբեռնվածության իրական բնութագրիչները ստանալու համար [19]:

Զուգահեռ ծրագրերի ստեղծումը և աշխատեցումը: Զուգահեռ ծրագրի (մասնավորապես program.c և program.f) կոմպիլյացիան համապատասխանաբար C և Fortran լեզուներում իրականացվում է օգտագործողի կողմից համապատասխան հրամանով (կատարվելիք ֆայլը կանվանենք program առանց ընդլայնման).

- mpicc -o program program.c
- mpif77 -o program program.f

Զուգահեռ ծրագրի կատարումը տեղի է ունենում *mpirun* փաթեթային ֆայլի (սկրիպտի) միջոցով հետևյալ կերպ.

mpirun –np N [mpi_args] program [command_line_args …]

որտեղ

- –np N –ը պարտադիր բանալի է: Զուգահեռ ծրագիրը կկազմվի N հատ խնդիր-պատճեններից, որոնք բեռնվում են program_file ծրագրային ֆայլից (խնդրի օրինակները ստանում են 0-ից մինչև N-1 համարները),
- mpi_args – ը MPI-համակարգի արգումենտներն են (ոչ պարտադիր),
- command_line_args … - հրամանի տողի արգումենտներն են (ոչ պարտադիր), որոնք փոխանցվում են խնդրի յուրաքանչյուր օրինակին:

Նշենք, որ երբ խնդիրն աշխատեցնում ենք N հաշվողական հանգույցների վրա, հաշվողական աշխատանքի առյուծի բաժինը սովորաբար կատարում են N-1 աշխատող SLAVE-հանգույցները: Մեկ ղեկավարող (MASTER)–հանգույց կոորդինացնում է մյուսների աշխատանքը (հաշվարկման համար նախա-

պատրաստում և բաշխում է տվյալները SLAVE-հանգույցներին, հավաքում է տվյալները և իրականացնում է դրանց լրացուցիչ մշակում):

Յուրաքանչյուր հանգույցի վրա ժամանակի տվյալ պահին կատարվում է միայն մեկ խնդիր (պրոցես). եթե պահանջվող քանակի հանգույցներ չկան (անջատված են, ներդրված չէ LINUX կամ եթե հանգույցները տվյալ պահին կատարում են կողմնակի աշխատանք), ապա խնդիրը սպասում է հերթի մեջ՝ մինչև ազատվեն պահանջվող քանակի հանգույցներ:

Կլաստերի ադմինիստրատորը հնարավորություն ունի աշխատելու հենց HOST-մեքենայից կամ հեռավար ռեժիմում: Երկրորդ դեպքում օգտագործվում են SSH-կլիենտի ծրագրերը: Հեռահար ռեժիմում աշխատելիս օգտագործողը պետք է կատարի հետևյալ գործողությունները.

- Ֆայլերի երկկողմանի փոխանակում իր (կլիենտի) մեքենայի և կլաստերի ղեկավարող մեքենայի միջև,
- Ֆայլային համակարգի ղեկավարում (ֆայլերի և կատալոգների ստեղծում/ փոփոխում/ջնջում),
- Խնդիրների ղեկավարում (խնդրի աշխատեցում և կանգնեցում, վիճակի մասին տեղեկանքի ստացում):

Վերը նշված հրամանները կարելի է իրականացնել կամ անմիջականորեն հենց հրամանային տողից, կամ օգտագործելով ֆայլ-մենեջերներ: Մասնավորապես, հարմար գործիք է Midnight Commander ծրագիրը, որը կարելի է բեռնել հրամանային տողից՝ աշխատացնելով mc –ac հրամանը: Կլաստերի գլխավոր մեքենային հասանելիությունն ապահովելու համար հաճախ օգտագործվում են հետևյալ կլիենտային ծրագրերը.

- PSCP (PuTTY Secure CoPy client, ղեկավարվում է հրամանային տողով). ապահովում է տվյալների անվտանգ փոխանցում օգտագործողի և սերվերի միջև:

- WinSCP (Windows Secure CoPy, PSCP պատուհանային տարբերակը), հնարավորություն է ընձեռում ընտրել Norton Commander կամ MS Explorer-ֆայլային մենեջերը.
- PuTTY (ապահովում է Telnet և SSH արձանագրությունները).
- SSH-կլիենտներ պատուհանային ռեժիմով (օրինակ՝ SSH Secury Shell՝ իր մեջ ներառված SSH Secure File Transfer Client մոդուլով).

Նշենք, որ բոլոր կլիենտային ծրագրերը պահանջում են HOST-մեքենայի IP կամ դոմեն հասցեի ներմուծում, որից հետո ներմուծվում են օգտագործողի login և password, որպեսզի հնարավոր լինի կապ հաստատել կլաստերի գլխավոր հանգույցի հետ: Ծրագրերը կարող են պատրաստվել նաև օգտագործողի մեքենայի վրա (մասնավորապես աշխատող Windows օպերացիոն համակարգում) և հետագայում տեղափոխվեն կլաստերի վրա: Խնդիրների փաթեթային մշակման ժամանակ ենթադրվում է, որ ծրագիրը չպետք է լինի ինտերակտիվ: Հետևաբար, տվյալները ծրագրին պետք է տրվեն կամ հրամանային տողի միջոցով, կամ ֆայլի միջոցով:

Աշխատանքների կատարման հաջորդականությունը: Ուսանողը Windows կլիենտային մեքենայից մտնում է կլաստերի ղեկավարող հանգույց, օգտվելով վերը նշված SSH-կլիենտներից, և իրականացնում գրանցում (login և password-ը տրվում են դասախոսի կողմից):

- Օգտագործողը ծանոթանում է ղեկավարող հանգույցի սկավառակային հիշողության ֆայլային համակարգի հետ (սովորաբար օգտագործողի ընթացիկ կատալոգը է /home/quest-ն է), վարժվում է կլիենտի մեքենայի և ղեկավարող հանգույցի միջև ֆայլերի փոխանակմանը (հարմար է /home/quest –ում

ստեղծել յուրօրինակ անունով ենթակատալոգ), ֆայլերի վերանվանմանը և ֆայլերի հետ կապված այլ օպերացիաներին:

- Ուսանողը ծանոթանում է կլաստերային համակարգի ղեկավարման, մոնիտորինգի և խնդիրների փաթեթային մշակման համակարգերի հնարավորություններին:
- Ուսանողը կոմպիլացնում է որևէ հաջորդական կիրառական ծրագիր, աշխատեցնում այն՝ վերլուծության ենթարկելով ելքային տվյալները:
- Մուտքային ֆայլի խմբագրումը կարող է իրականացվել օգտագործողի միջավայրի մեջ ներկառուցված որևէ խմբագրիչի միջոցով:
- Օգտագործելով վերը նշված կանոնները՝ ուսանողը կոմպիլացնում է զուգահեռ ծրագիրը (օրինակ՝ cpi.c, որը ընդգրկված է MPICH ծրագրային փաթեթի կազմում), աշխատեցնում այն, վերլուծության ենթարկում ելքային տվյալները:
- Կոմպիլացված խնդիրն աշխատեցնում է խնդիրների փաթեթային համակարգի միջոցով:

Հարցեր ինքնաստուգման համար

1. Ի՞նչ համակարգային ծրագրային ապահովում է օգտագործվում կլաստերային համակարգերում:
2. Ի՞նչ է խնդիրների պլանավորման համակարգը:
3. Ինչպե՞ս է իրականացվում կլաստերային համակարգերի ղեկավարումը և նրա ռեսուրսների վերլուծությունը:
4. Ինչպե՞ս է իրականացվում զուգահեռ ծրագրերի ստեղծումը և աշխատեցումը:
5. Ինչպիսի՞ ցանցեր են անհրաժեշտ հաշվողական կլաստերների ստեղծման ժամանակ:

Լաբորատոր աշխատանք 2.

Պրոցեսների կյանքի ցիկլը և նրանց միջև տվյալների պարզագույն փոխանակումը: Փակուղային իրավիճակներ:

Աշխատանքի նպատակն է՝ տալ պարզագույն MPI-ծրագրերի կոմպիլյացիայի և աշխատեցման վերաբերյալ գործնական գիտելիքներ, ինչպես նաև ապացուցել զուգահեռ ծրագրերի ճյուղերի սինքրոնացման անհրաժեշտությունը:

Տեսական մաս: Կլաստերային համակարգերում կատարվող ծրագրային կոդը կոմպիլացվում է գլխավոր մեքենայի վրա, այնուհետև բաշխվում հաշվողական հանգույցներին և յուրաքանչյուրի վրա սկսում կատարվել օպերացիոն համակարգի կողմից: Ծրագրի ճյուղերի աշխատանքների սկիզբը կարող է միաժամանակյա չլինել, նաև պատահականորեն չի կարելի ճշգրիտ որոշել կոնկրետ ճյուղում որևիցե օպերատորի կատարման պահը: Զուգահեռ ծրագրավորման ժամանակ պրոցեսների սինքրոնացման և տվյալների փոխանցման խնդիրների պրակտիկ կիրառման համար դիտարկենք MPI-ի կառուցվածքը և բազային ֆունկցիաները:

MPI ծրագրի կառուցվածքը: Յուրաքանչյուր MPI ծրագիր պարունակում է պրեպրոցեսորի դիրեկտիվ.

```
#include "mpi.h"
```

mpi.h ֆայլը պարունակում է սահմանումներ, մակրոսահմանումներ և ֆունկցիաների նախատիպեր, որոնք անրաժեշտ են MPI ծրագրի կոմպիլյացիայի համար: Նախքան MPI–ի կամայական այլ ֆունկցիաներ կանչելը, զուգահեռ ծրագրում անհրաժեշտ է մեկ անգամ կանչել MPI_Init() ֆունկցիան: Նրա արգումենտները main() ֆունկցիայի argc և argv պարամետրերի վրայի ցուցիչներն են: MPI գրադարանն օգտագործող ծրագրի ավարտին անհրաժեշտ է կանչել MPI_Finalize(): Այս

ֆունկցիան ավարտում է MPI-ի բոլոր անավարտ գործողությունները, մասնավորապես, փոխանցման անվերջ սպասումները: MPI-ի տիպային ծրագիրն ունի հետևյալ կառուցվածքը.

```
#include "mpi.h"
. . .
main(int argc, char** argv) {
. . .
/* MPI –ի ֆունկցիաները չի կարելի կանչել մինչ այդ պահը*/
MPI_Init(&argc, &argv);
. . .
MPI_Finalize();
/* MPI –ի ֆունկցիաները չի կարելի կանչել այդ պահից հետո*/
. . .
} /* main */
```

Պրոցեսի համարի որոշումը: MPI-ը առաջարկում է MPI_Comm_rank() ֆունկցիան, որը վերադարձնում է պրոցեսի ռանգը: Նրա շարահյուսությունը(syntax) հետևյալն է.

```
int MPI_Comm_rank(MPI_Comm comm, int *rank);
```

Առաջին արգումենտը կոմունիկատոր է: Ըստ էության, կոմունիկատորը պրոցեսների հավաքածու է, և այդ պրոցեսները կարող են միմյանց ուղարկել հաղորդագրություններ: Ոչ մեծ ծրագրերի համար միակ անհրաժեշտ կոմունիկատորը MPI_COMM_WORLD-ն է: Այն MPI-ում նախասահմանված է և պարունակում է ծրագրի սկսվելու պահից հետո կատարվող բոլոր պրոցեսները: Պրոցեսի ռանգը վերադարձվում է երկրորդ՝ rank արգումենտում:

Ծրագրային շատ կառուցվածքներ կախված են ծրագիրը կատարող պրոցեսների ընդհանուր թվից: Դրա համար MPI-ը պարունակում է MPI_Comm_size() ֆունկցիան, որը որոշում է

դրանց քանակությունը: Այդ ֆունկցիայի շարահյուսությունը հետևյալն է.

```
int MPI_Comm_size(MPI_Comm comm, int *size);
```

comm կոմունիկատորում պրոցեսների քանակը վերադարձվում է size փոփոխականի մեջ:

Հաղորդագրության կառուցվածքը: Ծրագրում հաղորդագրությունների փաստացի փոխանցումը կատարվում է MPI_Send() և MPI_Recv() ֆունկցիաների միջոցով: Առաջին ֆունկցիան ուղարկում է հաղորդագրություն որոշակի պրոցեսի: Երկրորդը ստանում է հաղորդագրություն որևէ պրոցեսից: Այս ֆունկցիաները MPI-ում հաղորդագրությունների փոխանցման ամենահիմնական հրամաններն են: Որպեսզի հաղորդագրությունը հաջող փոխանցվի, համակարգը պետք է <ավելացնի> ևս մի փոքր ինֆորմացիա փոխանցվող տվյալներին: Այդ լրացուցիչ ինֆորմացիան ձևավորում է հաղորդագրության ծրարը, որը պարունակում է հետևյալ տեղեկությունը.

- ստացողի ռանգը,
- ուղարկողի ռանգը,
- հաղորդագրության տեգը,
- կոմունիկատորը:

Այս դետալները կարող են գործածվել ստացողի կողմից, որպեսզի ճանաչվի ստացվող հաղորդագրությունը: Տարբեր պրոցեսներից ստացված հաղորդագրությունները տարբերելու համար օգտագործվում է source արգումենտը: Տեգը օգտագործողի կողմից նշված int արժեքն է, որը նախատեսված է մի պրոցեսից ստացվող հաղորդագրությունները տարբերելու համար: Ենթադրենք A պրոցեսը B պրոցեսին ուղարկում է երկու հաղորդագրություն: Երկու հաղորդագրություններն էլ պարունակում են float տիպի մի արժեք: Արժեքներց մեկը պետք է

օգտագործվի հաշվարկներում, իսկ մյուսը պետք է դուրս բերվի էկրանի վրա: Որոշելու համար, թե դրանից որն է առաջինը, A-ն օգտագործում է այդ հաղորդագրությունների տարբեր տեգեր: Եթե B-ն օգտագործում է այդ նույն տեգերը ընդունման ժամանակ, ապա նա կիմանա, թե դրանց հետ ինչպես վարվի: MPI-ը ենթադրում է, որ որպես տեգեր կարող է օգտագործել 0 – 32767 միջակայքում ընկած ամբողջ թվերը: Ռեալիզացիաների մեծ մասը թույլ է տալիս օգտագործել ավելի շատ արժեքներ:

Հաղորդագրությունների փոխանցման ֆունկցիաներ: Հաղորդագրությունների փոխանցումը կարելի է տրոհել ստանդարտ և ոչ ստանդարտ ռեժիմների: Ստանդարտ ռեժիմի արգելափակող հաղորդագրություն փոխանցող ֆունկցիան ունի հետևյալ տեսքը.

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

- message - հաղորդագրության փոխանցման բուֆերի սկզբնական հասցեն
- count - հաղորդագրության մեջ ուղարկվող տարրերի քանակը (ոչ բայթերի քանակը)
- datatype - փոխանցվող տարրերի տիպը
- dest - ստացող պրոցեսի համարը
- msgtag - հաղորդագրության իդենտիֆիկատոր կամ տեգ (0÷32767, ընտրում է օգտագործողը)
- comm - խմբի իդենտիֆիկատոր (լռելյայն ստեղծվող կոմունիկատորի համար՝ MPI_COMM_WORLD)

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

- message - հաղորդագրություն ընդունող բուֆերի սկզբնական հասցեն (վերադարձվող արժեքը)

- count - ընդունվող հաղորդագրության մեջ տարրերի առավելագույն քանակը
- datatype - ընդունվող հաղորդագրության տարրերի տիպը
- source - ուղարկող պրոցեսի համարը
- msgtag - ստացվող հաղորդագրության իդենտիֆիկատորը (տեգը)
- comm - խմբի իդենտիֆիկատոր
- status - ստացված հաղորդագրության պարամետրերը (վերադարձվող արժեքը)

Հաղորդագրությունը պահվում է հիշողության այն բլոկում, որտեղ ցույց է տալիս message արգումենտը: Հաջորդ երկու՝ count և datatype արգումենտները հնարավորություն են տալիս համակարգին որոշել հաղորդագրության վերջը. այն պարունակում է count քանակի հաղորդագրություն, որոնցից յուրաքանչյուրը datatype տիպի տվյալ է: Այդ տվյալները C-ի ներդրված տիպեր չեն, չնայած՝ նախապես որոշված տիպերի մեծ մասը համապատասխանում է C-ի տիպերին: MPI-ի նախապես որոշված տիպերը և C-ի համապատասխան տիպերը բերված են աղյուսակ 1-ում:

Աղյուսակ 1.

MPI և C տիպերի համապատասխանությունները.

MPI տիպ	C-ի համապատասխան տիպ
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	-

Վերջին երկու՝ MPI_BYTE և MPI_PACKED տիպերը չեն համապատասխանում C-ի ստանդարտ տիպերին: MPI_BYTE տիպն օգտագործվում է այն ժամանակ, եթե համակարգը չպետք է կատարի ներկայացված տարբեր տվյալների միջև ձևափոխություններ (օրինակ, աշխատող կայանների հետերոգեն ցանցում, ուր գործածվում են տվյալների տարբեր ներկայացման ձևեր): MPI տիպերի յուրահատկությունն այն է, որ դրանք թույլ են տալիս ծրագրերին հետերոգեն ճարտարապետությունների վրա միատեսակ փոխգործակցել՝ կատարելով տիպերի անհրաժեշտ ձևափոխությունները օգտագործողի համար թափանցիկ ձևով:

Հարկ է նշել, որ ստացման բուֆերին հատկացված հիշողության չափը կարող է ճշտությամբ չհամապատասխանել ստացվող հաղորդագրության չափսին: Օրինակ, աշխատանքի ժամանակ 1-ին պրոցեսի ուղարկած հաղորդագրության չափսը strlen (message +1) = 28 սիմվոլ է, իսկ 0-րդ պրոցեսը այդ հաղորդագրությունն ընդունում է մի բուֆերի մեջ, որի չափսը հավասար է 100 սիմվոլի: Ամեն դեպքում ստացող պրոցեսը չի կարող նախապես իմանալ իրեն ուղարկվող հաղորդագրութ-

յան ճշգրիտ չափսը: Այդ իսկ պատճառով էլ MPI-ը հնարավորություն է տալիս ստանալ այնքան հաղորդագրություն, որքան տեղ կա իրեն հատկացված հիշողության մեջ: Եթե հիշողության այդ չափսը բավարար չի լինում, առաջանում է հիշողության տիրույթից դուրս գալու սխալ:

Dest և source արգումենտները համապատասխանում են ստացող և ուղարկող պրոցեսների ռանգին: MPI-ը հնարավորություն է տալիս source-ին ընդունել նախապես որոշված MPI_ANY_SOURCE հաստատունի արժեքը: Այն կարող է օգտագործվել այն ժամանակ, եթե պրոցեսը պատրաստ է ընդունել հաղորդագրություն կամայական այլ պրոցեսից: Dest-ի համար նման հաստատուն չկա:

MPI-ը ունի հաղորդագրությունների տիրույթների բաժանման երկու մեխանիզմ՝ տեգեր և կոմունիկատորներ: Գոյություն ունի խմբային MPI_ANY_TAG տեգը, որը կարող է որոշել հաղորդագրության ցանկացած տեգը:

MPI_Recv()-ի վերջին արգումենտը՝ status-ը, վերադարձնում է մի ինֆորմացիա, որը վերաբերում է փաստացի ստացված տվյալներին: Այն հղվում է երկու դաշտերից բաղկացած գրության վրա: Դաշտերից առաջինը նախատեսված է ուղարկողի համար, մյուսը՝ տեգի: Օրինակ, եթե որպես ուղարկող նշված է MPI_ANY_SOURCE, ապա status-ը կպարունակի հաղորդագրությունն ուղարկող պրոցեսի ռանգը:

Փոխանցման ստանդարտ ռեժիմի օգտագործման ժամանակ ծրագրավորողը պետք է հետևի հետևյալ խորհուրդներին.

- Նա չպետք է հույսը դնի այն բանի վրա, որ փոխանցումը կավարտվի նախքան հաղորդագրության ընդունումը: Եթե փոխանցումը արգելափակող է, ապա որոշ դեպքերում կառաջանա փակուղային իրավիճակ:

- Չի կարելի մտածել, որ փոխանցումը կավարտվի հաղորդագրության ընդունումը սկսելուց հետո: Այս դեպքում հաղորդագրությունների հաջորդականության ընդունման կարգը կարող է խախտվել:
- Պրոցեսները պետք է ընդունեն և մշակեն իրենց հասցեագրված բոլոր հաղորդագրությունները:

Հաղորդագրությունների ուղարկման համար կարելի է օգտագործել նաև MPI_Send() ֆունկցիայի ոչ ստանդարտ ռեժիմի տարբերակները (սինխրոն, բուֆերացված և ուղարկելուն պատրաստվածության ռեժիմ):

- MPI_Bsend-ը հաղորդագրության փոխանցումն է բուֆերացումով: Եթե ուղարկվող հաղորդագրության ընդունումը դեռ չի ինիցիալիզացվել ընդունող պրոցեսի կողմից, ապա հաղորդագրությունը կգրանցվի հատուկ բուֆերի մեջ, և կառաջանա անհապաղ վերադարձ պրոցեդուրայից: Այդ պրոցեդուրայի կատարումը ոչ մի ձևով կախված չի հաղորդագրությունն ընդունող պրոցեդուրայի համապատասխան կանչից: Ամեն դեպքում պրոցեդուրան կարող է վերադարձնել սխալի կոդը, եթե բուֆերում բավարար տեղ չկա (բուֆերացման համար զանգվածի առանձնացման մասին պետք է հոգ տանի օգտագործողը):
- MPI_Ssend-ը հաղորդագրության փոխանցումն է սինքրոնացումով: Այս պրոցեդուրայից ելք կկատարվի միայն այն դեպքում, եթե ուղարկվող հաղորդագրության ընդունումը կինիցիալիզացվի ստացող պրոցեսի կողմից: Այսպիսի փոխանցման ավարտը խոսում է ոչ միայն ուղարկման բուֆերը կրկնակի օգտագործելու հնարավորության, այլ նաև ստացող պրոցեսի կողմից հաղորդագրության ընդունման երաշխավորված լինելու մասին: Հաղորդագրության փոխանցման սինքրոնացման տարբերակը դանդաղեցնում է ծրագրի աշխատանքը,

բայց թույլ է տալիս խուսափել համակարգում չընդունված բուֆերացված հաղորդագրությունների մեծ թվից:

- MPI_Rsend-ը հաղորդագրության փոխանցումն է ըստ պատրաստվածության: Այս պրոցեդուրան կարելի է օգտագործել միայն այն դեպքում, եթե ստացող պրոցեսն արդեն ինիցիալիզացրել է հաղորդագրության ընդունումը: Հակառակ դեպքում պրոցեդուրայի կանչը կլինի սխալ, իսկ կատարման արդյունքը՝ անորոշ: Մինչ MPI_Rsend պրոցեդուրայի կանչը հաղորդագրության ընդունման ինիցիալիզացիան կարելի է երաշխավորել այնպիսի գործողություններով, որոնք իրականացնում են պրոցեսների ակնհայտ կամ ոչ ակնհայտ սինքրոնացում (օրինակ, MPI_Barrier կամ MPI_Ssend): Շատ-շատ իրականացումներում MPI_Rsend պրոցեդուրան նվազեցնում է ուղարկողի և ստացողի միջև արձանագրությունները՝ այդ ձևով նվազեցնելով տվյալների փոխանցման վրա կատարվող ծախսերը:

Սինխրոն փոխանցումը կարող է էականորեն դանդաղ լինել ստանդարտ փոխանցումից: Սակայն այն չի հանգեցնում կոմունիկացիոն ցանցը հաղորդագրություններով գերհագեցնելուն և ապահովում է ծրագրի դետերմինացված կատարում: Այս ռեժիմի օգտագործումը հեշտացնում է նաև զուգահեռ ծրագրի կարգաբերումը(debugging):

Բուֆերիզացված փոխանցումը երաշխավորում է անհապաղ ավարտ, քանի որ հաղորդագրությունը սկզբում պատճենահանվում է սիստեմային բուֆերում, իսկ հետո նոր փոխանցվում է: Նրա թերությունն այն է, որ հարկ է առաջանում առանձնացնել և տրամադրել հատուկ բուֆերներ, որոնք օգտագործում են համակարգի ռեսուրսները:

Փոխանցումն ըստ պատրաստվածության ենթադրում է, որ փոխանցումն ինիցիալիզացվում է հենց այն նույն պահին,

երբ ստացողը կանչում է իրեն համապատասխանող ընդունվող հաղորդագրությունը: Այս ռեժիմով երաշխավորվում է, որ կոմունիկացիոն ցանցում չեն լինի մոլորված հաղորդագրություններ:

MPI_Send/MPI_Recv ֆունկցիաների օգնությամբ պրոցեսների միջև իրականացվում է տվյալների հուսալի (բայց ոչ այնքան արդյունավետ) փոխանցում: Սակայն որոշ դեպքերում (օրինակ, երբ ստացող կողմը սպասում է հազորդագրության, սակայն չգիտի ոչ նրա երկարությունը, ոչ էլ տիպը) ավելի հարմար է օգտագործել MPI_Probe արգելափակող ֆունկցիան, որը թույլ է տալիս նախքան հաղորդագրության՝ օգտագործողի բուֆերում տեղադրվելը, որոշել բնութագրիչները (երաշխավորվում է, որ հաջորդ կանչված MPI_Recv ֆունկցիան կկարդա հենց թեստավորված MPI_Probe հաղորդագրությունը).

int MPI_Probe(int source, int msgtag, MPI_Comm comm, MPI_Status *status);

- source - ուղարկող պրոցեսի (կամ MPI_ANY_SOURCE) համարը
- msgtag - սպասվող հաղորդագրության (կամ MPI_ANY_TAG) իդենտիֆիկատոր
- comm - խմբի իդենտիֆիկատոր
- status - հայտնաբերված հաղորդագրության պարամետրերը (վերադարձվող արժեքը)

Status (MPI_Status տիպի) կառուցվածքում պարունակվում է ինֆորմացիա հաղորդագրության մասին՝ նրա իդենտիֆիկատորը (MPI_TAG դաշտը), ուղարկող պրոցեսի իդենտիֆիկատորը (MPI_SOURCE դաշտը):

Հաղորդագրության փաստացի երկարությունը կարելի է իմանալ հետևյալ կանչերի միջոցով.

```
MPI_Status status;
int count;
MPI_Recv ( ... , MPI_INT, ... , &status );
MPI_Get_count (&status, MPI_INT, &count); /* տարրերի տիպը նույնն է, ինչ որ MPI_Recv-ինը, այժմ count-ում պահվում է ստացված MPI_INT տիպի տարրերի քանակը */
```

MPI-ում նախատեսված են մի խումբ պրոցեդուրաներ՝ տվյալների ասինխրոն փոխանցման համար: Ի տարբերություն արգելափակող պրոցեդուրաների, այս պրոցեդուրաներից վերադարձը տեղի է ունենում անմիջապես կանչից հետո՝ չդադարեցնելով ոչ մի պրոցեսի աշխատանք: Ծրագրի հետագա կատարման ընթացքում միաժամանակ տեղի է ունենում նաև ասինխրոն գործարկված պրոցեդուրայի մշակումը: Այս հնարավորությունը շատ օգտակար է արդյունավետ ծրագրերի ստեղծման համար: Շատ դեպքերում պարտադիր չէ սպասել հաղորդագրության ուղարկման ավարտին, որպեսզի կատարվեն հաջորդող հաշվարկները:

MPI_Send պրոցեդուրայի ասինքրոն անալոգն է MPI_Isend-ը (MPI_Recv-ի համար համապատասխանաբար MPI_Irecv-ն է): MPI_Send պրոցեդուրայի երեք մոդիֆիկացիաներին համապատասխան նախատեսված են MPI_Isend պրոցեդուրայի երեք լրացուցիչ տարբերակ.

- MPI_Ibsend՝ հաղորդագրության՝ առանց արգելափակման փոխանցում՝ բուֆերացումով
- MPI_Issend՝ հաղորդագրության՝ առանց արգելափակման փոխանցում սինքրոնացումով
- MPI_Irsend՝ հաղորդագրության՝ առանց արգելափակման փոխանցում ըստ պատրաստվածության

Շատ հաճախ MPI_Send/MPI_Recv ֆունկցիաները օգտագործվում են միաժամանակ և հենց այդ հերթականությամբ,

որի պատճառով էլ MPI-ում ստեղծված է երկու ֆունկցիա, որոնք տվյալներ են ուղարկում և այլ տվյալներ են ստանում միաժամանակ: Դրանցից մեկը MPI_Sendrecv-ն է (այդ ֆունկցիայի առաջին հինգ պարամետրերը նույնն են, ինչ MPI_Send-ը, իսկ մնացած յոթը նույնն են, ինչ MPI_Recv): Հարկ է նշել, որ

- ինչպես ընդունման, այնպես էլ ուղարկման ժամանակ օգտագործվում է մնույն կոմունիկատորը,
- MPI_Sendrecv-ը տվյալների ընդունման և փոխանցման հերթականությունն ընտրում է ավտոմատ, ընդ որում, այդ դեպքում երաշխավորվում է deadlock-ի բացակայությունը:
- MPI_Sendrecv-ը համատեղելի է MPI_Send-ի և MPI_Recv-ի հետ,

MPI_Sendrecv_replace ֆունկցիան հիմնական կոմունիկատորից բացի, օգտագործում է նաև ստացման-ուղարկման ընդհանուր բուֆերը: MPI_Sendrecv_replace-ը պետք է օգտագործել՝ հաշվի առնելով, որ

- ընդունվող տվյալները պետք է ավելի երկար չլինեն, քան ուղարկվողները,
- ստացվող և ուղարկվող տվյալները պետք է միևնույն տիպի լինեն,
- ստացվող տվյալները գրանցվում են ուղարկվողների փոխարեն,
- MPI_Sendrecv_replace-ը երաշխավորված չի կանչում deadlock.

Փակուղային իրավիճակներ: Ընդունման և փոխանցման արգելափակող պրոցեդուրաների օգտագործումը կապված է հնարավոր առաջ եկող փակուղային իրավիճակների հետ: Ենթադրենք աշխատում են երկու զուգահեռ պրոցես, և նրանք պետք է փոխանակեն տվյալներ: Բնական կլիներ յուրաքանչյուր պրոցեսում սկզբում օգտագործել MPI_Send պրոցեդու-

րան, իսկ հետո՝ MPI_Recv: Բայց հենց դա էլ չպետք է անել, քանի որ մենք նախօրոք չգիտենք, թե ինչպես է իրականացված MPI_Send պրոցեդուրան: Եթե ուղարկող բուֆերի կրկնակի ճիշտ օգտագործման համար մշակվել է այնպիսի սխեմա, որ փոխանցող պրոցեսը սպասում է ընդունող պրոցեսի սկսվելուն, ապա առաջանում է դասական փակուղի: Առաջին պրոցեսը չի կարողանում ավարտել պրոցեդուրան, քանի դեռ երկրորդը չի սկսել հաղորդագրությունների ընդունման պրոցեսը: Իսկ երկրորդ պրոցեսը չի կարողանում սկսել հաղորդագրությունների ընդունումը, քանի որ նույնպիսի պատճառով չի կարողանում ավարտել ուղարկման պրոցեդուրան: Ավելի վատ իրավիճակ է ստեղծվում այն դեպքում, երբ երկու պրոցեսներն էլ սկզբում սկսում են ընդունման արգելափակող MPI_Recv պրոցեդուրան,և հետո միայն սկսում ուղարկման պրոցեսը:

Փակուղի է առաջանում

Պրոցես 0	Պրոցես 1
MPI_Recv պրոցես 1-ից MPI_Send պրոցես 1-ին	MPI_Recv պրեցես 0-ից MPI_Send պրեցես 0-ին

Կարող է առաջանալ փակուղի

Պրոցես 0	Պրոցես 1
MPI_Send պրոցես 1-ին MPI_Recv պրոցես 1-ից	MPI_Send պրեցես 0-ին MPI_Recv պրեցես 0-ից

Դիտարկենք փակուղային իրավիճակներից խուսափելու տարբեր իրավիճակներ:

1. Ամենապարզ տարբերակներից մեկը կլինի այն, որ պրոցեսներից մեկում փոխվի ուղարկման և ընդունման պրոցեդուրաների հերթականությունը:

Պրոցես 0	Պրոցես 1
MPI_Send պրոցես 1-ին MPI_Recv պրոցես 1-ին	MPI_Recv պրոցես 0-ից MPI_Send պրոցես 0-ին

2. Մեկ այլ տարբերակ է չարգելափակող գործողությունների օգտագործումը: Հաղորդագրությունների ընդունման արգելափակող պրոցեդուրան փոխարինենք MPI_Irecv պրոցեդուրայով: Տեղադրենք այն MPI_Send պրոցեդուրայից առաջ, այսինքն՝

Պրոցես 0	Պրոցես 1
MPI_Send պրոցես 1-ին MPI_Recv պրոցես 1-ից	MPI_Irecv պրոցես 0-ից MPI_Send պրոցես 0-ին MPI_Wait

Այսպիսի իրավիճակում փակուղի երբեք չի առաջանա, քանի որ MPI_Send պրոցեդուրայի կանչի ժամանակ արդեն իսկ հաղորդագրության ընդունման հարցը դրված կլինի, իսկ դա նշանակում է, որ տվյալների փոխանցումը կարելի է սկսել: Ընդ որում, խորհուրդ է տրվում MPI_Irecv պրոցեդուրան ծրագրում հնարավորինս շուտ դնել, որպեսզի հնարավորություն տրվի փոխանցումը ավելի վաղ սկսել և առավելագույն չափով օգտագործել ասինքրոնության առավելությունները:

Անհրաժեշտ սարքավորումները. UNIX օպերացիոն համակարգով համատեղվող հաշվողական կլաստեր, MPI միջավայր, ծրագրավորողի աշխատանքային հարթակ՝ խնդիրների ղեկավարման համար:

Աշխատանքի իրականացման կարգը. ուսանողը պատրաստում է MPI-ծրագրային կոդը, կոմպիլացնում է, աշխատեցնում և վերլուծում ելքային տվյալները:

Աշխատանքի 1-ին մաս. առաջին առաջադրանքը պարզագույն EXAMPLE_1.C MPI-ծրագրի կոմպիլյացիան է, նրա աշխատեցումը դասախոսի կողմից հանձնարարված հաշվողական հանգույցի վրա, ապա նաև արդյունքների վերլուծումը:

```
// source code of EXAMPLE_1.C program
#include "mpi.h"
#include <stdio.h>
#include <sys/timeb.h> // for ftime function
int main(int argc, char **argv)
{
 int CURR_PROC, ALL_PROC, NAME_LEN;
char PROC_NAME[MPI_MAX_PROCESSOR_NAME];
struct timeb t; // time contain structure

MPI_Init (&argc, &argv);
/* current process */
    MPI_Comm_rank (MPI_COMM_WORLD, &CURR_PROC);
/* all process */
MPI_Comm_size (MPI_COMM_WORLD, &ALL_PROC);

// get processor name (really computer name)
MPI_Get_processor_name(PROC_NAME, &NAME_LEN);

ftime(&t); // get current time point
// t.time is time in sec since January 1,1970
// t.millitm is part of time in msec
// ATTENSION ! ftime may by used only after synchronization all
// processors time
// output to STDIN
    printf("I am process %d from %d and my name is %s (time: %.3f
    sec)\r\n", CURR_PROC, ALL_PROC, PROC_NAME, (t.time+1e-
    3*t.millitm));
```

```
MPI_Finalize();
} // end of EXAMPLE_1 program
```

Առավել հետաքրքիր է ծրագրի յուրաքանչյուր ճյուղի արտածած տողերի հաջորդականությունը.

```
I am process 0 from 5 and my name is Comp_07 (time: 1110617937.329 sec)
I am process 2 from 5 and my name is Comp_06 (time: 1110617937.329 sec)
I am process 4 from 5 and my name is Comp_11 (time: 1110617939.003 sec)
I am process 1 from 5 and my name is Comp_02 (time: 1110617939.003 sec)
I am process 3 from 5 and my name is Comp_01 (time: 1110617938.997 sec)
```

Ժամանակը 10^3 ճշտությամբ որոշելու համար օգտագործվում է C-ի ստանդարտ *ftime* ֆունկցիան (դուրս է բերում յուրաքանչյուր պրոցեսի համար լոկալ ժամանակը): MPI_Wtime ֆունկցիան յուրաքանչյուր կանչված պրոցեսի համար վերադարձնում է լոկալ աստղաբաշխական ժամանակը վայրկյաններով (double). Ենթադրվում է, որ որպես ժամանակի սկիզբ վերցված պահը չի փոփոխվի պրոցեսի գոյության ամբողջ ընթացքում:

MPI_Wtick ֆունկցիան վերադարձնում է double-արժեք, որը որոշում է պրոցեսորի թայմերի ընդլայնումը (երկու <զարկերի> միջև ժամանակահատվածը): *Ftime* և MPI_Wtick ֆունկցիաները՝ առանց բոլոր պրոցեսների ժամերի (հարկադիր) սինքրոնացման, կարելի է օգտագործել ժամանակի միայն հարաբերական (այլ ոչ բացարձակ) ժամանակահատվածները որոշելու համար: Այն դեպքում, երբ MPI_WTIME_IS_GLOBAL=1, ապա պրոցեսորների ժամերը սինքրոնացված են, եթե հավասար է 0-ի՝ ապա սինքրոնացված չեն:

Հարկավոր է ծրագրի կատարվող մոդուլը աշխատեցնել մի քանի անգամ՝ նշելով պրոցեսորների տարբեր քանակներ: Ցանկալի է նախապես փորձել պատասխանել տվյալների հաջորդական արտածման հնարավորության հարցին:

Աշխատանքի 2-րդ մասը. Խնդիրը հետևյալն է՝ յուրացնել տվյալների ուղարկման և ստացման պարզագույն՝ MPI_Send և MPI_Recv ֆունկցիաների ֆորմալ կիրառումը: Այս ֆունկցիաներն իրականացնում են պրոցեսների միջև պարզագույն <կետ-կետ> կապը բլոկավորումով (մի ճյուղը կանչում է MPI_Send ֆունկցիան ուղարկման, իսկ մյուսը՝ MPI_Recv-ը տվյալների ստացման համար): Այս երկու ֆունկցիաներն էլ (ինչպես և մյուսները) վերադարձնում են սխալի կոդը որպես ամբողջ թիվ, իսկ բարեհաջող ավարտին համապատասխանում է MPI_SUCCESS-ը (մանրանասները՝ mpi.h ֆայլում):

Հաջորդ օրինակում (EXAMPLE_2.C ֆայլը) զրոյական պրոցեսը տեքստային տող է ուղարկում (999 տեգով) մեկ պրոցեսին (<կետ-կետ> փոխանակում), այնտեղ էլ հենց այդ տողը տպվում է.

```
// source code of EXAMPLE_2.C program
#include "mpi.h"
#include <stdio.h>
int main (int argc, char **argv)
{
char message[20];
int i=1, all_rank, my_rank;

MPI_Status status;
MPI_Init (&argc, &argv);
        MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
MPI_Comm_size (MPI_COMM_WORLD, &all_rank);

if (my_rank==0) /* code for process zero */
{
    strcpy (message, "Hello, there!\0");
    //for(i=1; i<all_rank; i++)
```

```
        MPI_Send(message, strlen(message), MPI_CHAR, i, 999,
MPI_COMM_WORLD);
}
else /* code for other processes */
{
        MPI_Recv(message, 20, MPI_CHAR, 0, 999,
        MPI_COMM_WORLD, &status);
        //MPI_Recv(message, 20, MPI_CHAR, 0, MPI_ANY_TAG,
//   MPI_COMM_WORLD, &status);

printf ("I am %d process; received: %s\n", my_rank, message);
}

MPI_Finalize();
} // end of EXAMPLE_2 program
```

Քանի որ օգտագործվել է MPI_Recv արգելափակող ֆունկցիան, 2-ից ավելի պրոցեսորների վրա աշխատեցնելու դեպքում ծրագիրը ճշտորոշ ձևով չի ավարտվում (կախված օգտագործված MPI-տարբերակից՝ կատարումը կավարտվի սխալով կամ նախատեսված ժամանակի ավարտից հետո):

Եթե հեռացնենք մեկնաբանությունները հետևյալ տողում //for(i=0; i<all_rank; i++) և փոխարինենք MPI_Recv(message, 20, MPI_CHAR, 0, 999, MPI_COMM_WORLD, &status)-ը MPI_Recv (message, 20, MPI_CHAR, 0, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status)-ի հետ, ապա ծրագրի աշխատանքի արդյունքում կստանանք հետևյալ տողերը.

```
I am 2 process; received: Hello, there!
I am 1 process; received: Hello, there!
I am 4 process; received: Hello, there!
I am 3 process; received: Hello, there!
```

Վերջին դեպքում օգտագործվել են նախապես որոշված MPI-հաստատուններ (<ջոկերներ>)՝ MPI_ANY_SOURCE և MPI_ANY_TAG, որոնք նշանակում են «ընդունիր ցանկացածից» և «ընդունիր հաղորդագրություն ցանկացած տեգով»: Այս դեպքում ծրագիրը կավարտվի բնականոն ձևով (քանի որ կբարարավեն բոլոր MPI_Recv արգելափակող կանչերը):

Լրացուցիչ խնդիրներ (ուսանողների ինքնուրույն աշխատանք).

- Բարդացնել տեքստային տողի կառուցվածքը, որն ուղարկվում է որպես հաղորդագրություն (ներմուծել պրոցեսի համարը, քոմփյութերի անունը, փոխանցելու ժամանակը)
- Բոլոր պրոցեսներին հաղորդագրություն ուղարկելու համար օգտագործել միաժամանակյա փոխանցման MPI_Bcast ֆունկցիան:
- MPI_Send/MPI_Recv զույգի փոխարեն օգտագործել ուղարկման և ստացման MPI_Sendrecv և MPI_Sendrecv_replace ունիվերսալ ֆունկցիաները, գնահատել այս դեպքում ծրագրի կատարման ժամանակի հնարավոր նվազեցումը (իրականացնել կրկնության ցիկլ 1000-ից 10000 և վերցնել միջինը, կատարման ժամանակը գնահատելու համար օգտվել t1=MPI_Wtime(); ..... t2=MPI_Wtime(); dt=t2-t1; կոնստրուկցիայից):

Հարցեր ինքնաստուգման համար

1. Զուգահեռ ծրագրավորման ժամանակ քանի՞ փուլից է բաղկացած պրոցեսի կյանքի ցիկլը:

2. Ինչու՞ չի կարելի ճշգրիտ որոշել հաշվողական հանգույցների վրա պրոցեսների աշխատանքի սկզբնական պահը:

3. Ի՞նչ է կոմունիկատորը: Ի՞նչ են օգտագործողի տոպոլոգիաները և որո՞նք են դրանք (բացի զրոյական համարակալմամբ ստանդարտ տոպոլոգիայից):

4. Ո՞րն է հաղորդագրության ուղարկման արգելափակման մեխանիզմը: Ո՞ր MPI-ֆունկցիաներն են իրագործում արգելափակող փոխանցում:

5. Ի՞նչ է հաղորդագրության՝ բուֆերիզացիայով փոխանցումը: Ո՞ր MPI-ֆունկցիաներն են իրագործում բուֆերացումով փոխանցում:

6. Ի՞նչ է «ջոկերների» մեխանիզմը և որո՞նք են դրա առավելությունները: Ինչպիսի՞ խնդիրներ կարող են առաջանալ «ջոկերների» օգտագործման դեպքում:

7. Ի՞նչ է փոխադարձ բլոկավորումը (փակուղային իրավիճակ, ‘deadlock’): Ո՞ր պայմանների դեպքում է այդ երևույթը ի հայտ գալիս:

Լաբորատոր աշխատանք 3.

Զուգահեռ ծրագրի արտադրողականության և հաշվողական կլաստերի կոմունիկացիոն պարամետրերի որոշումը

Աշխատանքի նպատակը զուգահեռ ծրագրի իրական արտադրողականության և հաշվողական կլաստերի կոմունիկացիոն ցանցի արտադրողականությունը (թողունակությունը) որոշելն է:

Տեսական մաս: Յուրաքանչյուր աշխատանք, որը կապված է զուգահեռ ծրագրավորման հետ, մաթեմատիկական ալգորիթմի կառուցվածքի և մաթեմատիկական հաշվողական համակարգի փոխկապակցվածության հարցերի ուսումնասիրություն է, ընդ որում, զուգահեռ ծրագրի նախագծումն ու իրականացումը պետք է բավարարեն օգտագործողի պահանջներին՝ արտադրողականությանն ու ճիշտ աշխատանքին: Որպես զուգահեռ ծրագրի արտադրողականության չափորոշիչներ կարող են հանդես գալ ծրագրի կատարման ժամանակը, արագացումն ու արդյունավետությունը, հիշողության հետ կապված պահանջները, ցանցի թողունակությունը, տվյալների փոխանցման ժամանակ ժամանակի կորուստները, տեղափոխելիությունը, մասշտաբայնությունը, նախագծման, իրականացման, մշակման ծախսումները, սարքավորումներով հագեցվածության նկատմամբ պահանջները և այլն: Դիտարկենք դրանցից մի քանիսը, որոնք առավել հաճախ են օգտագործվում զուգահեռ ալգորիթմների թողունակության վերլուծության ժամանակ:

Զուգահեռ ծրագրի S_p արագացումը որոշվում է հետևյալ ձևով.

$$S_p = \frac{T_s}{T_p},$$

որտեղ T_s-ը զուգահեռ ալգորիթմի աշխատանքի ժամանակն է մեկ պրոցեսորի վրա, իսկ T_p-ն՝ զուգահեռ ալգորիթմի աշխատանքի ժամանակն է p հատ պրոցեսորների վրա:

E_p զուգահեռ ալգորիթմի արդյունավետությունը բնութագրվում է պրոցեսորների բեռնվածության աստիճանով և որոշվում է որպես S_p արագագործության հարաբերություն p հնարավոր առավելագույն արագագործությանը. $E_p = \frac{S_p}{p}$:

p-հատ պրոցեսորների վրա խնդրի լուծումը լավագույն դեպքում պետք է կատարվի p անգամ ավելի արագ, քան մեկ պրոցեսորի վրա, և/կամ պիտի հնարավորություն տա լուծել խնդիրը p–անգամ ավելի շատ տվյալների քանակով: Իրականում այդպիսի արագացման երբևէ չի հասնում: Դրա պատճառը շատ լավ ցույց է տալիս Ամդալի օրենքը [5].

$$S_p \le \frac{1}{f + (1-f)/p},$$

որտեղ S_p-ն ծրագրի աշխատանքի արագացումն է p –հատ պրոցեսորների վրա, իսկ f-ը՝ ծրագրում ոչ զուգահեռ կոդի մասնաբաժինը:

Ծրագրավորման ժամանակ այս բանաձևը ճիշտ է ինչպես ընդհանուր հիշողությամբ մոդելում, այնպես էլ հաղորդագրությունների փոխանցման մոդելում: Ընդհանուր հիշողությամբ մոդելում ոչ զուգահեռ ծրագրային կոդի մասնաբաժինը կազմում են այն օպերատորները, որոնց կատարում է ծրագրի միայն գլխավոր ճյուղը, իսկ հաղորդագրությունների փոխանցման մոդելում ոչ զուգահեռ մասը կազմվում է այն օպերատոր-

ների հաշվին, որոնք կատարվում են բոլոր պրոցեսորների կողմից: Բանաձևից հետևում է, որ p-ապատիկ արագացման կարելի է հասնել միայն այն դեպքում, եթե ոչ զուգահեռ կողի մասնաբաժինը լինի 0: Ակնհայտ է, որ հասնել դրան գործնականում հնարավոր չէ: Ամդալի օրենքն ավելի լավ երևում է աղյուսակ 2-ում.

Աղյուսակ 2.

Ծրագրի աշխատանքի արագացումը՝ կախված ոչ զուգահեռ ծրագրային կոդի մասնաբաժնից

Պրոցեսորների քանակ	Հաջորդական հաշվարկների մասնաբաժին %				
	50	25	10	5	2
	Ծրագրի աշխատանքի արագացումը				
2	1.33	1.60	1.82	1.90	1.96
4	1.60	2.28	3.07	3.48	3.77
8	1.78	2.91	4.71	5.93	7.02
16	1.88	3.36	6.40	9.14	12.31
32	1.94	3.66	7.80	12.55	19.75
64	1.96	3.82	8.76	15.42	28.32
128	1.97	3.90	9.34	17.41	36.15

Զուգահեռ հաշվարկների $\alpha = 1 - f$ մասնաբաժինը կարելի է որոշել հետևյալ ձևերով.

- *Պրոֆիլավորողի (profiler) օգնությամբ. Զուգահեռ հաշվումների մասնաբաժինը գնահատվում է* $\alpha = \frac{T_{par}}{T_{tot}}$ բանաձևով, որտեղ T_{par}-ն զուգահեռ պրոցեդուրաների աշխատանքի

ժամանակն է, իսկ T_{tot} -ը՝ ծրագրի աշխատանքի ամբողջ ժամանակը:

- *Փորձի արդյունքով.* (արագացումը *p–հատ պրոցեսորների վրա). Զուգահեռ հաշվարկների մասնաբաժինը որոշվում է* $\alpha = \frac{p(S_p - 1)}{S_p(p-1)}$ *բանաձևով:*

Կարելի է նշել հետևյալ գործոնները, որոնք հանգեցնում են արագացման նվազմանը.

- *Պրոցեսորների բեռնվածության անհավասարակշռությունը (load unbalancing).* պրոցեսորների միջև հաշվողական աշխատանքի անհավասարաչափ բաշխումը հանգեցնում է որոշ պրոցեսորների անգործության (idle processors), այսինքն չի ապահովվում զուգահեռացման բարձր մակարդակը, որն էլ իր հերթին նվազեցնում է արդյունավետությունը:
- *Պրոցեսորների սինքրոնացում (processor synchronization).* խոչընդոտ (Barriers) տիպի սինքրոնացում ապահովող ծրագրային կոդի աշխատանքը ժամանակային ծախսեր է պահանջում:
- *Կոմունիկացիաներ (communications).* պրոցեսորների միջև հաղորդագրությունների փոխանակման ժամանակ կարող են առաջ գալ ուշացումներ, որոնք բխում են ալգորիթմի ինչպես ծրագրային իրականացման առանձնահատկություններից (օրինակ պրոցեսորի կողմից հաղորդագրության ընդունման պահին այն դեռ ուղարկված չի լինում), այնպես էլ սարքավորումներից (օրինակ, ցանցի բեռնվածությունից):
- *Զուգահեռ մուտք/ելք (parallel input/output).* որպես օրինակ կարող են ծառայել այն ծրագրերը, որոնք աշխատում են տարբեր պրոցեսորների վրա, բայց միաժամանակ են արտածում տվյալները միևնույն ֆայլի մեջ:

Զուգահեռ ալգորիթմի Cost գինը որոշվում է հետևյալ բանաձևով.

$$Cost = T_p * p = \frac{T_s * p}{S_p} = \frac{T_s}{E_p}:$$

Գնային առումով օպտիմալ, կամ աշխատանքի առումով արդյունավետ (work-efficient), կամ ժամանակի առումով արդյունավետ (processor-time optimality) ալգորիթմը այնպիսի ալգորիթմն է, որտեղ խնդրի լուծման համար գնի Cost արժեքը համամասնական է մեկ պրոցեսորի վրա կատարվող ծրագրի ժամանակին. $Cost = T_p * p = k * T_s$, որտեղ k-ն հաստատուն է:

Զուգահեռ հաշվարկների կատարման T_p լրիվ ժամանակը դա կատարման վրա ծախսված T_{comp} ժամանակի և կոմունիկացիաների վրա ծախսված T_{comm} ժամանակի գումարն է: Հետևաբար, զուգահեռ հաշվարկների վրա ծախսված ընդհանուր ժամանակը կլինի $T_p = T_{comp} + T_{comm}$: Իր հերթին կոմունիկացիաների վրա ծախսված ժամանակը օգտագործվում է հաղորդագրությունների ինիցիալիզացման վրա (սկզբնական մոտավորություն)՝ $T_{startup}$ (message latency), և տվյալների փոխանցման ժամանակի վրա, այսինքն՝ $T_{comm} = T_{startup} + n * T_{data}$: $T_{startup}$ սկզբնական մոտարկումը կախված է ինչպես սարքավորումից, այպես էլ ծրագրային ապահովումից: Հետևաբար q հատ n երկարության հաղորդագրություն փոխանցելիս պահանջվող ժամանակը կլինի $T_{comm} = q(T_{startup} + n * T_{data})$: Կարևոր մեծություն է համարվում նաև հաշվողական ծախսումների և կոմունիկացիոն ծախսումների հարաբերությունը՝ T_{comp} / T_{comm}: Այս կոտորակը ցույց է տալիս կոմունիկացիոն ուշացումների քանակական արտահայտությունը հաշվումների ծավալի նկատմամբ:

Ալգորիթմի նախագծման ժամանակ պետք է ձգտել նվազագույնի հասցնել կոմունիկացիաների թիվը, որպեսզի հաշվումների և կոմունիկացիաների ժամանակների հարաբերությունը լինի հնարավորինս մեծ.

$$\frac{T_{comp}}{T_{comm}} = \frac{N_{comp}}{P_c, Mflops} \Big/ \frac{N_{comm}}{P_{net}, Mbytes/\sec},$$

որտեղ N_{comp} -ը օպերացիաների քանակն է, N_{comm} - ը կոմունիկացիաների քանակն է, P_c -ն՝ քոմփյութերի արտադրողականությունը (computer performance), P_{net} -ը՝ ցանցի թողունակությունը (network bandwidth).

Ինչպես երևում է վերը նկարագրվածից, ծրագրի արտադրողականության վրա մեծապես ազդում են կլաստերային համակարգի ճարտարապետությունը և բնութագրիչները: Մեքենայի իրական բնութագրիչները որոշելու և նախագծման փուլին դրանց համապատասխանությունը որոշելու համար կատարում են թեստավորում: Կլաստերների թեստավորման համար օգտագործվում են հատուկ թեստային ծրագրեր *(benchmarks)*: Առանձնանում են թեստային ծրագրերի երեք հիմնական տեսակներ՝ կախված դրանցով անցկացվող թեստերի առանձնահատկություններից.

- Ծրագրերի աշխատանքի արդյունավետության թեստեր (Application Domain Benchmarks), որոնք չափում են արտադրողականությունը: Այն հաշվարկվում է որոշակի ծրագիր լուծելու ժամանակ:
- Կոմունիկացիոն արտադրողականությունը որոշող թեստեր (Communication-Specific Benchmarks), որոնք չափում են կապուղիներորով տվյալների փոխանցման արդյունավետությունը:

• Կլաստերի տարրերի թեստեր (Architecture-Specific Benchmarks), որոնք գնահատում են կլաստերի տարբեր ենթահամակարգերի արտադրողականությունը:

Քանի որ առանձնահատուկ հետաքրքրություն է առաջացնում ամբողջական համակարգի արտադրողականությունը, ուստի սկզբում կանրադառնանք թեստերի առաջին խմբին: Այս խմբին պատկանող ամենահայտնի ծրագրային արտադրանքները հետևյալն են.

• *NPB (NAS Parallel Benchmarks)* [6] – MPI-ծրագրերի հավաքածու, որը մշակված է զուգահեռ քոմփյութերների տարբեր ասպեկտների արտադրողականությունը թեստավորելու համար:

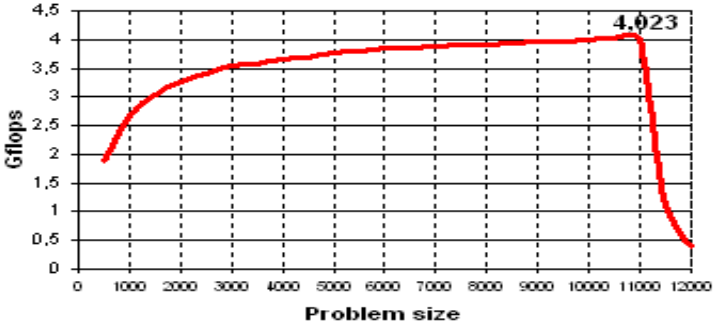
• *HPL (High Performance Linpack)* [7,8] - LU–ձևափոխման (ընդլայման) միջոցով գծային համավասարումների համակարգի լուծում: Այն օգտագործում են աշխարհի առաջատար 500 ամենամեծ արտադրողականությունն ունեցող հաշվողական համակարգերի ցուցակը ձևավորելու համար:

Այժմ դիտարկենք որոշ բնութագրերի ստացումը ԱրմԿլա-ստեր [19, 20, 21, 22] բարձր արտադրողականությամբ համակարգի օրինակի հիման վրա: Արմկլաստերի ընդհանուր արտադրողականությունը թեստավորելու համար օգտագործվել է HPL թեստը: Այն լուծում է պատահականորեն տրված գծային հավասարումների համակարգը LU-ձևափոխման (ընդլայման) մեթոդով, օգտագործում է 64-բիթանոց սահող ստորակետով թվաբանությունը և թույլ է տալիս դատողություններ անել համակարգի արտադրողականության վերաբերյալ՝ լուծելով այդ խնդիրը [8,9]: Այստեղ արտադրողականությունը որոշելիս վերցվում է 1 վայրկյանում սահող ստորակետով կատարվող գործողությունների քանակը: Այդ քանակը գնահատվում է

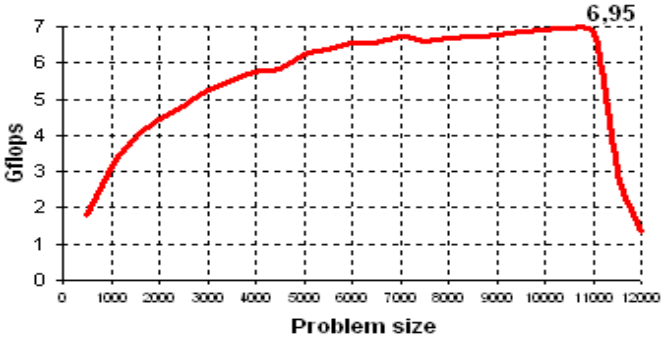
$L = \frac{2}{3} * n^3 + o(n^2)$ բանաձևով, որտեղ n-ը հավասարումների համակարգի չափն է, իսկ համակարգի արտադրողականությունը որոշվում է $R = \frac{L}{t}$ հարաբերությունից, որտեղ t-ն թեստի կատարման ժամանակն է:

Թեստի աշխատանքի համար անհրաժեշտ է MPI միջավայրը, ինչպես նաև գծային հանրահաշվի հիմնական պրոցեդուրաների BLAS գրադարանը: ԱրմԿլաստեր համակարգի համար օգտագործվել է ATLAS գրադարանը, gcc կոմպիլյատորը և զուգահեռ ծրագրավորման MPICH-GM փաթեթը: HPL թեստի հիման վրա ստացվել են կլաստերի հետևյալ բնութագրիչները[19, 28].

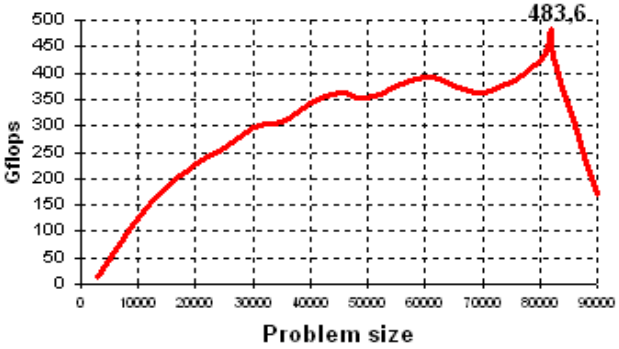
- **Արտադրողականություն**: Առավելագույն արտադրողականություն ստանում ենք, երբ բեռնում ենք կլաստերի ամբողջ օպերատիվ հիշողության 80%-ը: Մեկ պրոցեսորի վրա 10500×10500 չափերի մատրիցի լավագույն արտադրողականությունը ստացվել է 4.023 GFLOPS (նկ. 6): Մեկ հանգույցի վրա 10500×10500 չափերի մատրիցի լավագույն արտադրողականությունը ստացվել է 6.95 GFLOPS (նկ. 7): Արմկլաստերի արտադրողականությունը 82000×82000 չափերի մատրիցի համար կազմել է 483.6 GFLOPS (տես նկ. 8):
- **Արդյունավետ ություն:** Արդյունավետությունը ածանցյալ ցուցանիշ է արտադրողականությունից և ցույց է տալիս, թե որքան արդյունավետ են օգտագործվում կլաստերի սարքավորումների հնարավորությունները որոշակի տիպի խնդիրների լուծման ժամանակ:
- **Մասշտաբավորում ըստ Ամդալի:** Ցույց է տալիս, թե ինչպես է փոխվում արագացումը ֆիքսված չափի (մեկ հաշվողական հանգույցի օպերատիվ հիշողության 80%) HPL խնդրի կատարման ժամանակ՝ հանգույցների տարբեր քանակների դեպքում: (նկ. 9):



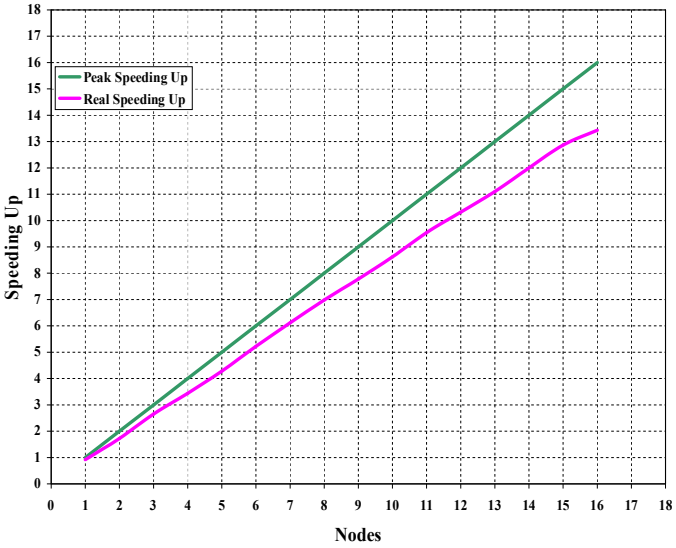
Նկ.6. Արմկլաստերի արտադրողականությունը մեկ պրոցեսորի վրա



Նկ.7. Արմկլաստերի արտադրողականությունը մեկ հանգույցի վրա



Նկ. 8. Արմկլաստերի արտադրողականությունը

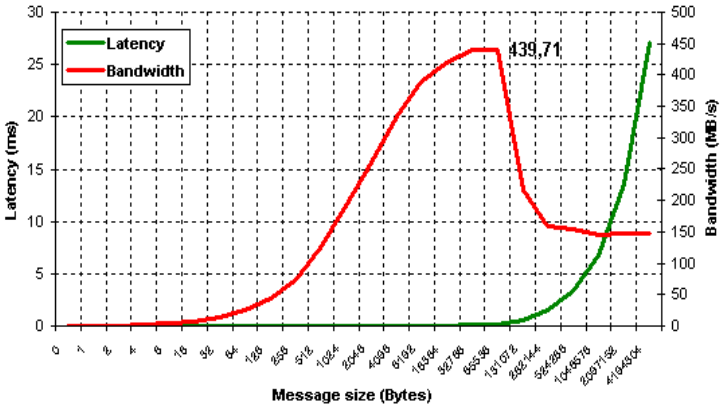


Նկ.9. Արմկլաստերի արագացման աճը 16 հանգույցների վրա

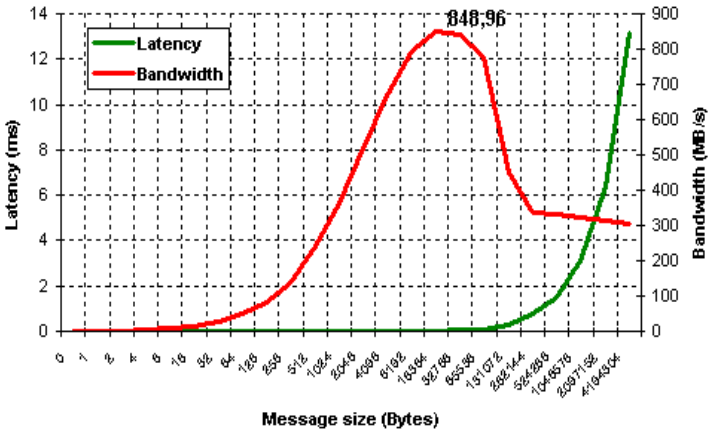
Քանի որ հանգույցների միջև տվյալների հոսքն անցնում է հաշվողական ցանցով, ուստի հսկայական արժեք են ներկայացնում նաև կոմունիկացիոն արտադրողականության թեստերը: Իրական խնդիրների հետ աշխատելիս հանգույցների միջև կոմունիկացիաները կատարվում են MPI հաղորդագրությունների փոխանցման ինտերֆեյսի մակարդակում: Հետևաբար, կապուղիներով փոխանցվող տվյալների արդյունավետությունը կախված է ոչ միայն ցանցի լատենտությունից և թողունակությունից, այլ նաև MPI ֆունկցիաների իրականացման որակից: Myrinet ցանցի թողունակության և լատենտության թեստավորման ժամանակ MPI հիմնական օպերացիաների մակարդակում օգտագործվել է PMB (Pallas MPI Benchmark) թեստը [10]: PMB թեստի նպատակն է MPI հիմնական կոմունիկացիոն պրոցեդուրաների թեստավորումը: Ստացվել են ցանցի թողունակության և լատենտության բնութագրերը.

Առանձին փոխանցումների ժամանակ (*PingPong, PingPing)* հաղորդագրությունները փոխանցվում են երկու պրոցեսորների միջև:

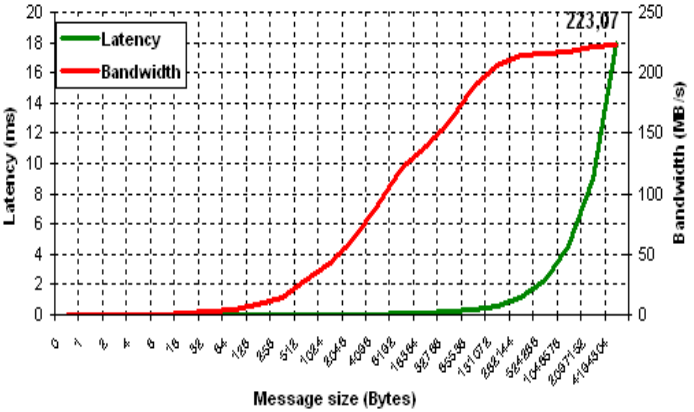
- Զուգահեռ փոխանցումների ժամանակ *(Sendrecv, Exchange, MultiPingPong, MultiSendrecv, MultiExchange)* յուրաքանչյուր պրոցեսի աշխատանքը կատարվում է ցանցի մյուս պրոցեսների աշխատանքին զուգահեռ, չափվում է հաղորդագրության փոխանցման արդյունավետությունը ցանցի համընդհանուր բեռնվածության ժամանակ, թողունակության որոշման ժամանակ վերցվում է բայթ-փոխանցվող հաղորդագրությունների ընդհանուր քանակությունը:
- Կոլեկտիվ փոխանցումների ժամանակ *(Allreduce, Reduce_scatter, Allgather, Allgaterv, Alltoal, Bcast, Barrier և թվարկված թեստերի Multi*-տարբերակները) որոշվում է MPI-ի իրականացման որակը:



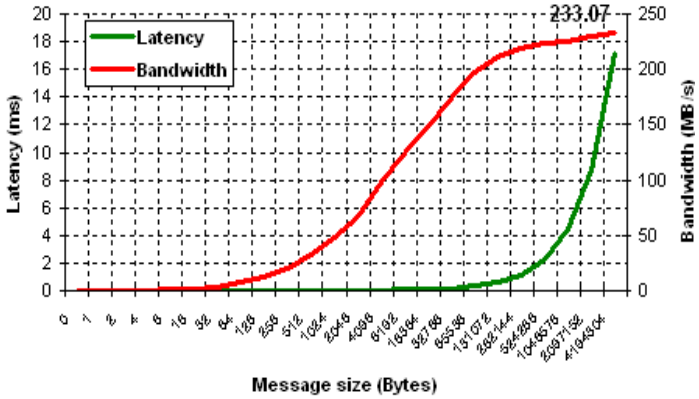
Նկ.10. PingPing տեստի ժամանակ մեկ հանգույցի երկու պրոցեսների համար ցանցի թողունակությունը և լատենտությունը



Նկ.11. PingPong տեստի ժամանակ մեկ հանգույցի երկու պրոցեսների համար ցանցի թողունակությունը և լատենտությունը



Նկ.12. PingPing թեստի ժամանակ երկու հանգույցների վրա երկու պրոցեսների համար ցանցի թողունակությունը և լատենտությունը



Նկ.13. PingPong թեստի ժամանակ երկու հանգույցների վրա երկու պրոցեսների համար ցանցի թողունակությունը և լատենտությունը

Անհրաժեշտ սարքավորումները. UNIX-համատեղելիությամբ ՕՀ-ով հաշվողական կլաստեր, նրա վրա նախապես տեղադրված MPI միջավայրը, ծրագրավորողի աշխատանքային հարթակ՝ օգտագործողի ծրագրերի ղեկավարման համար:

Աշխատանքի իրականամվան կարգը. ուսանողը պատրաստում է MPI-ծրագրի կոդը, կոմպիլացնում և աշխատեցնում է այն, վերլուծում ելքային տվյալները և ներկայացնում է դրանք գրաֆիկական տեսքով:

```
// source code of PROG_MPI.C program
#include "mpi.h"
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
static int message[10000000]; // max length of message
int main (int argc, char *argv[])
{
double time_start,time_finish;
int i, n, nprocs, myid, len;
```

```
MPI_Request rq;
MPI_Status status;

MPI_Init(&argc,&argv); /* initialization MPI */
MPI_Comm_size(MPI_COMM_WORLD,&nprocs); /* all processes */
        /*current process*/
        MPI_Comm_rank(MPI_COMM_WORLD,&myid);

printf("PROGMPI: nprocs=%d myid=%d\n", nprocs, myid);
fflush( stdout );
len = atoi( argv[1] ); // get length message from command line
printf( "length: %d\n", len );
if (myid == 0) /* I a m MASTER-process ! */
{
        n = 1000; /* cycles number */
MPI_Send(&n, 4, MPI_CHAR, nprocs-1,
99,MPI_COMM_WORLD);
        time_start=MPI_Wtime();
        for (i=1; i<=n; i++)
        {
MPI_Send(message, len, MPI_CHAR, nprocs-1, 99,
MPI_COMM_WORLD);
MPI_Recv(message, len, MPI_CHAR, nprocs-1, 99,
MPI_COMM_WORLD,
&status);
        }
        time_finish=MPI_Wtime();
        printf( "Time %f rate %f speed %f\n",
(float)(time_finish-time_start),
(float)(2*n/(time_finish-time_start)),
(float)(2*n*len/(time_finish-time_start)) );
} /* end if (myid==0) */
else if (myid == (nprocs-1)) /* I am is last of all processes ! */
```

```
{
MPI_Recv(&n,4, MPI_CHAR, 0, 99,
MPI_COMM_WORLD,&status);
for (i=1;i<=n;i++)
        {
MPI_Recv(message, len, MPI_CHAR, 0, 99,
MPI_COMM_WORLD, &status);
MPI_Send(message, len, MPI_CHAR, 0, 99,
MPI_COMM_WORLD);
        }
} /* end if (myid==(nproc-1)) */

fflush( stdout );
MPI_Finalize(); /* deinitialize MPI */
} // end of PROG_MPI program
```

Պահանջվում է կառուցել (Excel ծրագրով) կոմունիկացիոն ցանցի իրական արագության կախումը հաղորդագրության չափսից, գնահատել լատենտության չափը, կատարել եզրահանգումներ թեստի ժամանակ օգտագործված հաղորդագրության չափսի վերաբերյալ, որոնց դեպքում հասանելի է դառնում տեսական թողունակությունը:

Հարցեր ինքնաստուգման համար.

1. Որո՞նք են զուգահեռ ծրագրի արագացումը և արդյունավետությունը, զուգահեռ հաշվարկների վրա ծախսված ընդհանուր ժամանակը և գինը:

2. Ինչո՞վ է տարբերվում տարբեր երկարությամբ հաղորդագրությունների փոխանակման ժամանակ ցանցի արտադրողականությունը արտադրողի կողմից ներկայացվածից:

3. Որո՞նք են լատենտության և թողունակության չափորոշիչները: Դրանց չափման միավորները:

4. Ինչպե՞ս է կախված ցանցի իրական թողունակությունը փոխանցվող հաղորդագրությունների չափսից: Ինչ երկարությամբ հաղորդագրությունների դեպքում է արտադրողականությունը հասնում (տեսական) մակարդակին:

5. Փորձով որոշել, թե ինչքան գործողություն (բազմապատկում, գումարում, երկու-ճշտությամբ սահող թվերով բաժանում) է իրականացնում Ձեր կլաստերի հաշվողական հանգույցը լատենտությանը հավասար ժամանակահատվածի ընթացքում: Կատարեք որակական եզրահանգում Ձեր տրամադրության տակ գտնվող սարքավորման համար զուգահեռացման բլոկի ռացիոնալ չափսի վերաբերյալ:

Լաբորատոր աշխատանք 4.

Պարզագույն MPI- ծրագրեր (թվային ինտեգրում)

Աշխատանքի նպատակն է ձեռք բերել պրակտիկ գիտելիքներ պարզագույն MPI-ծրագրերի իրականացման համար:

Տեսական մաս: Թվային անալիզի խնդիրների մեջ շատ են հանդիպում այնպիսիք, որոնց զուգահեռացումը ակնհայտ է: Մասնավորապես, թվային ինտեգրումը բերվում է ինետգրալային ֆունկցիայի հաշվարկին, որն իր հերթին բնական է, որ կարելի է աշխատացնել առանձին պրոցեսների վրա, որտեղ գլխավոր պրոցեսը զբաղվում է հաշվողական պրոցեսի կազմակերպմամբ: Նույնատիպ ինտուիտիվ զուգահեռացումով օժտված են նաև տեսակավորման, որոնման, ֆունկցիայի արմատի թվային որոշման և այլ խնդիրներ:

Անհրաժեշտ սարքավորումները. UNIX-համատեղելիությամբ ՕՀ-ով ղեկավարվող հաշվողական կլաստեր, նրա վրա տեղադրված MPI միջավայր, ծրագրավորողի աշխատանքային հարթակ՝ օգտագործողի խնդիրները ղեկավարելու համար:

Աշխատանքի իրականացվան կարգը. ուսանողը պատրաստում է MPI-ծրագրի նախնական տեքստը, կոմպիլացնում և աշխատեցնում է այն, այնուհետև վերլուծում ելքային տվյալները (և դասախոսի կարգադրությամբ ներկայացնում դրանք գրաֆիկական տեսքով):

Աշխատանքի 1-ին մաս. որպես օրինակ դիտարկվում է սեղանի կանոնով որոշյալ ինտեգրալի հաշվման մի ծրագիր: Հաջորդական տարբերակով հաշվվում է $\int_b^a f(x)dx$ ինտեգրալը՝ բաժանելով *[a,b]* հատվածը *n* հավասար սեգմենտների և

կատարվում է բոլոր հատվածների համար հաշվարկված գնահատականների գումարում հետևյալ բանաձևով.

$$h[f(x_0)/2 + f(x_n)/2 + \sum_{i=1}^{n-1} f(x_i)] \ :$$

Այստեղ $h = (b-a/n)$, և $x_i = a + ih$, որտեղ i = 0, 1, ..., n: Եթե տեղադրենք *f(x)*-ի հաշվարկը առանձին ֆունկցիայի մեջ, ծրագրի հաջորդական տարբերակը կունենա հետևյալ տեսքը.

```
#include <stdio.h>
f loat f(float x) {
float return_val;
 /* Հաշվարկում է f(x).
Արդյունքը հիշվում է return_val. *-ում/
 . . .
 return return_val;
} /* f */

main()
{
 float integral; /* Հաշվարկի արդյունք */
 float a, b; /* Ձախ և աջ սահմաններ */
 int n; /* Հատվածների քանակ */
 float h; /* Հատվածի մեծությունը*/
 float x;
 int i;
printf('"Ներմուծեք a, b, կամ n\n");
scanf('"%f %f %d", &a, &b, &n);
h = (b-a)/n;
integral = (f(a) + f(b))/2.0;
x = a;
for (i = 1; i <= n-1; i++) {
```

```
 x += h;
integral += f(x);
            }
integral *= h;
printf(" n = %d սեղաններով ինտեգրալ \n", n);
printf(" %f-ից մինչև %f = %f` \n", a, b, integral);
} /* main */
```

Սեղանի կանոնի զուգահեռացումը: Այս ծրագրի զուգահեռացման տարբերակը բերվում է նրան, որպեսզի *[a,b]* հատվածը բաժանվի պրոցեսների միջև, և յուրաքանչյուր պրոցես գնահատի *f(x)* ինտեգրալը իր ենթահատվածում: Ամբողջական ինտեգրալի գնահատականը ստանալու համար բոլոր պրոցեսների լոկալ արժեքները կգումարվեն:

Ենթադրենք ծրագիրը պարունակում *p* հատ պրոցեսներ և *n* հատ սեղաններ, որտեղ *n*-ը *p*-ին բազմապատիկն է: Այդ դեպքում առաջին պրոցեսը հաշվարկում է առաջին *n/p* սեղանների տիրույթը, երկրորդը՝ հաջորդ *n/p* տիրույթը և այդպես շարունակ: *q*-րդ պրոցեսը կհաշվարկի ինտեգրալը այս հատվածի համար.

$$\left[a + q\frac{nh}{p}, a + (q+1)\frac{nh}{p} \right].$$

Յուրաքանչյուր պրոցես պետք է տիրապետի հետևյալ ինֆորմացիան.

- p պրոցեսների քանակը,
- սեփական ռանգը,
- ինտեգրման ամբողջ [a;b] միջակայքը,
- ենթահատվածների n քանակությունը:

Առաջին երկու կետերը կարելի է որոշել MPI_Comm_size() և MPI_Comm_rank() ֆունկցիաների օգնությամբ: Վերջին երկու

կետերը ներմուծվում են օգտագործողի կողմից: Սակայն դա կարող է պահանջել ևս մի քանի խնդիրների լուծում: Այդ պատճառով էլ ինտեգրալի հաշման առաջին փորձի համար այդ արժեքները տրվում են ծրագրի կոդում:

Պրոցեսների անհատական հաշվումների արդյունքների ամփոփումը հետևյալն է. յուրաքանչյուր պրոցես իր լոկալ արժեքը ուղարկում է 0-րդ պրոցեսին, իսկ 0-րդ պրոցեսը կատարում է վերջնական գումարումը: Ծրագիրը կընդունի հետևյալ տեսքը:

```
#include <stdio.h>
#include "mpi.h"

float f(float x) {
float return_val;

 /* Հաշվարկում է f(x).
Վերադարձնում է արժեքը return_val. */
 . . .
 return return_val;
}

float Trap(float local_a, float local_b, int local_n, float h) {
float integral; /* Հաշվարկների արդյունքը */
float x;
int i;
integral = (f(local_a) + f(local_b))/2.0;
x = local_a;
for (i = 1; i <= local-n-1; i++) {
 x += h;
 integral += f(x);
            }
```

```
integral *= h;
return integral;
} /* Trap */

main(int argc, char** argv) {
int my_rank; /* Պրոցեսի ռանգը */
int p; /* Պրոցեսների քանակը */
float a = 0.0; /* Ձախ սահմանը */
float b = 1.0; /* Աջ սահմանը */
int n = 1024; /* սեղանների քանակը */
float h; /* Սեղանների լայնությունը */
float local_a; /* Ձախ սահմանը պրոցեսի համար*/
float local_b; /* Աջ սահմանը պրոցեսի համար */
int local_n; /* Հաշվարկվող սեղանների քանակը */
float integral; /* Յուրաքանչյուր հատվածում ինտեգրալի
արժեքը */
float total; /* Ամբողջ ինտեգրալը */
int source; /* Պրոցեսը, որն ուղարկում է ինտեգրալը */
int dest = 0; /* 0-րդ պրոցեսը որպես վերջնական */
int tag = 50;
MPI_Status status;

MPI_Init(&argc, &argv);
/* Որոշել պրոցեսի ռանգը */
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

/* Որոշել պրոցեսների քանակը */
MPI_Comm_size(MPI_COMM_WORLD, &p);

h = (b-a)/n; /*h-ը նույնն է բոլոր պրոցեսների համար*/
local_n = n/p; /* Ստանալ սեղանների քանակը*/

/* Հաշվել հատվածի սահմանները և մասնակի ինտեգրալը*/
```

```
local_a = a + my_rank*local_n*h;
local_b = local_a + local_n*h;
integral = Trap(local_a, local_b, local_n, h);

/* Գումարել բոլոր ինտեգրալները*/
if (my_rank == 0) {
total = integral;
 for (source = 1; source < p; source++) {
MPI_Recv(&integral, 1, MPI_FLOAT,
source, tag, MPI_COMM_WORLD, &status);
total += integral;
            }
            } else {
 MPI_Send(&integral, 1, MPI_FLOAT, dest,
tag, MPI_COMM_WORLD);
            }
/* Արդյունքի արտածում */
if (my_rank == 0) {
printf(" n = %d սեղաններով գնահատականը \n", n);
printf(" ինտեգրալի %f-ից մինչև %f = %f` \n", a, b, total);
            }

/* MPI-ծրագրի ավարտը */
MPI_Finalize();
} /* main */
```

Զուգահեռ պրոցեսների ներմուծում/արտածում: Նախորդ օրինակում մի քանի արժեքներ հաստատուն որոշված էին ծրագրային կոդում: Նման դեպքերում շատ հաճախ անհրաժեշտություն է առաջանում օգտագործողի կողմից լրացուցիչ ներմուծել պահանջվող արժեքները: Որպես կանոն, տերմինալի հետ աշխատանքը ապահովելու համար մուտքն ու ելքը իրականացվում է մեկ պրոցեսի միջոցով: Որպես օրինակ ենթադ-

րենք, որ փոխանակությունը կատարվում է 0-րդ պրոցեսի միջոցով: Անհրաժեշտ է մյուս պրոցեսներին փոխանցել ինտեգրալի ձախ և աջ սահմանները և սեղանների քանակը: Դրա համար կարող է օգտագործվել հետևյալ ֆունկցիան.

```
void Get_data(int my_rank, int p, float* a_ptr, float* b_ptr, int* n_ptr)
{
int source = 0;
int dest;
int tag;
MPI_Status status;

if (my_rank == 0) {
printf("Ներմուծեք a, b, n\n");
scanf("%f %f %d", a_ptr, b_ptr, n_ptr);

for (dest = 1; dest < p; dest++) {
tag = 30;
MPI_Send(a_ptr, 1, MPI_FLOAT, dest, tag,
MPI_COMM_WORLD);
tag = 31;
MPI_Send(b_ptr, 1, MPI_FLOAT, dest, tag,
MPI_COMM_WORLD);
 tag = 32;
 MPI_Send(n_ptr, 1, MPI_INT, dest, tag,
MPI_COMM_WORLD);
            }
             } else {
tag = 30;
MPI_Recv(a_ptr, 1, MPI_FLOAT, source, tag,
MPI_COMM_WORLD, &status);
tag = 31;
MPI_Recv(b_ptr, 1, MPI_FLOAT, source, tag,
```

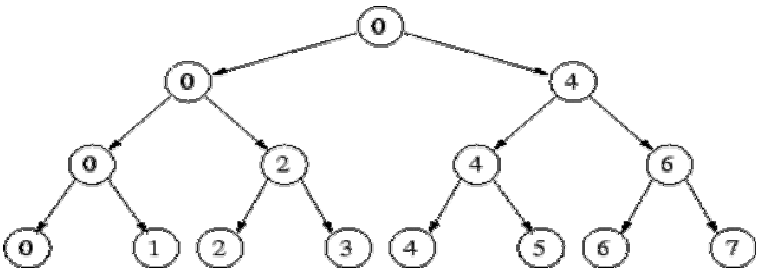
```
MPI_COMM_WORLD, &status);
tag = 32;
MPI_Recv(n_ptr, 1, MPI_INT, source, tag,
MPI_COMM_WORLD, &status);
        }
}/* Get_data */
```

Աշխատանքի 2-րդ մաս. Կոլեկտիվ կոմունիկացիաներ: Ինտեգրալի հաշվմման ծրագրի արտադրողականությունը կարելի է էապես բարձրացնել: Օրինակ, ենթադրենք ծրագիրը կատարվում է ութ պրոցեսորների վրա: Բոլոր պրոցեսները սկսում են կատարել ծրագիրը միաժամանակ: Սակայն հիմնական խնդիրների կատարումից հետո (MPI_Init(), MPI_Comm_size() և MPI_Comm_rank() կանչերը) 1-ից մինչև 7-րդ պրոցեսները կկանգնեն, մինչդեռ 0-րդ պրոցեսը կհավաքի մուտքային տվյալները: Այն բանից հետո, երբ 0-րդ պրոցեսը կհավաքի մուտքային տվյալները, ավելի բարձր ռանգի պրոցեսները կշարունակեն սպասել, մինչև որ 0-րդ պրոցեսը կուղարկի մուտքային տվյալները բոլոր պրոցեսներին: Այսպիսի ոչ արդյունավետ վիճակ կստեղծվի նաև ծրագրի ավարտին, երբ 0-րդ պրոցեսը կհավաքի և կսկսի գումարել լոկալ ինտեգրալների արժեքները:

Զուգահեռ մշակման կազմակերպման հիմնական խնդիրն այն է, որ համակարգի բոլոր պրոցեսորները լինեն հավասարապես ծանրաբեռնված: Եթե, ինչպես վերը հիշատակված խնդրում, երբ հիմնական ծանրաբեռնվածությունն ընկնում է պրոցեսներից մեկի վրա, որոշ դեպքերում առավել արդյունավետ է օգտագործել միապրոցես ձևը:

Ծառանման կոմունիկացիաներ: Պրոցեսորների միջև ծանրաբեռնվածության հավասարաչափ բաշխումը ենթադրում է որևէ կառուցվածքի ներսում տվյալների փոխանցման որոշակի կազմակերպում: Օրինակ, ինտեգրալի հաշման խնդրում պրոցեսների միջև մուտքային տվյալների բաժանման ժամանակ լավ արդյունք կարող է տալ պրոցեսների ծառանման կառուցվածքը, որի արմատը 0-րդ պրոցեսն է (նկ 14):

Տվյալների բաշխման առաջին փուլում 0-րդ պրոցեսը տվյալները փոխանցում է 4-րդ պրոցեսին: Հաջորդ փուլի ընթացքում 0-րդ պրոցեսը տվյալներն ուղարկում է 2-րդ պրոցեսին, միևնույն ժամանակ 4-րդ պրոցեսը այդ նույն տվյալներն ուղարկում է 6-րդ պրոցեսին: Վերջին փուլի ընթացքում 0-րդ պրոցեսը տվյալներն ուղարկում է 1-ին պրոցեսին, միևնույն ժամանակ 2-րդ պրոցեսը տվյալներն ուղարկում է 3-րդին, 4-րդը՝ 5-րդին, և, վերջապես, 6-րդը՝ 7-րդին: Ընդ որում, տվյալների բաշխման ցիկլը պակասեց 7-ից մինչև 3: Եթե գոյություն ունեն p հատ պրոցեսներ, ապա այս պրոցեդուրան հնարավորություն է տալիս բաշխել տվյալները $[\log_2(p)]$ հատ փուլերի ընթացքում՝ p-1 –ի փոխարեն, որը p-ի բավարար մեծ արժեքի դեպքում էականորեն արագացնում է պրոցեսը:



Նկ.14. Պրոցեսների կազմակերպումը ծառի տեսքով

Որպեսզի մոդիֆիկացվի Get_data() ֆունկցիան՝ ծառանման բաշխումն օգտագործելու համար, անհրաժեշտ է մտցնել $[\log_2(p)]$ իտերացիաներից բաղկացած ցիկլ: Ընդ որում, յուրաքանչյուր պրոցես ամեն մի քայլի ժամանակ որոշում է.

- Ընդունում է ինքն արդյոք տվյալներ, և եթե այո, ապա որն է այդ տվյալների աղբյուրը;
- Փոխանցում է ինքն արդյոք տվյալներ, և եթե այո, ապա որն է տվյալներն ընդունողը;

Նշենք, որ փոխանցումների կատարման կարգն ու պրոցեսների ծառանման կազմակերպումը կարող են իրականացվել տարբեր մեթոդներով: Այդ պատճառով էլ երբեմն պետք է օգտագործել փոխանցումների կազմակերպման ավելի արդյունավետ հնարավորություններ: Օրինակ, նկարագրված օրինակում փոխանցումը կատարվում է այս հաջորդականությամբ.

- 0-ն փոխանցում է 4-ին,
- 0-ն փոխանցում է 2-ին, 4-ը փոխանցում է 6-ին;
- 0-ն փոխանցում է 1-ին, 2-ը փոխանցում է 3-ին, 4-ը փոխանցում է 5-ին, 6-ը փոխանցում է 7-ին:

Կարելի է օգտագործել այլ տարբերակ, օրինակ.

- 0-ն փոխանցում է 1-ին,
- 0-ն փոխանցում է 2–ին, 1-ը փոխանցում է 3-ին;
- 0-ն փոխանցում է 4-ին, 1-ը փոխանցում է 5-ին, 2-ը փոխանցում է 6-ին, 3-ը փոխանցում է 7-ին:

Լայնասփյուռ (brodcast) փոխանցումներ: Կոմունիկատորի բոլոր պրոցեսների մասնակցությամբ կոմունիկացիաները կոչվում են կոլեկտիվ: Կոլեկտիվ կապը ենթադրում է երկու և ավելի պրոցեսների մասնակցություն: Լայնասփյուռ հաղորդագրությունը կոլեկտիվ կոմունիկացիա է, երբ առանձին պրոցեսը միատեսակ տվյալներ է ուղարկում յուրաքանչյուր պրո-

ցեսին: MPI-ում գոյություն ունի լայնասփյուռ փոխանցման MPI_Bcast() ֆունկցիան.

```
int MPI_Bcast(void* message, int count,MPI_Datatype datatype, int r
oot, MPI_Comm comm);
```

Այս ֆունկցիան ուղարկում է հաղորդագրություն պատճեն root պրոցեսից comm կոմունիկատորում գտնվող բոլոր պրոցեսներին: Այն պետք է կանչեն կոմունիկատորում գտնվող բոլոր պրոցեսները՝ միևնույն արգումենտներով՝ root-ի և comm-ի համար: Լայնասփյուռ հաղորդագրությունը չի կարող ստացվել MPI_Recv()-ով: count և datatype պարամետրերն ունեն նույն արժեքը, ինչ որ ունեն MPI_Send()-ում, և MPI_Recv()-ում՝ որոշում են հաղորդագրության բովանդակությունը: Սակայն ի տարբերություն ուղղորդված փոխանցման ֆունկցիայի, MPI-ը պահանջում է, որպեսզի կոլեկտիվ համագործակցության ժամանակ count-ը և datatype-ը ունենան կոմունիկատորի բոլոր պրոցեսների համար միևնույն արժեքները:

Տվյալների ստացման Get_data() ֆունկցիան MPI_Bcast()-ի օգտագործմամբ կարելի է արտագրել հետևյալ ձևով.

```
void Get_data2(int my_rank, float* a_ptr, float* b_ptr, int* n_ptr) {

int root = 0; /* MPI_Bcast –ի արգումենտները */
int count = 1;

if (my_rank == 0) {
printf("'Ներմուծեք a, b, և n\n");
scanf("'%f %f %d", a_ptr, b_ptr, n_ptr);
}
MPI_Bcast(a_ptr, 1, MPI_FLOAT, root, MPI_COMM_WORLD);
MPI_Bcast(b_ptr, 1, MPI_FLOAT, root, MPI_COMM_WORLD);
MPI_Bcast(n_ptr, 1, MPI_INT, root, MPI_COMM_WORLD);
} /* Get-data2 */
```

Ռեդուկցիա: Դիտարկված ծրագրում մուտքային փուլից հետո յուրաքանչյուր պրոցեսոր կատարում է միևնույն հրամանները՝ մինչև գումարման վերջնական փուլը: Այդ պատճառով էլ, եթե *f(x)* ֆունկցիան լրացուցիչ բարդացրած չէ (այսինքն չի պահանջում նշանակալից աշխատանք՝ գնահատելու ինտեգրալը *[a;b]* հատվածի առանձին կտորներում), ապա ծրագրի այդ հատվածը պրոցեսորների միջև բաշխվում է հավասարաչափ: Ծրագրի վերջնական գումարման փուլում դարձյալ 0-րդ պրոցեսը ստանում է մյուսների համեմատ անհամաչափ քանակի աշխատանք: Այստեղ ևս կարելի է գումարման աշխատանքը կազմակերպել բոլոր պրոցեսորների միջոցով՝ օգտագործելով ծառանման կառուցվածքը հետևյալ ձևով.

1) 1-ին պրոցեսը արդյունքն ուղարկում է 0-րդ պրոցեսին, 3-րդ պրոցեսը արդյունքն ուղարկում է 2-րդին, 5-րդ պրոցեսը արդյունքն ուղարկում է 4-րդ պրոցեսին, 7-դ պրոցեսը արդյունքն ուղարկում է 6-րդ պրոցեսին:

2) 0-րդ պրոցեսը գումարում է արդյունքը 1-ին պրոցեսի հետ, 2-րդ պրոցեսը գումարում է արդյունքը 3-րդ պրոցեսի հետ, և այլն:

3) 2-րդ պրոցեսն ուղարկում է արդյունքը 0-րդ պրոցեսին, 6-րդ պրոցեսն արդյունքն ուղարկում է 4-րդ պրոցեսին:

4) 0-րդ պրոցեսը գումարում է արդյունքը 2-րդ պրոցեսի հետ, 4-րդ պրոցեսը գումարում է արդյունքը 6-րդ պրոցեսի հետ:

5) 4-րդ պրոցեսն ուղարկում է արդյունքը 0-րդ պրոցեսին:

6) 0-րդ պրոցեսը գումարում է արդյունքը 4-րդ պրոցեսի հետ:

Հնարավոր են նաև փոխանցումների կազմակերպման այլ տարբերակներ, ինչպես որ մուտքային տվյալների բաշխման ժամանակ էր: Այդ պատճառով էլ պետք է օգտագործել այդ նպատակին ավելի հարմար այլ մեխանիզմներ:

Պահանջվող հաշվարկվելիք «ընդհանուր գումարը» կոմունիկացիաների կոլեկտիվ գործողությունների ընդհանուր դասի օրինակ է, որոնք կոչվում են ռեդուկցիայի գործողություններ: Ռեդուկցիայի գլոբալ գործողության ժամանակ կոմունիկատորում բոլոր պրոցեսները փոխանցում են տվյալներ, որոնք միավորվում են՝ օգտվելով բինար գործողություններից: Հիմնական բինար գործողություններն են՝ գումարում, մաքսիմումի և մինիմումի հաշվում, տրամաբանական և այլ նմանատիպ գործողությունները: MPI-ը իր մեջ ներառում է ռեդուկցիայի գործողության կատարման համար հատուկ ֆունկցիա.

```
int MPI_Reduce(void* operand, void* result, int count,
MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
```

MPI_Reduce()-ը միավորում է *operand-ում պահվող օպերանդները՝ օգտագործելով op օպերատորը և արդյունքը պահում է root հիմնական պրոցեսի *result փոփոխականի մեջ: Եվ օպերանդը, և արդյունքը հղվում են count քանակի datatype տիպի հիշողության բջիջների վրա: MPI_Reduce()-ը պետք է կանչեն comm կոմունիկատորի բոլոր պրոցեսները: Կանչի ժամանակ count, datatype և op-ի արժեքները պետք է բոլոր պրոցեսներում լինեն միատեսակ: op արգումենտը կարող է ընդունել աղյուսակ 3-ում նշված ֆիքսված արժեքները: Գոյություն ունի նաև սեփական արժեքների որոշման հնարավորություն:

Ռեդուկցիայի գործողության տարբերակները.

Գործողության անվանումը	Իմաստը
MPI_MAX	Մաքսիմում
MPI_MIN	Մինիմում
MPI_SUM	Գումար
MPI_PROD	Արտադրյալ
MPI_LAND	Տրամաբանական ԵՎ
MPI_BAND	Բիթային ԵՎ
MPI_LOR	Տրամաբանական ԿԱՄ
MPI_BOR	Բիթային ԿԱՄ
MPI_LXOR	Տրամաբանական բացառող ԿԱՄ
MPI_BXOR	Բիթային բացառող ԿԱՄ
MPI_MAXLOC	Մաքսիմումը և նրա տեղը
MPI_MINLOC	Մինիմումը և նրա տեղը

Այսպիսով, ինտեգրալի հաշվման ծրագրի ավարտը կլինի հետևյալը.

```
        /* Գումարում յուրաքանչյուր պրոցեսից */
MPI_Reduce(&integral, &total, 1, MPI_FLOAT,
        MPI_SUM, 0, MPI_COMM_WORLD);
        /* Արդյունքի արտածում*/
```

Կոլեկտիվ կոմունիկացիաների այլ ֆունկցիաներ: MPI-ը ունի կոլեկտիվ կոմունիկացիաների կազմակերպման մի քանի այլ ֆունկցիաներ.

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

MPI_Barrier()-ը ներկայացնում է comm կոմունիկատորում բոլոր պրոցեսների սինքրոնացման մեխանիզմը: Կոմունիկատորի յուրաքանչյուր պրոցես դադարեցնում է իր աշխատանքը, քանի դեռ comm-ում բոլոր պրոցեսները չեն կանչել MPI_Barrier()-ը.

```
int MPI_Gather(void* send_buf, int send_count,MPI_Datatype send_
type,
    void* recv_buf, int recv_count, MPI_Datatype recv_type,int root,
    MPI_comm comm);
```

Comm-ում յուրաքանչյուր պրոցես root ռանգով պրոցեսին է ուղարկում send_buf-ի պարունակությունը: Root պրոցեսը միավորում է ստացված տվյալները պրոցեսների ռանգի հերթականությամբ և այն տեղադրում է recv_buf: Recv արգումենտներն արժեք ունեն միայն root ռանգով պրոցեսում: Recv_count արգումենտը ցույց է տալիս յուրաքանչյուր պրոցեսից ստացված տարրերի քանակը, բայց ոչ թե ընդհանուրը.

```
int MPI_Scatter(void* send_buf, int send_count,MPI_Datatype send_
type,
    void* recv_buf, int recv_count, MPI_Datatype recv_type, int root,
    MPI_Comm comm);
```

Root ռանգով պրոցեսը բաշխում է send_buf–ի պարունակությունը մյուս բոլոր պրոցեսների միջև: Send_buf -ի

պարունակությունը բաժանվում է p հատ սեգմենտների, որոնցից յուրաքանչյուրը պարունակում է send_count հատ տարր: Առաջին սեգմենտը հանձնվում է 0-րդ պրոցեսին, երկրորդը՝ առաջին պրոցեսին և այդպես շարունակ: Միայն root պրոցեսում send-ի արգումենտները ունեն արժեք.

```
int MPI_Allgather(void* send_buf, int send_count,MPI_Datatype sen
d_type,
        void* recv_buf, int recv_count, MPI_Datatype recv_type,
MPI_comm comm);
```

MPI_Allgather()-ը հավաքում է յուրաքանչյուր պրոցեսի send_buf-երի պարունակությունները: Կանչի արդյունքը համեմատենք MPI_Gather() –ի p հատ հաջորդական կանչերի հետ, որոնցից յուրաքանչյուրում root պրոցեսն ունի սեփական ռանգը.

```
    int MPI_Allreduce(void* operand, void* result,
int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
```

MPI_Allreduce()-ը op ռեդուկցիայի գործողության արդյունքը յուրաքանչյուր պրոցեսի result բուֆերում:

Լրացուցիչ առաջադրանքներ (ուսանողների ինքնուրույն աշխատանք)

- Իրականացնել հաջորդականության առավելագույն և նվազագույն արժեքների ստացման զուգահեռ ալգորիթմները: Խնդիրն իրականացնել <կետ-կետ> փոխանցումների և կոլեկտիվ փոխանցումների միջոցով:

- Իրականացնել արագ և Շելլի տեսակավորման զուգահեռ ալգորիթմները: Կատարել համեմատական վերլուծություն այդ ալգորիթմների միջև:
- Վերը դիտարկված ալգորիթմներում գնահատել հաշվումների հնարավոր արագացումը պրոցեսորների թվի ավելացման դեպքում (ծրագիրն աշխատեցնել տարբեր քանակի պրոցեսորների վրա:

Հարցեր ինքնաստուգման համար

1. Որո՞նք են կոլեկտիվ ֆունկցիաները: Դրանց առավելությունները և կիրառման սկզբունքները:
2. Ինչպե՞ս ապահովել զուգահեռ մշակման կազմակերպման ընթացքում համակարգի բոլոր պրոցեսորների հավասարապես ծանրաբեռնվածությունը:
3. Բերել լավ զուգահեռացվող ալգորիթմների օրինակներ: Ո՞րն է իդեալականին մոտ զուգահեռացման պայմանը:

Լաբորատոր աշխատանք 5: Մատրիցների բազմապատկում: Հաջորդական և զուգահեռ տարբերակներ:

Աշխատանքի նպատակը կլաստերի հաշվողական հանգույցների միջև տվյալների բաշխման մեթոդների հստակեցումն է, ստանդարտ ալգորիթմների իրականացման համար MPI-ծրագրեր ստեղծելու համար գիտելիքների և հմտությունների ձեռքբերումը:

Տեսական մաս: Մեծ չափերի (հազար, հարյուր հազար) մատրիցների հետ գործողությունները շատ թվային մեթոդների հիմքն են (օրինակ, վերջավոր տարրերի մեթոդը): Միայն մեկ 3000x3000 double-թվերով քառակուսաձև մատրիցը օպերատիվ հիշողությունից պահանջում է 72 Մբայթ (ենթադրում ենք, որ մատրիցը լրիվ-հագեցած է, այսինքն, զրոյական տարրերի քանակը քիչ է, այլապես կպահանջվեն զրոյական տարրերի պահման հատուկ մեթոդներ): Պարզ է, որ այս դեպքում ծրագրավորողը պարտավոր է կլաստերի պրոցեսորների միջև բաշխել ոչ միայն հաշվողական պրոցեդուրաները, այլ նաև տվյալներ (նույն մատրիցները): Մատրիցների պահման համար սկավառակային հիշողության օգտագործումը չափազանց դանդաղեցնում է հաշվողական պրոցեդուրաները:

Մատրիցային բազմապատկում: Ամենապարզ մատրիցային գործողություններից է բազմապատկումը: Հայտնի է, որ [C]=[A]×[B] մատրիցների բազմապատկումից ստացված մատրիցի տարրերը հաշվարկվում են հետևյալ կերպ (ստանդարտ մեթոդ).

$$c_{ij} = \sum_{r=1}^{n} a_{ir} b_{rj} \quad (i = 1, 2, \ldots m;\ j = 1, 2, \ldots q).$$

Ստանդարտ մեթոդով (օգտագործվում է մեկ պրոցեսորով հաջորդաբար հաշվարկվող մեթոդը) մատրիցների բազմապատկման MM_SER.C պարզագույն ծրագիրը հետևյալն է.

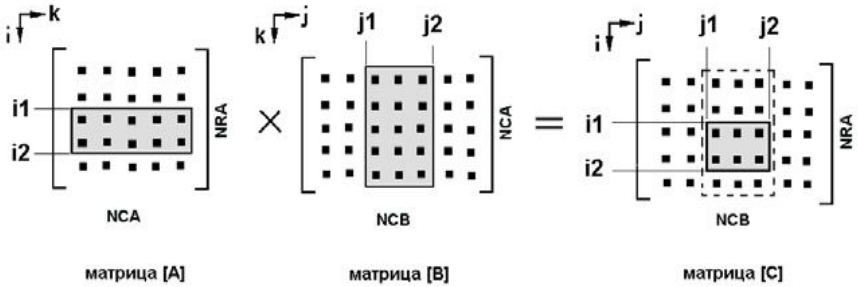
```
// source code of MM_SER.C program
#include <stdio.h>
#include <sys/timeb.h> // for ftime
double f_time(void); /* define real time by ftime function */
#include “f_time.c”

#define NRA 3000 /* number of rows in matrix A */
#define NCA 3000 /* number of columns in matrix A */
#define NCB 10 /* number of columns in matrix B */

int main(int argc, char *argv[])
{
int i, j, k; /* indexes */
double a[NRA][NCA], /* matrix A to be multiplied */
b[NCA][NCB], /* matrix B to be multiplied */
c[NRA][NCB], /* result matrix C */
t1,t2; // time’s momemts

/* Initialize A, B, and C matrices */
for (i=0; i<NRA; i++)
        for (j=0; j<NCA; j++)
                a[i][j] = i+j;

for (i=0; i<NCA; i++)
        for (j=0; j<NCB; j++)
                b[i][j] = i*j;

for(i=0; i<NRA; i++)
```

```
            for(j=0; j<NCB; j++)
                        c[i][j] = 0.0;

t1=f_time(); // get start time's moment
/* Perform matrix multiply */
for(i=0; i<NRA; i++)
            for(j=0; j<NCB; j++)
                        for(k=0; k<NCA; k++)
                                    c[i][j] += a[i][k] * b[k][j];

t2=f_time(); // get ended time's moment

printf ("Multiplay time= %.3lf sec\n\n", t2-t1);
printf("Here is the result matrix:\n");
for (i=0; i<NRA; i++)
{
            printf("\n");
            for (j=0; j<NCB; j++)
                        printf("%6.2f ", c[i][j]);
}
printf ("\n");
} // end of MM_SER.C program
```

Տեսականորեն (առանց հաշվի առնելու տվյալների փոխանակման համար պահանջվող ժամանակը) մատրիցների բազմապատկման խնդիրը զուգահեռացվում է իդեալական ձևով (ելքային տվյալները հաշվարկվում են մեկ միասնական ալգորիթմով՝ մուտքային տվյալների հիման վրա և հաշվարկման ընթացքում վերջիններիս չեն փոփոխում: Սակայն խնդրի պրակտիկ զուգահեռացման ժամանակ պետք է հաշվի առնել ինչպես տվյալների փոխանակման, այնպես էլ

հաշվարկները հանգույցների միջև բաշխելու վրա ծախսած ժամանակի կորուստները:

Ժապավենային(երիզային) ալգորիթմ: Տվյալների բաժանման ժապավենային սխեմայի դեպքում սկզբնական մատրիցները բաժանվում են հորիզոնական (A մատրիցի համար) և ուղղահայաց (B մատրիցի համար) երիզների:



Նկ.15.-Մատրիցների բազմապատկան երիզային սխեմա (ընդգծված են սկզբնական մատրիցների ուղարկվող երիզները և ստացվող մատրիցի հաշվարկված բլոկը)

```
// source code of MM_MPI_2.C program
// Ros Leibensperger / Blaise Barney. Converted to MPI: George L.
#include “mpi.h”
#include <stdio.h>

#define NRA 3000 /* number of rows in matrix A */
#define NCA 3000 /* number of columns in matrix A */
#define NCB 10 /* number of columns in matrix B */
#define MASTER 0 /* taskid of MASTER task */
#define FROM_MASTER 1 /* setting a message type */
#define FROM_WORKER 2 /* setting a message type */
#define M_C_W MPI_COMM_WORLD
int main(int argc, char *argv[])
```

```
{
int numtasks, /* number of tasks in partition */
taskid, /* a task identifier */
numworkers, /* number of worker tasks */
source, /* task id of message source */
dest, /* task id of message destination */
rows, /* rows of matrix A sent to each worker */
averow, extra, offset, /* used to determine rows sent to each worker */
i, j, k, rc; /* indexes */

double a[NRA][NCA], /* matrix A to be multiplied */
b[NCA][NCB], /* matrix B to be multiplied */
c[NRA][NCB], /* result matrix C */
t1,t2; // time's momemts

MPI_Status status;
rc = MPI_Init(&argc,&argv);
rc|= MPI_Comm_size(M_C_W, &numtasks);
rc|= MPI_Comm_rank(M_C_W, &taskid);

if (rc != MPI_SUCCESS)
        printf ("error initializing MPI and obtaining task ID
                information\n");
else
        printf ("task ID = %d\n", taskid);

numworkers = numtasks-1;

/************************** master task ********************************/
if (taskid == MASTER)
{
        printf("Number of worker tasks = %d\n",numworkers);
        for (i=0; i<NRA; i++)
```

```
            for (j=0; j<NCA; j++)
                        a[i][j]= i+j;

for (i=0; i<NCA; i++)
            for (j=0; j<NCB; j++)
                        b[i][j] = i*j;

/* send matrix data to the worker tasks */
averow = NRA/numworkers;
extra = NRA%numworkers;
offset = 0;
t1=MPI_Wtime(); // get start time's moment

for (dest=1; dest<=numworkers; dest++)
{
            if(dest <= extra)
                        rows = averow + 1;
            else
                        rows = averow;

            rows = (dest <= extra) ? averow+1 : averow;
            printf("...sending %d rows to task %d\n", rows,
                        dest);

            MPI_Send(&offset, 1, MPI_INT, dest,
                                    FROM_MASTER, M_C_W);
            MPI_Send(&rows, 1, MPI_INT, dest,
                                    FROM_MASTER, M_C_W);
            MPI_Send(&a[offset][0], rows*NCA,
            MPI_DOUBLE, dest, FROM_MASTER,M_C_W);
            MPI_Send(&b, NCA*NCB, MPI_DOUBLE, dest,
                        FROM_MASTER, M_C_W);
            offset += rows;
```

```
        }

        /* wait for results from all worker tasks */
        for (source=1; source<=numworkers; i++)
        {
                MPI_Recv(&offset, 1, MPI_INT, source,
                        FROM_WORKER, M_C_W, &status);
                MPI_Recv(&rows, 1, MPI_INT, source,
                      FROM_WORKER, M_C_W, &status);
                    MPI_Recv(&c[offset][0], rows*NCB,
                    MPI_DOUBLE, source,
                    FROM_WORKER,M_C_W, &status);
        }

        t2=MPI_Wtime(); // get ended time's momemt
        printf ("Multiply time= %.3lf sec\n\n", t2-t1);
        printf("Here is the result matrix:\n");
        for (i=0; i<NRA; i++)
        {
                printf("\n");
                for (j=0; j<NCB; j++)
                printf("%6.2f ", c[i][j]);
        }
        printf ("\n");
}

/************************* worker task ********************************/
if (taskid > MASTER)
{
        MPI_Recv(&offset, 1, MPI_INT, MASTER,
                FROM_MASTER, M_C_W, &status);
        MPI_Recv(&rows, 1, MPI_INT, MASTER,
                FROM_MASTER, M_C_W, &status);
```

```
        MPI_Recv(&a, rows*NCA, MPI_DOUBLE, MASTER,
                 FROM_MASTER,M_C_W, &status);
        MPI_Recv(&b, NCA*NCB, MPI_DOUBLE, MASTER,
                 FROM_MASTER,M_C_W, &status);

        for (k=0; k<NCB; k++)
        {
                for (i=0; i<rows; i++)
                {
                        c[i][k] = 0.0;
                        for (j=0; j<NCA; j++)
                                c[i][k] += a[i][j] * b[j][k];
                        }
        }

        MPI_Send(&a_offset, 1, MPI_INT, MASTER,
                 FROM_WORKER, M_C_W);
        MPI_Send(&a_rows, 1, MPI_INT, MASTER,
                 FROM_WORKER, M_C_W);
        MPI_Send(&c, a_rows*NCB, MPI_DOUBLE, MASTER,
                 FROM_WORKER,M_C_W);
}

MPI_Finalize();
} // end of MM_MPI_2.C program
```

[A] մատրիցի տողերը բաշխվում են numworkers պրոցեսների միջև՝ rows-ական քանակներով: Յուրաքանչյուր պրոցեսի՝ dest=1...numworkers, MPI_Send(&a[offset][0], rows*NCA, MPI_DOUBLE, dest, FROM_MASTER, MPI_COMM_WORLD) ֆունկցիայի կանչի միջոցով փոխանցվում են rows տող (rows=averow, եթե NRA

ամբողջապես բաժանվում է numworkers-ի, և rows=averow+1, հակառակ դեպքում, offset – առաջին փոխանցվող տողի համարն է): [B] մատրիցը փոխանցվում է բոլոր numworkers պրոցեսներին՝ MPI_Send(&b, NCA*NCB, MPI_DOUBLE, dest, FROM_MASTER, MPI_COMM_WORLD) ֆունկցիայի միջոցով: Երբ որ բոլոր պրոցեսները ստանում են տվյալները, սկսվում է բազմապատկումը արտաքին ցիկլով k=1...NCA (այսինքն [A]–ի տողերով և [B]-ի սյուներով և i=1...rows ներքին ցիկլով (այսինքն [A] и [C]-ի տողերի փոխանցված մասերով)).

```
for (k=0; k<NCB; k++)
{
        for (i=0; i<rows; i++)
        // … հանգույցի վրա ներկա են միայն rows տողեր [A]
        // մատրիցից
        {
                c[i][k] = 0.0;
                for (j=0; j<NCA; j++)
                        c[i][k] += a[i][j] * b[j][k];
        }
   }
```

Աշխատանքի վերջում բոլոր numworkers պրոցեսները գլխավոր պրոցեսին են վերադարձնում են [C]-ի rows*NCB տող՝ կանչելով MPI_Send(&c, rows*NCB, MPI_DOUBLE, MASTER, FROM_MASTER, MPI_COMM_WORLD) ֆունկցիան, ընդ որում նախապես առաջին տողի offset համարը փոխանցվում է MPI_Send(&offset, 1, MPI_INT, MASTER, FROM_MASTER, MPI_COMM_WORLD) –ի միջոցով, իսկ rows տողերի քանակը՝ MPI_Send(&rows,1, MPI_INT, MASTER, FROM_MASTER, MPI_COMM_WORLD)-ի միջոցով: MASTER-պրոցեսն ընդունում է այդ տվյալները MPI_Recv(&c[offset][0],

rows*NCB, MPI_DOUBLE, source, FROM_WORKER, MPI_COMM_WORLD, &status) կանչով:

Ֆոքսի ալգորիթմը: Երիզային ալգորիթմի հետագա (տվյալների փոխանցումների քանակի նվազեցման ուղղությամբ) զարգացումն է Ֆոքսի[26] ալգորիթմը և Քենոնի ալգորիթմը (այս դեպքում պրոցեսներին ուղարկվում են ոչ թե երիզներ, այլ սկզբնական մատրիցների ուղղանկյունաձև բլոկներ, այս դեպքում բարդացում է համարվում անհրաժեշտ լրացուցիչ միջպրոցեսային փոխանակումը, հակառակ դեպքում [A]-ի և [B]-ի ուղղանկյունաձև բլոկների հիման վրա հնարավոր չէ հաշվարկել հենց նույն տիպի [C]-ի բլոկ. անհրաժեշտ է հաշվումների ցիկլիկ տեղաշարժ):

Տվյալների բլոկային ներկայացման դեպքում մատրիցային բազմապատկման զուգահեռ հաշվողական սխեման կարելի է ներկայացնել ամենապարզ տեսքով, եթե հաշվողական ցանցի տոպոլոգիան ունի ուղղանկյունաձև ճաղերի (ցանցի) տեսք (եթե ցանցի իրական տոպոլոգիան ունի այլ տեսք, ապա ցանցը կարելի է տրամաբանական մակարդակով ներկայացնել ճաղի տեսքով): Բլոկային ներկայացված մատրիցների զուգահեռ մեթոդների հիմնական դրույթները հետևյալն են.

- Ճաղի յուրաքանչյուր պրոցեսոր պատասխանատու է C մատրիցի միայն մեկ բլոկի հաշվարկման համար:
- Հաշվարկների ընթացքում պրոցեսորներից յուրաքանչյուրի վրա տեղակայվում է սկզբնական A և B մատրիցների մեկական բլոկ:
- Ալգորիթմների իտերացիաների կատարման ժամանակ A մատրիցի բլոկները հաջորդաբար տեղաշարժվում են

պրոցեսորային ճաղի տողի երկայնքով, իսկ B մատրիցի բլոկները՝ ճաղի սյուների երկայնքով:

- Հաշվումների արդյունքում պրոցեսորներից յուրաքանչյուրի վրա հաշվարկվում է C մատրիցի բլոկը, ընդ որում, ալգորիթմի իտերացիաների ընդհանուր քանակը p է (որտեղ p –ն պրոցեսորների քանակն է):

Այժմ դիտարկենք մատրիցների բազմապատկման զուգահեռ ալգորիթմի հիմնական քայլերը: Դիցուք պրոցեսորները միավորված են *Q չափանի* քառակուսային ցանցի միջոցով: Ալգորիթմը հետևյալ քայլերի հաջորդականությունն է:

for(step=0; step<Q; step++)
{

1) Պրոցեսներից յուրաքանչյուրի տողից ընտրել [A] մատրիցի ենթամատրից (r-րդ տողում գտնվում է Aru ենթամատրիցը, որտեղ u=(r+step) mod Q):

2) Յուրաքանչյուր տողի պրոցեսի համար այդ նույն տողի բոլոր մյուս պրոցեսներին ուղարկել ընտրված մատրիցը:

3) Յուրաքանչյուր պրոցես բազմապատկում է [A] մատրիցի ստացված ենթամատրիցը [B] մատրիցի ենթամատրիցի հետ, որը գտնվում էր պրոցեսի Օպերատիվ հիշողության մեջ:

4) Յուրաքանչյուր պրոցեսի համար [B] մատրիցի ենթամատրիցն ուղարկել այն պրոցեսին, որը գտնվում է ճաղում ավելի բարձր դիրքում (առաջին տողից կատարվում է փոխանցում վերջին տողին):

}

Անհրաժեշտ սարքավորումները. UNIX-համատեղելիությամբ ՕՀ-ով ղեկավարվող հաշվողական կլաստեր, նրա վրա տեղադրված MPI միջավայր, ծրագրավորողի աշխատանքային հարթակ ՝ օգտագործողի խնդիրները ղեկավարելու համար:

Աշխատանքի անցկացման կարգը. ուսանողը պատրաստում է MPI-ծրագրի նախնական տեքստը, կոմպիլացնում է, աշխատեցնում է և դասախոսի կարգադրությամբ վերլուծում ելքային տվյալները:

Ուսանողի անհատական աշխատանք.

- Տրված հաշվողական համակարգի համար որոշել (կոմպիլյատորի հնարավորություններով կամ օպերատիվ հիշողության չափսով սահմանափակված) [A] մատրիցի առավելագույն չափսը՝ [B] մատրից-վեկտորով բազմապատկելիս (զուգահեռ և հաջորդական տարբերակները պրոցեսորների N=2,3,4,5… քանակների դեպում):
- Պարզել MM_MPI_2 արագագործության կախվածությունը (ալգորիթմի կատարման ժամանակին հակադարձ մեծությունը) պրոցեսորների N=3,4,5… թվից (N=2 դեպքում ծրագրի արտադրողականությունը ընդունել հավասար 1-ի):

Լրացուցիչ առաջադրանքներ (ուսանողի ինքնուրույն աշխատանք)

- MM_MPI_2 ծրագիրը վերափոխել այնպես, որ [A], [B] և [C] մատրիցները չլինեն յուրաքանչյուր պրոցեսի օպերատիվ հիշողության մեջ, այլ միայն գլխավորի մեջ:

- Իրականացնել մատրիցների բազմապատկման ծրագիր՝ յուրաքանչյուր պրոցեսորով կատարելով.
    - $a_{ik} \times b_{kj}$ տարրերի բազմապատկում՝ ստանալու համար c_{ij} մասնակի գումար և ապա մասնակի գումարների գումարում գլխավոր պրոցեսով՝ ստանալու համար c_{ij} (MM_MPI_0.C ծրագիրը):
    - [A] մատրիցի տողերի բազմապատկում [B]-ի սյուներով, այնուհետև դրանք գումարում յուրաքանչյուր պրոցեսորի վրա՝ ստանալու համար c_{ij} վերջնական արժեքը (MM_MPI_1.C ծրագիրը). Այս դեպքում պրոցեսորների վրա գործողությունների կատարման ալգորիթմն ի՞նչ առանձնահատկություն կունենա: Պրոցեսորների տարբեր քանակությունների դեպքում այս ծրագրերի արագագործությունը համեմատել MM_MPI_2.C ծրագրի հետ:
- Ֆոքսի ալգորիթմի արդյունավետությունը համեմատել MM_MPI_2.C ծրագրի հետ (որոշել պրոցեսորների տարբեր քանակների դեպքում և տարբեր մեծությունների մատրիցներ բազմապատկելիս ծրագրի կատարման ժամանակը)

Հարցեր ինքնաստուգման համար.

1. Ո՞ր դեպքում մատրիցների բազմապատկման ժամանակը կլինի ավելի երկար. N=2 պրոցեսորների քանակի դեպքում MM_SER-ինը թե՞ MM_MPI_0-ն:

2. Ալգորիթմների զուգահեռացման դեպքում հաշվումների և տվյալների բլոկների բաշխման ռազմավարություն մշակելու ինչպիսի՞ դատողություններից է պետք օգտվել:

3. Ինչպե՞ս է ավելի նպատակահարմար բաշխել սկզբնական և հաշվարկված մատրիցների բլոկները՝ մատրիցների արտադրյալը գտնելիս, մատրիցը սկալյարով բազմապատկելիս կամ բաժանելիս, մատրիցները տրասպոնացնելիս:

4. Առաջարկել և հիմնավորել հաշվողական հանգույցների միջև մեծ չափսերի մատրիցների բաշխման ավելի արդյունավետ (վերևում ցույց տրվածի համեմատությամբ) ռազմավարություն (անպայման հաշվի առնել մատրիցի տողերի ու սյուների թվի և պրոցեսորների թվի ոչ պատիկությունը):

Գրականություն

1. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. – СПб.: БХВ-Петербург, 2004. –608 с.

2. Антонов А.С. Введение в параллельное программирование (методическое пособие). –М.: НИВЦ МГУ, 2002, -69 с. (http://pilger.mgapi.edu/metods/1441/antonov.pdf)

3. Антонов А.С. Параллельное программирование с использованием технологии MPI (учебное пособие). – М.: НИВЦ МГУ, 2004, -72 с. (http://parallel.ru/tech/tech_dev/MPI,http://pilger.mgapi.edu/metods/1441/mpi_book.pdf

4. Шпаковский Г.И., Серикова Н.В. Программирование для многопроцессорных систем в стандарте MPI. –Минск: БГУ, 2002. –325 с. (http://www.cluster.bsu.by/download/book_PDF.pdf, http://pilger.mgapi.edu/metods/1441/pos_mpi.pdf)

5. Amdahl G. Validity of the single-processor approach to achieving large-scale computing capabilities. // Proc AFIPS Conf., AFIPS Press – 1967. V. 30. P 483.

6. Bailey D., Harris T., Sahpir W., van der Wijngaart, Woo A., Yarrow M. The NAS Parallel Benchmarks 2.0. // Report NAS-95-020, Dec 1995. http://science.nas.nasa.gov/Software/NPB

7. Dongarra J., Luszczek P., Petitet A. The Linpack Benchmark: Past, Present, and Future. July 2002.

8. Petitet A. Whaley C. Dongarra J. and Cleary A. HPL – A Portable Implementation of the High Performance Linpack Benchmark for Distributed Memory Computers. //

Innovative Computing Laboratory, September 2000. http://www.netlib.org/benchmark/hpl

9. Geist G. and Romine C. LU Factorization Algorithms on Distributed-Memory Multiprocessor Architectures. // SIAM Journal on Scientific and Statistical Computing, Vol. 9, pp. 639-649, 1988.

10. «PMB» http://www.pallas.com

11. «MPI» http://www.mpi-forum.org

12. Argonne National Lab. http://www.mcs.anl.gov

13. Ohio Supercomputer Center http://www.osc.edu

14. Notre-Damme University, http://www.lsc.nd.edu

15. University of Coimbra, Portuga lhttp://dsg.dei.uc.pt/wmpi

16. CHIMP/MPI (Edinburgh Parallel Computing Centre, http://www.epcc.ed.ac.uk)

17. ScaLAPACK http://www.netlib.org/scalapack

18. TotalView (http://www.dolphinics.com)

19. Гюрджян Микаел Казаросович. Диссертация. Некоторые вопросы организации параллельных вычислений в многопроцессорных вычислительных системах кластерного типа. 2004.

20. H. Astsatryan, T. Grigoryan, M. Gyurjyan, V. Sahakyan, Yu. Shoukourian, Development of Web Environment for Efficient Exploitation of Linux Cluster Computing Resources. Proceedings of the Seventh International meeting on VECPAR'06, July 10-13, 2006, Rio de Janeiro, Brazil.

21. V. Sahakyan, H. Astsatryan, M. Gyurjyan, Armenia Entered a New Era of Computations, Gitutyan Ashxarhum

(In the World of Science) Journal, pp. 19-28, N2, 2007, ISSN 1829-0345

22. M. Gyurjyan, D. Khachatryan, Software Package for Efficient Exploitation (SPEE) of High Performance Computing Cluster, Proceedings of the Fifth International Conference on Computer Science and Information Technologies (CSIT '2005), ISBN: 5-8080-0631-7, pp. 363-366, September 19-23, 2005, Yerevan, Armenia.

23. Foster I. Designing and Building Parallel Programs. // Addison-Wesley. 1995. P 430.

24. Snir M., Otto S., Huss-Lederman S., Walker D., Dongarra J. MPI: The Complete Reference. // Massachusetts Institute of Technology, 1996. http://www.netlib.org/utk/papers/mpi-book/mpi-book.html

25. Корнеев В. Параллельное программирование в MPI. Москва-Ижевск: Институт компьютерных исследований. 2003. 303 с.

26. Geoffrey Fox, et al., Solving Problems on Concurrent Processors, Englewood Cliffs, NJ, Prentice-Hall, 1998

27. M.Gyurjyan, V. Sahakyan, Queues modelling of the multimachine computing system. Mathematical Problems of Computer Science, Transactions of IIAP NAS RA, Volume 23, pp. 166-174, 2004, Yerevan.

28. A. Poghosyan, L. Arsenyan, H. Astsatryan, M. Gyurjyan, H. Keropyan and A. Shahinyan, "NAMD Package Benchmarking on the Base of Armenian Grid Infrastructure," Communications and Network, Vol. 4 No. 1, 2012, pp. 34-40.

Միքայել Ղազարոսի Գյուրջյան

ԶՈՒԳԱՀԵՌ ԾՐԱԳՐԱՎՈՐՈՒՄ MPI ՄԻՋԱՎԱՅՐՈՒՄ

Ուսումնական ձեռնարկ

Խմբագիր՝ **Ն.Ա.Խաչատրյան**