**Департамент Информатика**

**Програма Информатика**

**ДИПЛОМНА РАБОТА**

**ТЕМА:**

**Среда за паралелно програмиране на Java с MPI.**

**Дипломат:**

**Николай Николов**

**F78397**

**Ръководител:**

**Проф. д-р Красимир Манев**

София 2019

Съдържание

[1. Увод 5](#_Toc29218220)

[1.1. Цел 5](#_Toc29218221)

[1.2. Задачи 5](#_Toc29218222)

[1.2.1. Изследване 5](#_Toc29218223)

[1.2.2. Реализация 5](#_Toc29218224)

[1.2.3. Тестове 6](#_Toc29218225)

[1.3. Структура на работата 6](#_Toc29218226)

[2. Терминология 7](#_Toc29218227)

[2.1. Паралелно изчисление (Concurrent Computing) 7](#_Toc29218228)

[2.2. Процес 7](#_Toc29218229)

[2.2.1. Общи характеристики 7](#_Toc29218230)

[2.2.2. Memory Layout 8](#_Toc29218231)

[2.3. Паралелна комуникация 9](#_Toc29218232)

[2.4. Inter Process Communication (IPC) 9](#_Toc29218233)

[2.4.1. Споделени файлове 9](#_Toc29218234)

[2.4.2. Изпращане на съобщения (Message Passing) 9](#_Toc29218235)

[2.4.3. Опашка за съобщения (message queues) 10](#_Toc29218236)

[2.4.4. Споделяне на памет 10](#_Toc29218237)

[2.4.5. Remote Process Calls (RPC) 11](#_Toc29218238)

[2.4.6. Синхронизация 11](#_Toc29218239)

[2.4.7. Network 12](#_Toc29218240)

[2.5. IPC и MPI 12](#_Toc29218241)

[2.6. MPI и Android 13](#_Toc29218242)

[2.6.1. Интеграция в Android 13](#_Toc29218243)

[2.6.2. Сигурност 13](#_Toc29218244)

[2.6.3. Имплементация и използване 13](#_Toc29218245)

[2.7. Свързване на Java със C (Java Binding) 13](#_Toc29218246)

[2.8. Прецизиране на целта и задачите 13](#_Toc29218247)

[3. MPI 15](#_Toc29218248)

[3.1. Увод 15](#_Toc29218249)

[3.2. Point-to-Point комуникация 16](#_Toc29218250)

[3.3. Datatypes 17](#_Toc29218251)

[3.3.1. Вектори 17](#_Toc29218252)

[3.3.2. Структури 18](#_Toc29218253)

[3.4. Колективна комуникация 19](#_Toc29218254)

[3.4.1. Операцията broadcast 20](#_Toc29218255)

[3.4.2. Операциите gather/scatter 20](#_Toc29218256)

[3.4.3. Операцията allgather 22](#_Toc29218257)

[3.5. Групи, контекст и комуникатори 22](#_Toc29218258)

[3.6. Виртуални топологии 24](#_Toc29218259)

[ Декартова топология 24](#_Toc29218260)

[ Графова топология 24](#_Toc29218261)

[3.7. Едностранна комуникация (one-side communication) 24](#_Toc29218262)

[3.7.1. Модел на паметта 25](#_Toc29218263)

[3.7.2. Модели за синхронизация 25](#_Toc29218264)

[3.8. Паралелни входно/изходни операции 26](#_Toc29218265)

[4. MPICH 28](#_Toc29218266)

[4.1. Процес мениджмънт (Hydra) 28](#_Toc29218267)

[4.1.1. Потребителски интерфейс – UI (mpiexec) 28](#_Toc29218268)

[4.1.2. Ядро за управление на ресурсите (Resource Management Kernel RMK) 28](#_Toc29218269)

[4.1.3. Процес менажер 28](#_Toc29218270)

[4.1.4. Процес мениджър прокси 29](#_Toc29218271)

[4.1.5. „Bootstrap server“ (ssh, fork, pbs, slurm, sge) 29](#_Toc29218272)

[4.1.6. „I/O demux engine“ 29](#_Toc29218273)

[4.2. Поток на изпълнение 29](#_Toc29218274)

[4.3. Mpich архитектура 30](#_Toc29218275)

[4.3.1. Процеси, групи и виртуални връзки (Virtual Connections - VC) 31](#_Toc29218276)

[4.3.2. Abstraction Device Interface (ADI3) 31](#_Toc29218277)

[4.3.3. CH3 31](#_Toc29218278)

[4.3.4. CH4 32](#_Toc29218279)

[4.3.5. Канали за комуникация 32](#_Toc29218280)

[4.3.6. ROMIO 33](#_Toc29218281)

[5. JNI 34](#_Toc29218282)

[5.1. Дизайн 34](#_Toc29218283)

[5.2. Типове 35](#_Toc29218284)

[5.3. Достъп от C/C++ към Java 37](#_Toc29218285)

[5.4. Функции 38](#_Toc29218286)

[*5.5.* АPI за извиквания (The Invocation API) 40](#_Toc29218287)

[5.5.1. JNI\_OnLoad 41](#_Toc29218288)

[5.5.2. JNI\_OnUnload 41](#_Toc29218289)

[5.5.3. JavaVM 41](#_Toc29218290)

[5.6. Опасности 42](#_Toc29218291)

[6. Java-Mpi спецификация 43](#_Toc29218292)

[6.1. Въведение 43](#_Toc29218293)

[6.2. Point-To-Point Комуникация 43](#_Toc29218294)

[6.3. Типове Данни 47](#_Toc29218295)

[6.4. Колективна Комуникация 52](#_Toc29218296)

[6.5. Групи, Контекст, Комуникатори 55](#_Toc29218297)

[6.6. Виртуални Топологии 57](#_Toc29218298)

[6.7. Управления на средата 59](#_Toc29218299)

[6.8. UI компонент, mpiexec 59](#_Toc29218300)

[6.9. Информационен обект 60](#_Toc29218301)

[6.10. Създаване на процеси и тяхното управление 60](#_Toc29218302)

[6.11. Едностранна комуникация 63](#_Toc29218303)

[6.12. Входно/Изходни операции 66](#_Toc29218304)

[6.13. Обобщение 67](#_Toc29218305)

[6.14. Подобен софтуер 68](#_Toc29218306)

[7. Имплементация 70](#_Toc29218307)

[7.1. Обекти 70](#_Toc29218308)

[7.2. Mpich промени 72](#_Toc29218309)

[7.3. Типове 76](#_Toc29218310)

[7.4. Потребителски типове 77](#_Toc29218311)

[7.5. Комуникации 79](#_Toc29218312)

[7.5.1. Методи за достъп на данни от Java към C и обратно 80](#_Toc29218313)

[7.5.2. Блокиращи методи за комуникация между два процеса 82](#_Toc29218314)

[7.5.3. Неблокиращи методи 82](#_Toc29218315)

[7.6. „persistent“ комуникация 83](#_Toc29218316)

[7.7. Обобщение 84](#_Toc29218317)

[8. Тестове 85](#_Toc29218318)

[8.1. Увод 85](#_Toc29218319)

[8.2. Smoke Tests 85](#_Toc29218320)

[8.3. Максимален път в Даг 85](#_Toc29218321)

[9. Заключение 87](#_Toc29218322)

[10. Използвана литература 88](#_Toc29218323)

# Увод

Паралелното програмиране е един от основните методи за решаване на изчислителни задачи. Голям набор от алгоритми и методи използват този подход за извършване на изчисления в науката, промишлеността, анализ на данни и много други области изискващи много пресмятания. Но ползата от паралелните изчисления не се изчерпва с това. Паралелният подход може да се използва за извършването на произволна работа. Например, градинар може да окоси градина за N часа, но двама градинари ще го направят за N/2 часа. Така, че както за компютърни изчисления, така и за каквато и да е друга дейност или задача, може да се търси паралелно решение.

Разбира се, използването на паралелизъм не винаги води до намаляване на времето. Много зависи от вида на задачата и от това, какви са възможностите да бъде решавана паралелно. Добър пример за това е изработването на някакво изделие – например, лека кола. Определени части от колата могат да се изработя паралелно, но не и да се сглобят паралелно. Не може да се монтира двигателя, без да има на какво или да се тества скоростта, без да е приключило сглобяването. С две думи, много трябва де се внимава как една задача ще се разпредели и дали комуникацията между отделните паралелни компоненти е налице.

## Цел

Основния проблем, с който ще се занимаваме е **комуникацията между паралелно работещи Java процеси**. Затова, и целта на тази дипломна работа е да разработи **среда за паралелно програмиране на Java**. Тя ще ни позволи комуникация между два и повече процеса намиращи се на няколко компютъра свързани в мрежа. Имплементацията на този софтуер ще се състои от два основни компонента – Java програми и MPICH (библиотека за комуникация между процеси). Като краен резултат ще получим интерфейс за паралелно програмиране – библиотека базирана на тези компоненти с име java-mpi.

## Задачи

Задачите, с помощта на които ще реализираме целта могат да се разделят в три основни етапа – изследване, реализация и тестване.

### Изследване

Това е етап, който продължава дори по време на самата разработка. В него ще се запознаем с това, какво точно е процес, как може да се осъществи комуникацията между процеси. В рамките на тази задача ще трябва да се запознаем със стандарта **MPI (Message Passing Interface)** и **имплементацията му, *MPICH***.

Задачите, са:

* Запознаване със същността на проблема.
* Запознаване с терминологията.
* Изследване на проблема.
* Запознаване с MPI и MPICH.
* Изследване на това как може да използваме MPICH в Java.

### Реализация

След като вече сме запознати с термините и инструментите, които ще използваме е време да преминем към имплементацията на самият интерфейс. Тук задачата ще ни е да разработим въпросния интерфейс. Освен имплементация, ще имаме за цел да предоставим и спецификация на това как може да се използва разработеният софтуер.

### Тестове

Създаването на тази имплементация ще е безсмислено, ако не направим и съответни тестове. С тях ще проверим дали сме постигнали целта и дали сме решили проблема. Те ще ни дадат солидна основа за нашият извод.

## Структура на работата

В **Глава 2. Терминология** са представени основни термини, свързани с дипломанта работа. В нея е показана логическата връзка между различните компоненти от проекта. Много от тях няма да бъдат разгледани изцяло, защото са представени в следващи глави. В **Глава 3. MPI, стандарт за комуникация между процеси** са разгледани основите на стандарта MPI. Дадено е кратко описание на неговите възможности и предлаганата от стандарта функционалност. В **Глава 4.** **MPICH, библиотека имплементираща MPI** е представена архитектурата на имплементацията MPICH и е специфициран един от основните компоненти, които сме използвали за реализирането на java-mpi, а в **Глава 5. JNI** е представен друг много важен компонент, използван в дипломната работа – Java Native Interface.

**Глава 6. Java-Mpi-Спецификация** е първата от основните глави, посветени на разработването на java-mpi. В нея е специфициран програмно-приложният интерфейс, който е даден за ползване на потребителя. **Глава 7. Имплементация** описва архитектурата на проекта и дава подробности за основни моменти от имплементирането на софтуера.

В **Глава 8. Тестове** сме показали, че имплементацията е работоспособна и използваема. Това сме постигнали чрез тестове и измервания, които са описани тук.

**Заключението** е последната глава на работата. В нея сме резюмирали крайните резултати от проведените тестове и не на последно място, сме посочили някои възможности за приложения на java-mpi.

# Терминология

## Паралелно изчисление (Concurrent Computing)

*Паралелно изчисление* е такова изчисление, при което няколко задачи се решават **едновременно,** в една и съща времева рамка, вместо едно след друго. Може да се каже, че това е свойство на системата, на която се извършва изчислението. T.е., не всяка система за пресмятане поддържа подобна форма на изчисление. Системите, който го поддържат, се наричат още *многозадачни системи*. В повечето случай, ако не във всички, става дума за операционни системи (ОС), така че говорим за *многозадачни операционни системи*. Такива системи са Linux, Windows, FreeRTOS и други.

*Паралелно програмиране* и паралелно изчисление може да се разглеждат като синоними. Идеята на такъв вид изчисления е една задача да се раздели на няколко по-малки задачи, които при възможност да се изпълнявaт едновременно. Времето за паралелно изпълнение на цялата задача зависи много от същността ѝ. В зависимост от това ще имаме различно забързване, което може да се предвиди чрез оценяване на сложността на използвания алгоритъм (например, *O*(*n*), *O*(log *n*) и т.н.).

По-подробна информация за паралелното програмиране, многозадачните ОС и как те се имплементират, ще разгледаме в следващата глава. Сега ще обърнем внимание върху основния проблем на този вид изчисления, а именно – комуникацията между паралелно изпълнявани компоненти.

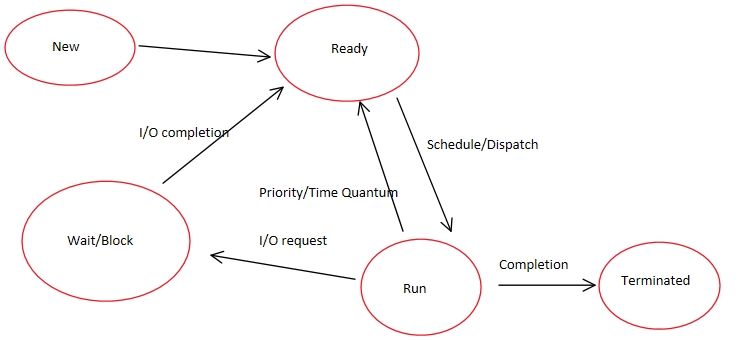
## Процес

### Общи характеристики

Един процес се дефинира като програма извикана за изпълнение. Ако в една многозадачна ОС, 10 потребителя използват програмата „*/usr/bin/python.exe*“, то за изпълнението на тази програма ще бъдат създадени 10 отделни процеса. Подобна аналогия може да се наблюдава и в обектно-ориентираното програмиране. В него обектът е инстанция на класа със свой собствени променливи и данни. Абсолютно същата е и връзката програма-процес. Един процес е инстанция на една програма, със своя собствена памет и характеристики.

* Състояние

Един процес може да има множество състояния. На Фиг. 2.1 са показани само основните.

  
Фиг. 2.1

* Нов процес (New)

Това е първото и входно състояние за един процес. В него системата подготвя себе си за по нататъшна обработка

* Готов процес (Ready)

В това състояние процесът чака да се освободи време за неговото изпълнение.

* Изпълняващ се процес (Run)

След като даден процес бъде избран да получи ресурс, той преминава в състояние на изпълнение. След като му свърши времето или трябва да изчака събитие, той преминава в състояние на изчакване.

* Чакащ/блокиран процес(Wait/Block)

Един процес се озовава в това състояние, когато се налага да изчака за ресурс или събитие. След като ресурсът се освободи или събитието се случи, той преминава обратно в състояние на готовност, сигнализирайки на планиращия алгоритъм, че е готов за изпълнение.

* Унищожен процес (Terminated)

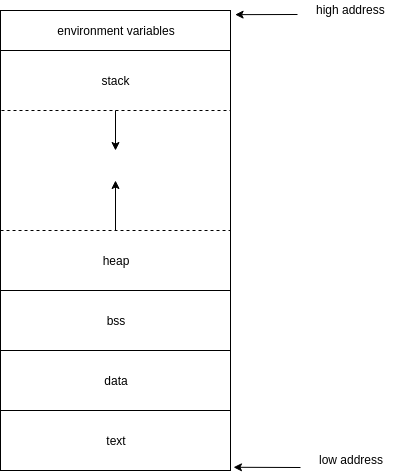
Един процес може да бъде унищожен от състояние на изпълнение, което означава, че работата му е приключила нормално или чрез външна намеса.

* Идентификатор на процеса

Всеки процес в системата се идентифицира с този номер. Той се определя от операционната система и се освобождава от нея за ново използване. Различни алгоритми и функционалности използват този идентификатор. Например системната функция на ОС “kill” го използва за локализиране на процеса, който трябва да бъде унищожен.

### Memory Layout

Основното разпределение на паметта (memory layout) на един процес може да се види на Фиг. 2.2. Всяка секция от нея е свързана пряко с изпълнението на само една програма.

  
Фиг. 2.2

* Environment variables

Това е секция намираща се в най-горното адресно пространство на процеса. В нея се запазват всички променливи на операционната система.

* Stack

Това е следващия сегмент. Използва се за запаметяване на важна информация за изпълнението на програмата в този процес. При всяко извикване на нова функция, се генерира „stack frame“, в който има адреси, променливи и други важни данни.

* Heap

Динамично заделяната памет на един процес. Всяка програма разполага с подобна и това е паметта, която един процес може да заделя за използване по-време на своето изпълнение.

* Bss (Block Stated by Symbol)

Съдържа статични глобални променливи, които не са инициализирани.

* Data

Това е секцията съдържаща всички инициализирани глобални променливи.

* Text

Последния сегмент от паметта на един процес. Той съдържа инструкциите за изпълнение. Обикновено се споделя между различни инстанции на една програма, тъй като е памет, която е само за четене (read-only) и се използва от всяка инстанция, няма смисъл да се заделя памет за повече от едно копие. Разликата между различните процеси, които използват една и съща програма е само в брояча на инструкции (instruction counter).

## Паралелна комуникация

Тук е моментът да отбележим, че за да имаме паралелно изпълняващи се процеси, не е задължително те да са на една и съща машина. Единствените условия, които трябва да са спазени са:

* обща времева рамка, както споменахме по рано;
* да са части от една и съща задача;
* да комуникират по между си.

Това означава, че една задача може да се изчислява върху повече от една машина. Grid- системите са изградени на този принцип, както и повечето *разпределени изчислителни системи*, при които изчислението се базира върху повече от една машина. Идеята е да се обединят множество компютърни ресурси от различно естество с цел решаване на една задача [Grid Computing].

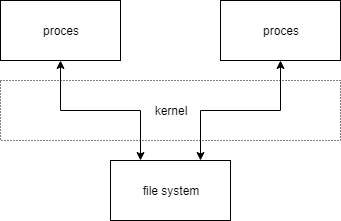
Точно *комуникацията* е един от основните проблеми, на който се търси решение. Той е основният проблем на паралелното програмиране. Такъв е защото, на практика няма паралелен алгоритъм, който не използва комуникация. С него обаче идва и набор от други проблеми, като *синхронизация*, откриване на грешки (debugging) в паралелните програми и разбира се, имплементирането му. Колкото по-лесно се използва една техника, толкова по-лесно е за един разработчик да се фокусира върху имплементацията на алгоритъма. Множество библиотеки се опитват да решат този и други проблеми свързани с комуникацията, на която ще се спрем в следващия раздел.

## Inter Process Communication (IPC)

*Inter Process Communication* (*IPC*) e набор от техники за комуникация между различни процеси или по-точно възможността един процес да предостави данни от своето адресно пространство на друг процес, независимо от това дали двата процеса се намират на една или повече машини.

### Споделени файлове

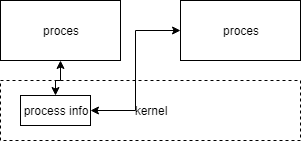
Това е вид комуникация, осъщест­вяваща се, по една или друга причина, през файловата система на ОС. За да може един процес да получи данни от друг процес, задължително трябва да мине през ядрото (Фиг. 2.3). Като обикновено, на прог­рамно ниво, комуникацията става през функции като ”*write*”, “*read*”, ”*lseek*” и прочие. Някаква форма на синхронизация е необходима при обновяване или писане във файла.

  
Фиг. 2.3

### Изпращане на съобщения (Message Passing)

В този вид механизми (Фиг. 2.4) влизат „*pipes*“, FIFOs и опашки за съобщения (message queues).

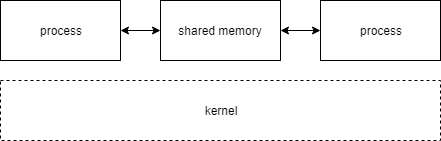
„*Pipes*“ и „*FIFOs*“ са механизми ориентирани по-скоро към поточното предаване на данни, отколкото към предаване на съобщения. Това, обаче, не означава, че имплементацията не го позволява. Реализацията им става чрез два дескриптора – числа наподобяващи на файлови дескриптори, като в случая с FIFOs това е точно така. От тях, единият се ползва за запис, а другият за четене. Двата метода изпълняват една и съща функция, разликата е в това, че механизмът „pipes“ не притежава голяма гъвкавост, няма идентификатор и комуникационният обект не може да се запазва. Докато това може да стане с „FIFOs“.

  
Фиг. 2.4

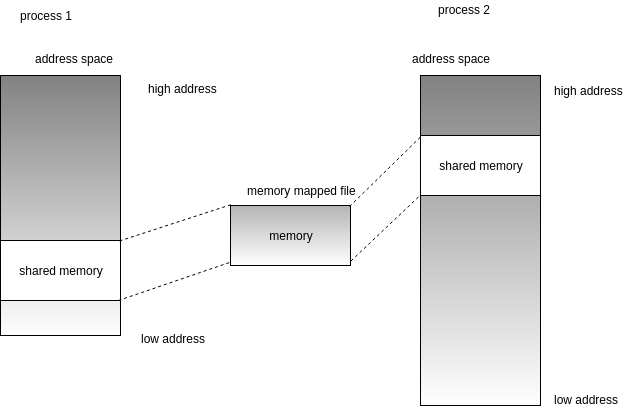
### Опашка за съобщения (message queues)

Това е двусвързан списък (опашка), от съобщения, като всяко съобщение разполага с набор от свойства – големина, приоритет и други. Опашката може да се ползва от повече от един процес, като всеки процес се бори за връзка. Идентификацията на този вид IPC комуникация е с име, като всеки процес има достъп до обекта чрез дескриптор.

### Споделяне на памет

  
Фиг. 2.5

Споделената памет (Фиг. 2.5), или още „*shared memory*”, е един от най-бързите методи за IPC комуникация. Имплемента­цията от пре­диш­ната група, изпращане на съобщения, обикновено изискват повече от две копирания, като това не включва копиранията из­върш­­­ващи се в приложението. Сравнено с това, споделената памет изисква едно-две копирания, главно необходими за синхронизация.

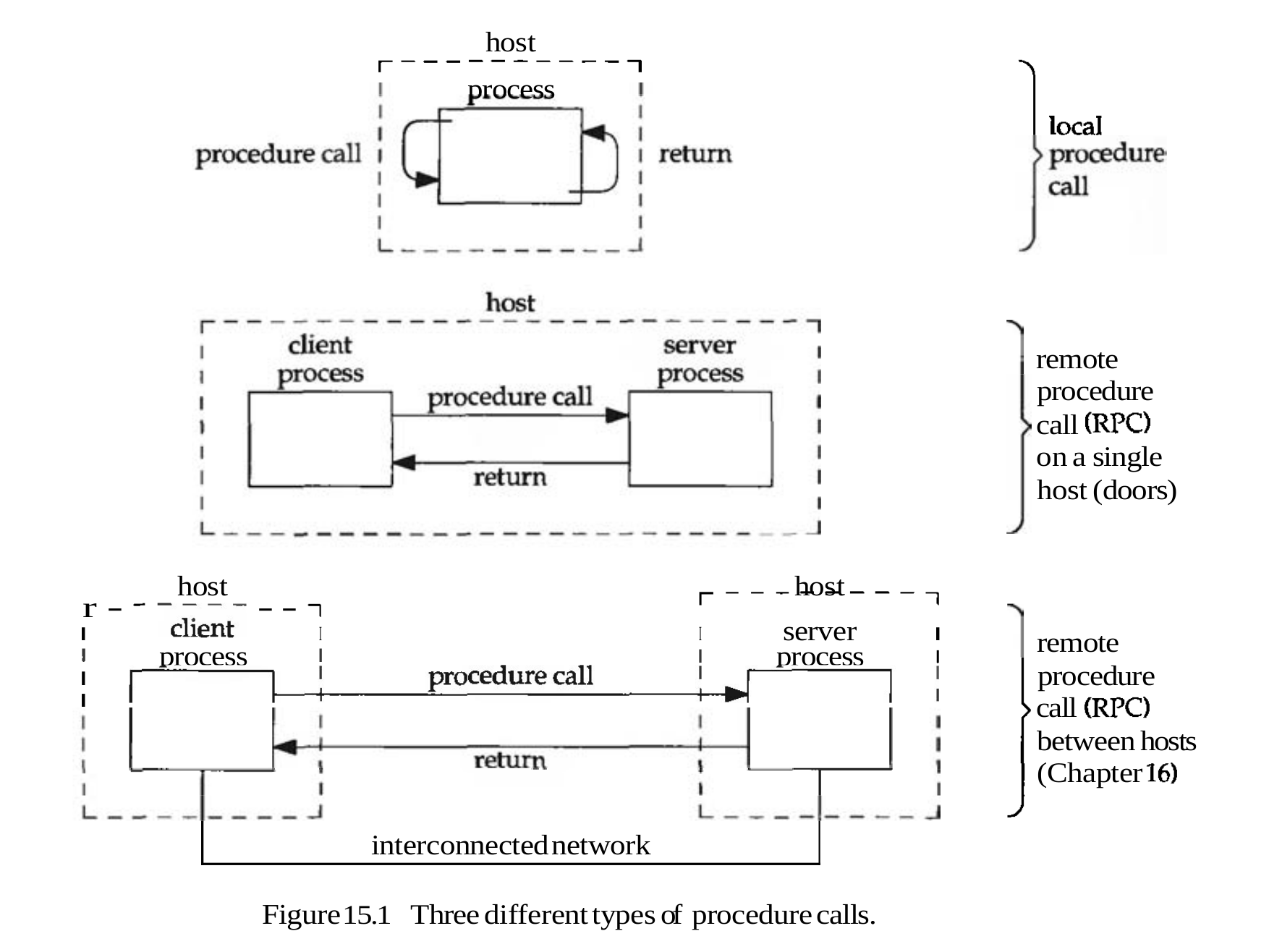


Фиг.2.6

По-рано бяхме споменали, че не се изисква директно участие на ядрото за подобни механизми. Това не е точно така. Ядрото участва по време на заделянето и синхронизацията на паметта, тъй като не се дава директен достъп до нея. Обикновеният начин, по който се процедира, е от адресното пространство на процеса се заделя „memory mapped” регион, който се синхронизира с оригиналната памет. Това може да стане автоматично или да се направи от потребителя, когато той реши за нужно. Парчето памет, което се споделя, може да бъде файл или анонимно парче предоставено от ядрото (Фиг. 2.6).

### Remote Process Calls (RPC)

Това е мощна техника за комуникация на клиент-сървър базирани приложения. Областта на IPC комуникация, която включва е огромна. Може да се изпълни както на една машина така и на две и повече, свързани чрез интернет. Блок-схемата на Фиг. 2.7 илюстрира тази техника [Stevens].

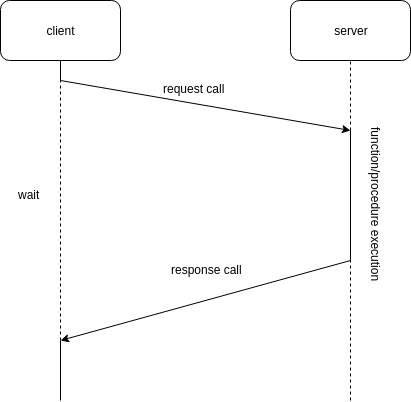
  
Фиг. 2.7

Независимо от това, на какво разстояние се случва комуникацията, тя протича по подобен начин. Първото нещо, което се прави е, да се дефинира услугата (функцията), която ще се изпълнява. Това става от страна на сървъра. В резултат на тази операция се създава идентификатор на съответната функция. Този идентификатор се използва от клиента. С него той прави заявка до сървъра. Съответната функция се изпълнява от сървъра и резултат се връща обратно към клиента.

### Синхронизация

Механизмите за комуникация са от висока важност в обменна на данни между процеси или нишки. Обикновено една от основните цели на синхронизацията е, да обгради дадено парче код и да я третира като критична част за изпълнение. В нея само един от процесите или нишките може да се изпълнява в даден момент.

Когато се каже синхронизация в паралелен алгоритъм, това което се има предвид обикновено е секция от кода, която не позволява неконтролираното модифициране на данни, споделени между процесите. В такива случаи могат да се използват няколко механизма. Единственото условие е, да има някаква форма на споделена памет между таях. Това са „mutex” и условни променливи, ключове за четене и запис и семафори (semaphores). Основната им идея е да създадат критична секция, до която само един процес или нишка да има достъп в даден момент. Някои от горепосочените механизми могат да правят много повече от това, а именно методи като „broadcasting” и “scattering” на сигнали.

  
Фиг. 2. 8

Относно синхронизация между процеси изпълняващи се на различни машини – тя обикновено се осъществява чрез сигнализиране.

### Network

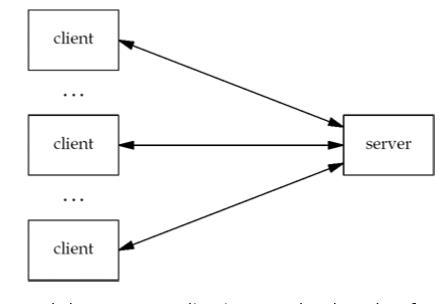
С развитието на комуникационните технологии се появява и друг ясно изразен IPC механизъм, а именно мрежовите технологии. Благодарение на тях, в XXI век, комуникацията между две приложения, намиращи се дори на различни континенти е напълно възможна. Това е и основният начин за IPC комуникация във всички MPI имплементации.

Но, за да работи подобна комуникация, от висока степен на важност е да има протокол, който да описва начина на комуникация между приложенията (процесите). В повечето случай, определено повече от един, една обикновена Ethernet комуникация включва в себе си минимум два протокола IP (Internet Protocol) и TCP (Transport Control Protocol) или UDP (User Datagram Protocol). Те се намират в транспортното ниво на OSI модела.

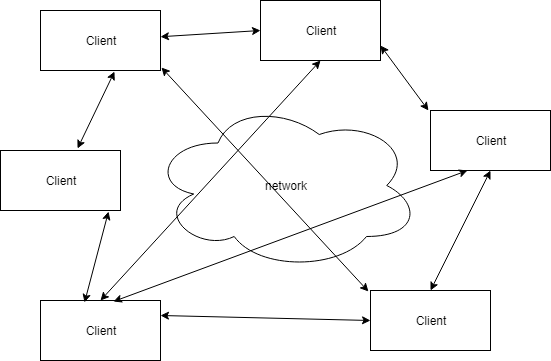
Два от основните модела, които се ползват в такъв тип комуникация са:

* Клиент-Сървър

Това е най-използваният метод в момента, интернет се базира на него. Комуникацията се осъществява между клиент и сървър. Клиентът достъпва сървъра, за да получи информация, обратната операция не е възможна. Обикновено се приема, че един клиент може да прави връзка към един сървър, същият сървър може да приема заявки от множество от клиенти. Но това не винаги е вярно. За пример мога да дам браузера. Той може да прави множество връзки към различни сървъри. На Фиг. 2.10 може да се види диаграма, представяща клиент-сървър комуникацията [Stevens].

  
Фиг. 2.10

* Peer-to-peer

  
Фиг. 2.11

Освен клиент-сървър моделът, който е централизиран, имаме и peer-to-peer (P2P) модел, който е децентрализиран. Вместо да работи с един клиент и един сървър, P2P комуникацията работи с множество клиенти свързани помежду си. MPI използва такъв вид комуникации.

## IPC и MPI

Връзката между MPI и IPC е проста. MPI използва различни IPC механизми, за да предостави удобни за ползване комуни­ка­ци­­онни операции на управля­ваните процеси.

Обикновено една имплементация на MPI разделя различните видове IPC на канали. Тези канали може да варират от версия към версия на стандарта и от имплементация към имплементация. В повечето случаи има един основен, но той не е фиксиран. Библиотеките предоставят възможност за настройки на каналите по време на компилиране. За пример може да вземем MPICH. Тя предоставя мрежова комуникация като основна. RMA операциите се извършват върху нея. Това обаче не е окончателно. В новите версии на библиотеката може да се намерят съвсем нови имплементации на канали, използващи и други IPC механизми. Виж, например [CH4].

## MPI и Android

Първоначалната идея на тази дипломна работа беше да се имплементира MPI стандартът и по-точно библиотеката mpich (виж официалната страница на проекта [MPICH]), в Android. Но това доведе до множество проблеми и в края на краищата, до неуспех. Ето някои от проблемите.

### Интеграция в Android

Android не е просто Linux Kernel, а нещо повече. Той е една от най-използваните ОС в света. Милиони различни устройства работят ежедневно под управлението на Android. Такава ОС изисква огромен ресурс от програмен код. В подобна система, интегрирането на нов код не е лесно и изисква огромно време и други ресурси, за да се направи. Кодът на mpich лесно се компилира за Android. Трудността идва, когато трябва да се направи имплементацията използваема, в смисъл да може да работи оптимално .

### Сигурност

Тъй като ОС Android се ползва от толкова много хора, сигурността и е на много високо ниво. Т.е., за да може да се подкара MPI на Android е необходимо да се преодолеят определени прегради. Това са неща като забранен достъп до определени части от ОС и много други.

Тук изниква проблем, поради който MPI не може да се имплементира директно в Android. MPI библиотеката не е замислена за масово използване. Използва се главно от учени, анализатори, и т.н. и затова не е наблегнато върху нейната сигурност. Дори и да успее интеграцията на библиотеката в Android, тя няма да е достатъчно сигурна за употреба.

### Имплементация и използване

Друг голям проблем е невъзможността да се тества имплементацията на обикновено устройство работещо под Android. Тези устройства обичайно се заключват от производителите, от съображения за сигурност. Също така се цели, да не може да се наруши целостта на ОС. С други думи, за да може да се подкара една подобна библиотека на управлявано от Android устройство, ще се наложи да се „хакне“ системата. И в крайна сметка, ако се направи всичко това, устройството може да стане неизползваемо. Много по-лесно е с подобен краен резултат, да се да се подкара Linux и MPI върху Raspberry pi или да се използва платка за разработка на Android, Orange Pi е подобна платка [Orange Pi].

## Свързване на Java със C (Java Binding)

Точна дефиниция за термина binding няма. Обикновено става дума за механизъм, позволяващ в един език да се използват библиотеки написани на съвсем друг език. За пример може да дадем резултатът от тази работа. Той е базиран точно на това. MPICH е библиотека написана на C. Но имплементацията ни ще позволява извикване на функциите от Java. С други думи може да смятаме *java-mpi* за свързващия компонент, или *Java-Binding* библиотека.

## Прецизиране на целта и задачите

Цялото това проучване, което се наложи да направя за постигане целите на Дипломната работа, ме доведе до друг аспект на предизвикателството „паралелно програмиране в Android“. Тъй като Java е основният език за програмиране там, освен Kotlin, наложи се да се запозная с JNI (Java Native Interface). Това е интерфейс на Java, който позволява комуникация между написани на C/C++ библиотеки (native-библиотеки) и Java Virtual Machine (JVМ). Тогава си дадох сметка, че за да може MPI да работи ефективно под Android, ще се наложи да се напише интерфейс, който да свърже MPI-имплементация с JVM-а на Android. Така се породи и идеята да напишем подобен интерфейс.

Точно този интерфейс ще бъде решение на проблема, свързан с комуникацията между процесите. Идеята е да ***създадем среда за програмиране на Java, позволяваща комуникация по MPI-семантичен начин***. Разбира се, решението няма да е пълна аналогия на стандарта поради факта, че той не включва Javа-binding. Но семантиката на MPI e напълно използваема, а начините за комуникация са напълно възможни и в нашия случай.

  
Фиг. 2.12

Пълна архитектура на проекта ще дадем в по-долните глави, а засега ще предоставим само основен поглед върху проекта и това, как ще бъде имплементиран.

На Фиг. 2.12 е показана архитектурата на проекта в най-общ вид. Тя е изградена от няколко слоя. Първият и *основен слой* е ОС, защото тя ще извършва същинската работа, като ни дава връзка с хардуера. Вторият слой е комуникационен и обединява двата компонента mpich и JVM. Те ще са инсталирани върху ОС, като mpich библиотеката ще е с наши имплементационни промени. Третият и четвъртият слой са една друга двойка компоненти – JNI и java-mpi. JNI ще ни помогне да свържем двата основни езика, които ще ползваме – C и Java. Докато java-mpi ще е самата връзка, като в нея ще има и C, и Java компоненти. Това е същността на настоящата дипломна работа. Последният слой на архитектурата ще е *приложението*, като то ще използва класове и компонентите на java-mpi.

# MPI

## Увод

MPI или още Message Passing Interface (виж [MPI Standard]) е стандарт, описващ комуникация между едновременно изпълняващи се процеси или по-точно спецификация на интерфейс, който да бъде имплементиран в съответни библиотеки. Отнася се за такава парадигма за паралелно програмиране, при което данни се обменят между две или повече адресни пространства (процеси). Това се осъществява с помощта на *кооперативни операции*, предназначени да улеснят разработчиците. Някои от тези операции са: дистанционни операции за достъп (remote-memory access operations), динамично създаване на процеси и паралелни I/O операции. Стандартът предполага реализация на тези операции чрез функции и методи, като се предвиждат най-вече езиците C, C++ и Fortran.

Стандартът е предложен от общност на доставчици на паралелни изчислителни машини, компютърни учени и разработчици на приложения [MPI forum], [Open MPI], [horovod].

Целите на стандарта са:

* Да специфицира приложно-програмен интерфейс (Application Programming Interface, API). Идеята е да дефинира интерфейс, който да се ползва от широк кръг от потребители, машини и софтуер.
* Да позволи ефективна комуникация в обща времева рамка на изчислителните алгоритми, без копиране на памет.
* Да дава възможност за имплементиране в хетерогенни среди.
* Да позволява лесно обвързване (binding).
* Да дава сигурност на потребителя, за да не е необходимо да следи за комуникационни грешки.
* Семантиката на функциите да е независима от езика.
* Интерфейсът да предлага безопасно използване на нишките (thread safe).

Някой от по-популярните имплементации на стандарта са OpenMPI [Open MPI], MPICH [MPICH], InterMPI library и Microsoft MPI. Общото между тях, е че имплементират един и същ интерфейс, което ги прави взаимозаменяеми. В стандарта, както и в имплементациите влизат множество методи за IPC, които се делят на съобщения и споделена памет. В тях се включват елементи като PointToPoint комуникация, колективна комуникация, няколко вида споделяне на паметта и споделени файлове. Не е необходимо разработчикът да се притеснява, как ще се осъществи комуникацията, което довежда до това, че стандартът е перфектен за „Grid“ и други подобни системи, разположени на повече от един компютър.

През годините са обявени множество версии на MPI стандарта (MPI 1.0, MPI 1.1, MPI 1.2, MPI 1.3, MPI 2.0, MPI 2.1, MPI 2.2, MPI 3.0, MPI 3.1, MPI 4.0). Няма да се спираме на определена версия на стандарта, а ще разгледаме общите характеристики на всички версии. Освен интерфейса ще разгледаме и неговата имплементация MPICH2 (на версията MPI 2.0).

Да разгледаме следната MPI-програма:

**Пример 3.1**. [MPI Standard]

#include "mpi.h"

int main(int argc, char \*argv[])

{

char message[20];

int myrank;

MPI\_Status status;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &myrank );

if (myrank == 0) /\* code for process zero \*/

{

strcpy(message,"Hello, there");

MPI\_Send(message, strlen(message)+1, MPI\_CHAR, 1, 99, MPI\_COMM\_WORLD);

}

else if (myrank == 1) /\* code for process one \*/

{

MPI\_Recv(message, 20, MPI\_CHAR, 0, 99, MPI\_COMM\_WORLD, &status);

printf("received :%s:\n", message);

}

MPI\_Finalize();

return 0;

}

Една MPI-програма винаги започва с включване на mpi.h. Всички необходими декларации на функции, глобални променливи и константи се намират в този заглавен файл. Функцията MPI\_Init() се използва за инициализация на дадения процес. Всяка MPI програма завършва с MPI\_Finalize(). Двете функции трябва да се извикват в една и съща нишка. С други думи всяка MPI програма представлява:

MPI\_Init(&argc, &argv);

// основен код (комуникацията и изчисления се извършват тук)

MPI\_Finalize();

MPI\_Comm\_rank() също е много важна функция. Тя връща *ранга* на процеса в определен *комуникатор* (група от процеси, свързани помежду си).

Самата логика на програмата от Пример 3.1 е много проста. Процесът с ранг 0 изпраща съобщение до процеса с ранг 1, а процесът с ранг 1 получава от процеса с ранг 0 това съобщение и го изпраща на стандартния изход.

Самото извикване на програмата за изпълнение може да стане с командния ред:

**mpiexec –n 2 [program] [args if any]**

## Point-to-Point комуникация

Една от основните функции на MPI е да обменя данни между процеси (point to point комуникация). Примерът, който разгледахме по горе, показва точно това. В него използвахме функциите MPI\_Send()и MPI\_Recv(). И двете функции са *блокиращи*, т.е. след извикване на MPI\_Send() програмата в изпращащия процес спира изпълнение, докато MPI\_Recv()в получаващия процес се изпълни успешно и обратно. Разбира се, освен блокиращи има и не-блокиращи функции, както и четири вида имплементации за тях (стандартен, буфериране, синхронен, готов). Разграничават се една от друга чрез префикс, а именно:

* Стандартен (standard): без префикс – MPI\_Send()/MPI\_Recv()*.* Стандартният вид имплементация много зависи от имплементацията на MPI интерфейса. Има два варианта или динамично да се избере подход за запазване на съобщението, или да се преконфигурира. Второто се прави с цел оптимизация. При динамичния подход се гледат наличните ресурси. Например имаме изпращаща функция, но процесът получател няма достатъчно памет да я буферира, в такъв случай се прибягва до това, изпращащата функция да изчака пълното получаване на съобщението. Обаче ако процесът получател има достатъчно памет и буферирането е възможно, изпращащата функция не се блокира (не изчаква пълното получаване на съобщението).
* Буфериране (buffering): добавя се префикс B – MPI\_Bsend()/MPI\_Recv()*.* Това е модел от локален тип, за който няма значение дали ще има получаваща функция. При изпращане на съобщение, ако получаващия процес не е предоставил още MPI\_Recv() функция, то MPI буферира съобщението в буфер предоставен от получателя. Съобщението се изпраща веднага след като получателят предостави MPI\_Recv(). Това води до метод, който не е точно блокиращ но не е и точно не блокиращ, MPI\_Bsend() връща управлението, когато MPI буферира съобщението или то се изпрати.
* Синхронен (synchronous): добавя се префикс S – MPI\_Ssend(). Това е модел, при който изпращащата функция не приключва докато не се извика получаващата функция.
* Готов (ready): добавя се префикс R – MPI\_Rsend()*.* Това модел, при който за да се започне каквато и да е комуникация получаващата функция трябва да е готова за изпълнение т.е. тя трябва да е извикана преди да се извика изпращащата.

Други много удобна функция е:

* MPI\_Probe(). Тя дава възможност на потребителя да проверява статус на point-to-point операция (изпращане/получаване) без тя да е завършила напълно. Информацията, която връща функцията е в структурата от MPI тип MPI\_Status (MPI\_TAG, MPI\_SOURCE,MPI\_ERROR, count, cancelled).

Вторият вид функции са неблокиращи, при тях интерфейсът не изчаква операцията да свърши. Подобни функции са MPI\_Isend() и MPI\_Irecv(). Комуникацията тук е малко по-различна от преди. Функциите не се блокират, а само заявяват операция. След което се оставя на системата да си върши работа. За да е сигурно, че процесът-адресат е получил данните, трябва да се използва допълнителна функция за проверка, например MPI\_Wait() или MPI\_Test().

Напълно е възможно да се ползват и двата вида функции заедно (блокиращи и неблокиращи), достатъчно е да си съответстват. Неблокиращите функции и комбинацията им с блокиращи дава голямо предимство. Благодарение на това се постигат големи оптимизации спестяващи време.

И най-накрая, но не на последно място, ще споменем за функцията MPI\_Sendrecv(). Тя предоставя удобна комуникация на функция от един процес с едноименната функция в друг процес. В MPI различните процеси на една група се образуват чрез копиране на един оригинал, т.е. „fork”. Това означава, че всеки от получените процеси е точно копие на оригинала и тази функция е много удобна в подобни случай.

## Datatypes

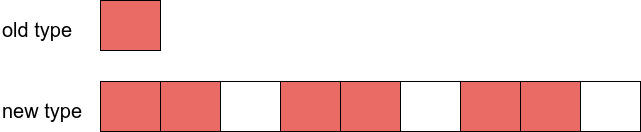
MPI поддържа всички основни типове данни за C и Fortran, но освен тях решава и друг проблем, а именно дефинирането на нови типове, наричани *извлечени типове* (Derive Types). Както в C може да се създават нови типове така и MPI има механизъм за това. За разлика от C, MPI не е компилируем език, а библиотека (интерфейс), и самото дефиниране става по специфичен начин.

Ще разгледаме два от основните вида дефиниране на нови типове, а именно вектори и структури. Приликата между двете е начинът им за описване структурата на типа чрез множество от двойки съдържащи тип и изместване, а именно {(тип0, изместване0), …, (тип1, изместване1)} наречен *типова маска*, а множеството от типовете {тип0, …, тип1} се нарича *сигнатура* на новия тип (type signature).

### Вектори

*Вектор* е метод за конструиране на производни типове данни. Той позволява на няколко *блока* състоящи се от еднотипни елементи да се третират като един тип. Всеки от блоковете се получава чрез обединяване на зададен брой данни от вече съществуващ тип, като тази конструкция позволява, също така, да се дефинират и разстояния между блоковете. MPI функцията, която конструира този тип е MPI\_Type\_vectоr(count, blocklength, stride, old\_type, new\_type). Аргументът old\_type играе много важна роля. В него се задава типът на основната градивна единица на новият тип, а аргументът new\_type указва новия тип данни (новия шаблон). Останалите аргументи определят колко голям ще е блокът и какво ще е разстоянието между блоковете. Пример 3.2 показва дефиницията на нов тип с параметри {*count: 3, blocklength: 2: stride: 3*}. Резултатът може да се види на Фиг. 3.1. А в Пример 3.3 може да се проследи и математиката, която стой зад цялата конструкция на вектори в MIP.

**Пример 3.2.**

  
Фиг. 3.1

MPI\_Type\_vector(2,2,3,old\_type,new\_type);

…

*MPI\_Send(&buf[1],2,new\_type,..);*

**Пример 3.3** [MPI Standard]

{(type 0, disp 0 ),..., (type n−1, disp n−1),

(type 0, disp 0+ex),..., (type n−1, disp n−1 + ex),...,

(type 0, disp 0+(bl−1)\*ex),...,(type n−1,disp n−1+(bl−1)\*ex),

(type 0, disp 0+stride\*ex),...,(type n−1,disp n−1+stride\*ex),...,

(type 0, disp 0+(stride+bl−1)\*ex),...,

(type n−1,disp n−1+(stride+ bl − 1) · ex),...,

(type 0, disp 0 + stride · (count − 1)\*ex),...,

(type n−1, disp n−1 + stride · (count − 1) \*ex),...,

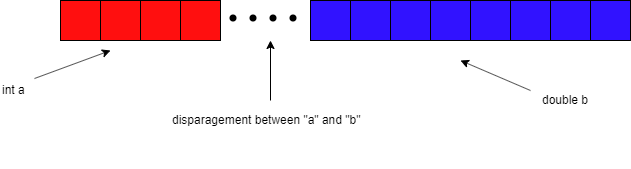
(type 0, disp 0 + (stride · (count − 1) + bl − 1)\*ex),...,

(type n−1, disp n−1 + (stride\*(count − 1) + bl − 1)\*ex)}

### Структури

Главният проблем със структурите, особено в C, е невъзможността да се определи точното подравняване. Различните компилатори слагат различни измествания на основните типове (променливи) в една структура, което прави невъзможно изпращане­то им. В MPI това се решава чрез производни типове. В основата си те не са по-различни от векторни типове. Те също разполагат с отместване на всеки елемент. Разликата идва от там, че докато векторните типове са *N* на брой данни от един и същ тип намиращи се в един блок, то структурните типове са *N* на брой данни от различни типове намиращи се в една C структура.

В пример 3.4. е показано как се създава структу­рен (производен) тип. На фигура 3.2 може да видите примерно разположение на структурата value, в която има int и double променливи. Относно разстоянието, няма как да го определим преди започване изпълнението на програмата. В случая най-вероятно няма да има подобно, но в общия случай, когато са повече на брой променливи няма как да сме сигурни. Това се получава поради факта, че различни компилатори и архитектури, изискват и прилагат различни подравнявания на данните, от където следва, че няма как да определим отместването. В случая на пример 3.4 и фигура 3.2, използваните типове са int и double. След изпълняване на MPI\_Type\_struct и MPI\_Type\_commit се създава нов тип (с име new\_type в пример 3.4), който има формата показана на фигура 3.2. Големината му се определя като броя на активните байтове и не се включват изместванията.

  
Фиг. 3.2

**Пример 3.4.**

MPI\_Type\_struct(count,array\_of\_blocklengths,array\_of\_displace ments,array\_of\_types,new\_type)

....

struct {int a; double b;} value;

// параметри

int blocklengths[2];

MPI\_Aint displacements[2];

MPI\_Datatype old\_types[2];

MPI\_Init(&argc,&argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD,&rank);

// конструкция на параметрите

/\*

една стойност за всеки тип например ако структурата беше {int a; double b[6]; short c[5]}

то стойностите на blocklengths щяха да са {1,6,5}

\*/

blocklengths[0] = 1;

blocklengths[1] = 1;

// основни типове

old\_types[0] = MPI\_INT;

old\_types[1] = MPI\_DOUBLE;

/\*

Тук определяме локацията на всеки елемент! Това може да стане и на ръка, но по-сигурен начин е да се използва MPI функцията MPI\_Address като после ще се наложи само да ги направим релативни т.е да изчислим измесването от началния адрес на структурата.

\*/

MPI\_Address(&value.a,&displacements[0]);

MPI\_Address(&value.b,&displacements[1]);

displacements[1] = displacements[1] - displacements[0];

displacements[0] = 0;

// следва конструкцията и използването на типа.

MPI\_Type\_struct(2,blocklengths,displacements,old\_types,&new\_type);

MPI\_Type\_commit(&new\_type);

MPI\_Send(&value,1,new\_type,..);

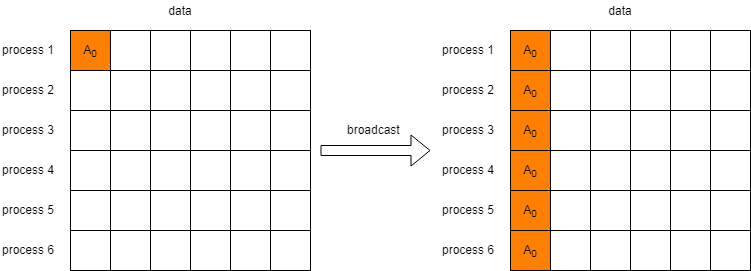
...

След като дефинираме тип е задължително да го направим „достъпен“ (commited), за да може да го ползваме с функцията MPI\_Type\_commit(&new\_type);.

## Колективна комуникация

MPI дефинира колективната комуникация, като комуникация между процесите от една група. Основните функции, който дефинира стандарта са:

### Операцията broadcast

  
Фиг. 3.3

Фиг. 3.3 представя операцията „broadcast” на MPI, която се изпълнява от функцията MPI\_Bcast(buffer, coutn, datatype, root, comm). Както може да се види на фигурата, „broadcast” е операция, която изпращаща съобщение от един процес до всички останали. MPI\_Bcast() изпраща съобщение до всички без основния процес (процеса, който изпраща данните в примера това е процесът с ранг 0). Код използващ операцията е показан на следния пример:

**Пример 3.5.**

....

int buf;

const int root=0;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

...

/\*

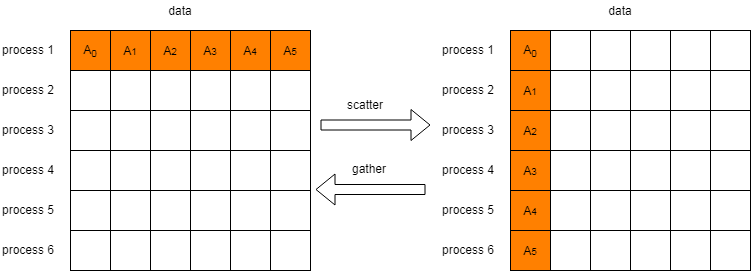
Всеки процес извиква MPI\_Bcast. В резултат от което, всеки процес получава стойността на buf променливата от root процеса(в нашия случай 0).

\*/

MPI\_Bcast(&buf, 1, MPI\_INT, root, MPI\_COMM\_WORLD);

....

### Операциите gather/scatter



Фиг. 3.4

Фиг. 3.4 илюстрира операциите „scatter“ и „gather“ на MPI. За разлика от „broadcast”, “scatter” изпраща различни данни на различни процеси от един процес. А „gather“ е обратната операция на „scatter“, тя взима данни от няколко процеса и ги изпраща до един (root процеса). Операцията се извършва от функцията MPI\_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcounts, recvtype, root,comm). Примери 3.6 и 3.7 илюстрират използването на двете операции.

**Пример 3.6.**

....

/\*

необходими променливи

\*/

MPI\_Comm comm;

int gsize,sendarr[3];

int root, \*rbuf;

...

/\*

Тук изчисляваме необходимата големина на буфера-получател.

\*/

proc\_num = 4

rbuf = (int \*)malloc(proc\_num\*3\*sizeof(int));

/\*

Всички процеси извикват MPI\_Gather. Буферът-получател, в root процеса, се пълни с всички елементи изпратени до него от

останалите процеси.

\*/

MPI\_Gather(sendarr, 3, MPI\_INT, rbuf, 3, MPI\_INT, root, comm);

...

MPI\_Scatter(sendbuf,sendcount,sendtype,recvbuf,recvcount,recv type,root,comm)

...

**Пример 3.7**

...

/\*

необходими променливи

\*/

MPI\_Comm comm;

int gsize,\*sendbuf;

int root, rbuf[100];

...

/\*

Изчисляваме големината на буфера за изпращане, след което го създаваме и пълним.

\*/

MPI\_Comm\_size( comm, &gsize);

sendbuf = (int \*)malloc(gsize\*100\*sizeof(int));

...

/\*

След изпълнението на тази операция, всеки процес ще има 100 елемента идващи от root процеса.

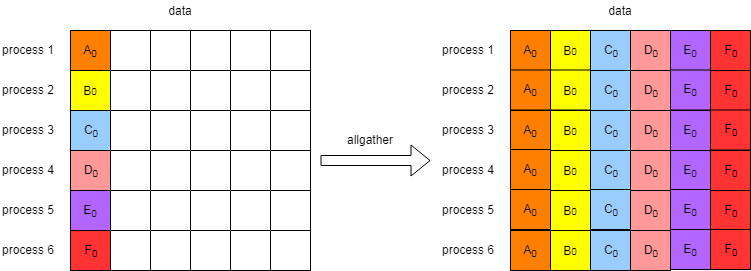
\*/

MPI\_Scatter( sendbuf, 100, MPI\_INT, rbuf, 100, MPI\_INT,root,comm);

...

### Операцията allgather

Може да се приеме, че операцията „allgather” е по-обща разновидност на „gather” с разликата, че всички процеси получават резултата, а не само root процеса. Операцията се изпълнява от функцията MPI\_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm). На фиг. 3.5 може да се види илюстрация на „allgather“ операцията.

  
Фиг. 3.5

## Групи, контекст и комуникатори

MPI стандартът дефинира няколко механизма, с които цели да предостави разделяне на комуникацията, тоест създаването на контекст. Контекстът се имплементира чрез така наречените *комуникатори*. Това са обекти в MPI, дефинирани като такива обхващащи всички комуникационни механизми в групата. Друг механизъм е поставяне на *таг*, с който също е възможно да се дефинира контекст. Това са числа, добавяни като аргументи на point-to-point функциите за комуникация.

Групи в MPI се дефинират, като подредена колекция от процеси, всеки със свой собствен ранг, който се използва като идентификатор за процеса. С тяхна помощ се осъществява point-to-point комуникацията. В допълнение, в една група се извършват всички колективни операции, които споменахме по горе (MPI\_Broadcast(), MPI\_Scatter() и прочие.). В MPI, групите могат да се управляват отделно от комуникаторите, но само комуникаторите могат да се използват в комуникационни операции.

Комуникаторите от своя страна се разделят на два вида. „Intra-communicator“ – комуникатор предназначен за комуникация между процеси намиращи се в една група. Чрез тях се извършват point-to-point и колективни комуникации, както и дават възможност за дефиниране на топология на процесите.

Другият вид – „Inter-communicator“ – са комуникатори позволяващи комуникация между процеси намиращи се в различни групи. Те позволяват, както point-to-point, така и колективни комуникации. Те разполагат с контекст, локални и дистанционни процеси, групи и така нататък. Основни разлики между intra-комуникатор и inter-комуникатор са:

* има малки разлики в комуникацията – не в дефиницията на функциите, но в това как точно се изпълнява и какво върши дадената функция. Не всички функции могат да се използват в inter-комуникация;
* inter-комуникаторитe не позволяват дефиниране ни топология на връзките между процеси.

Една група или комуникатор не може да се създадат от нулата. Винаги се изисква съществуващ комуникатор или група. За това MPI дефинира готови за използване обекти:

* MPI\_COMM\_WORLD е комуникатор, създаван от MPI при инициализация, дефиниран като #define MPI\_COMM\_WORLD ((MPI\_Comm)0x44000000). Той обхваща в група, по премълчаване, всички първоначално стартирани процеси. Също така се използва за създаването на inter-communicator при динамично стартиране на процес.
* MPI\_COMM\_SELF. #define MPI\_COMM\_SELF ((MPI\_Comm)0x44000001). Този комуникатор е дефиниран за всеки процес и съдържа група състояща се само от себе си.
* MPI\_GROUP\_EMPTY. Валиден идентификатор (handler) за празна група.
* MPI\_GROUP\_NULL. Несъществуваща група.

Поради големия брой на функция за манипулиране на групи и комуникатори, няма как да се спрем на всички, вместо това ще дадем кратки примери за използването на някои от тях.

**Пример 3.9.** [MPI Standard]

int main(int argc, char \*argv[]){

int me, count, count2;

static int ranks[] = {0};

...

MPI\_Init(&argc, &argv);

MPI\_Comm\_group(MPI\_COMM\_WORLD, &group\_world);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &me); /\* local \*/

MPI\_Group\_excl(group\_world, 1, ranks, &grprem); /\* local \*/

MPI\_Comm\_create(MPI\_COMM\_WORLD, grprem, &commslave);

if(me != 0){

/\* compute on slave \*/

...

MPI\_Reduce(send\_buf,recv\_buf,count, MPI\_INT, MPI\_SUM, 1, commslave);

...

MPI\_Comm\_free(&commslave);

}

/\* zero falls through immediately to this reduce, others do later... \*/

MPI\_Reduce(send\_buf2, recv\_buf2, count2, MPI\_INT, MPI\_SUM, 0,MPI\_COMM\_WORLD);

MPI\_Group\_free(&group\_world);

MPI\_Group\_free(&grprem);

MPI\_Finalize();

return 0;

}

Фрагментът от Пример 3.9 създава комуникатор и съответно група, които не включват процеса с ранг 0 от MPI\_COMM\_WORLD.

**Пример 3.10.** [MPI Standard]

#define TAG\_ARBITRARY 12345

#define SOME\_COUNT

int main(int argc, char \*argv[]){

int me;int ranks[] = {2, 4, 6, 8};

...

MPI\_Init(&argc, &argv);

MPI\_Comm\_group(MPI\_COMM\_WORLD, &group\_world);

MPI\_Group\_incl(group\_world, 4, ranks, &subgroup); /\* local \*/

MPI\_Group\_rank(subgroup, &me);

/\* local \*/

MPI\_Comm\_create(MPI\_COMM\_WORLD, subgroup, &the\_comm);

if(me != MPI\_UNDEFINED){

MPI\_Irecv(buff1, count, MPI\_DOUBLE, MPI\_ANY\_SOURCE, TAG\_ARBITRARY, the\_comm, request);

MPI\_Isend(buff2, count, MPI\_DOUBLE, (me+1)%4, TAG\_ARBITRARY,the\_comm, request+1);

for(i = 0; i < SOME\_COUNT; i++)

MPI\_Reduce(..., the\_comm);

MPI\_Waitall(2, request, status);

MPI\_Comm\_free(&the\_comm);

}

MPI\_Group\_free(&group\_world);

MPI\_Group\_free(&subgroup);

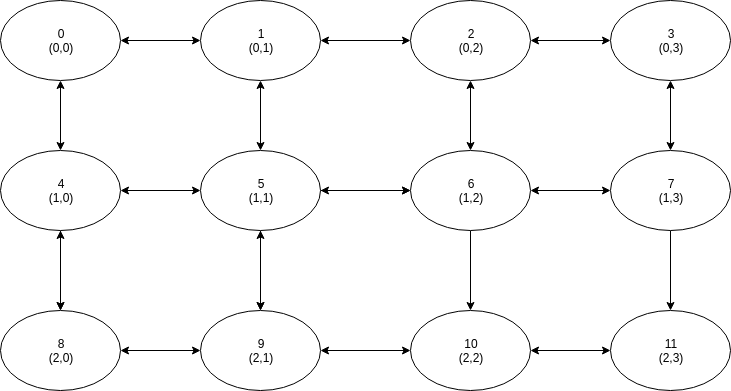
MPI\_Finalize();

return 0;

}

Пример 3.10 е предназначен да покаже безопасността на използване на point-to-point и колективна комуникация. MPI гарантира, че двата вида комуникация няма да си пречат, и че са в два коренно различни контекста.

## Виртуални топологии

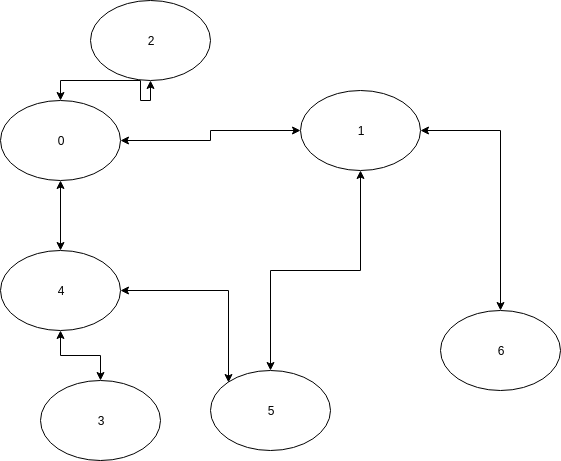


Фиг. 3.6

Това е механизъм, подпомагащ разпределянето (mapping) на процеси по процесори. Също така помага за по-лесното наимену­ване на различните процеси. Възмож­но е само за inter-комуника­ци­ята. Характерни топологии са:

### Декартова топология

На фиг. 3.6 е илюстрирана декартова топология на 12 процеса. Функцията, с която се създава такава топология е: MPI\_Cart\_create(comm\_old, ndims, dims, periods, reorder, comm\_cart).



Фиг. 3.7

### Графова топология

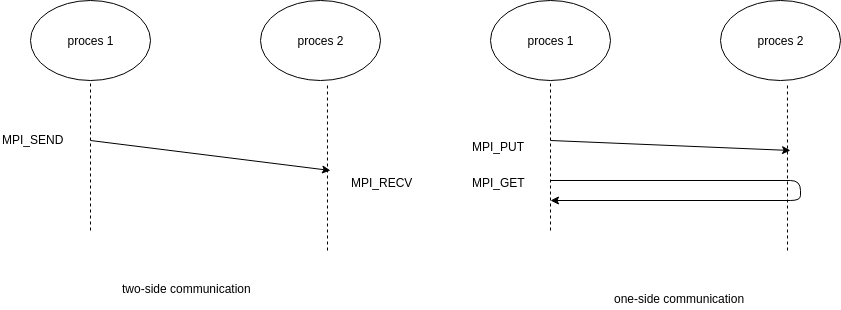
На фиг. 3.7 е илюстрирана примерна топология тип граф. Функцията е: MPI\_Graph\_create(comm\_old, nnodes, intindx[], edges[], reorder, comm\_graph).

## Едностранна комуникация (one-side communication)

MPI дефинира допълнителен механизъм за комуникиране, „Remote Memory Access“ (RMA). Това е механизъм позволяващ на един процес да определи всички комуникационни параметри, както за изпращащата страна така и за получаващата страна. Този начин на работа улеснява достъпа до динамично променящи се данни. Обикновен send/receive модел изисква синхронизация и съответстващи си операции. Докато RMA операциите са напълно независими една от друга. Точно това е описано на фиг. 3.8. „Put“ и „get“ операциите се изпълняват само в един процес, но самата комуникация включва два процеса. Механизмът работи чрез така наречените „windows“ обекти. Като главни операции за достъп са:

* писане (MPI\_PUТ, MPI\_RPUT);
* четени (MPI\_GET, MPI\_RGET);
* обновяване (MPI\_ACCUMULATE, MPI\_REACCUMULATE);
* четене и обновяване (MPI\_GET\_ACCUMULATE, MPI\_RGET\_ACCUMULATE, MPI\_FETCH\_AND\_OP);
* „atomic swap“ операция (MPI\_COMPARE\_AND\_SWAP).

Основната идея на този вид комуникация е да участва само един процес, т.е. комуникация да се осъществява между два процеса, но само единия от тях да извършва същинската инициализация. Целта е да се намали необходимостта от синхронизация.



Фиг. 3.8

На фиг. 3.8 може да се видят съответни времеви диаграми и разликите между MPI\_Send()/MPI\_Recv() и MPI\_Put()/MPI\_Get()операциите.

### Модел на паметта

MPI поддържа два фундаментално различни модела на паметта, отделени (separated) и унифицирани (unified). За да може да разберем по-добре каква е разликата между двата ще използваме концепцията за публичен и личен буфер (win обект). Публичният буфер е регион от паметта, споделен за ползване от всички процеси участващи в RMA комуникацията. В допълнение, всеки от тях може да има и личен или още локален буфер, който да е копие на споделената памет. Такива системи са или кохерентни, или не кохерентни. Когато промяна в основната (споделената) памет предизвиква обновяване и на локалната памет, системата се нарича кохерентна. Когато обаче промяна в основната памет предизвиква конфликт в локалната памет и синхронизация е необходима, системата се нарича не кохерентна. Точно това представляват и двата модела по-горе. Когато става дума за кохерентна система, MPI я дефинира, като унифициран (unified) модел на комуникация. В противен случай е отделен (separated) модел.

### Модели за синхронизация

RMA комуникацията има два основни метода за синхронизация.

* активна целева комуникация (active target communication)

Това е комуникация, при която данни се обменят между адресните пространства на два или повече процеса, като всички процеси са изрични участници в комуникацията. Достъпът на данни може да се извършва само от един процес, но задължение на всеки от участниците в една група е да извършат синхронизация. Функции влизащи в тази категория са: MPI\_Win\_fence(), MPI\_Win\_start(), MPI\_Win\_complete(), MPI\_Win\_post() и MPI\_Win\_wait((). Пример 3.11 и фигура 3.9 описва подобна синхронизация [MPI Standard].

**Пример 3.11.**

// synchronization

MPI\_Win\_fence();

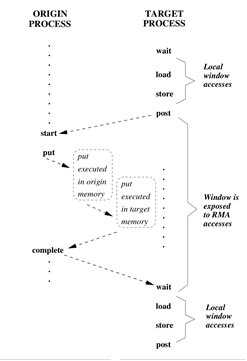
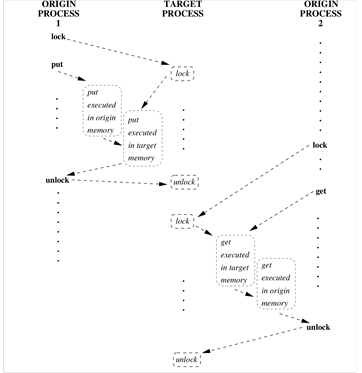
// operations

MPI\_Put():

MPI\_Accumulate();

// synchronization

MPI\_Win\_fence();

Фиг. 3.10 Фиг. 3.11

* пасивна целева комуникация (passive target communication)

Това е комуникация, също позволяваща обмен на данни между адресното пространство на два или повече процеса, но при него не е задължително участието на всеки процес. Функции влизащи в тази категория са: MPI\_Win\_lock(), MPI\_Win\_unlock() и техните разновидности. Фигура 3.10 описва подобна синхронизация [MPI Standard].

## Паралелни входно/изходни операции

Входно/изходните операции са основен проблем на една паралелна система. С многобройните процесори, устройства за съхранени на данни, писането и четенето на един файл не е от най-бързите операции. Отгоре на всичко, входните и изходните операции трябва да са паралелни. MPI дефинира набор от функции използвани точно за тази цел.

Процесът на записване е доста лесен и не се различава особено от обикновеното серийно записване на файл. Всеки процес започва с отварянето на файл, чрез MPI\_File\_open()функцията, като разликата тук е, че трябва да се добави комуникатор, който ще дефинира в каква област от процеси може да се използва файлът. Функцията MPI\_File\_open() е колективна. Накрая, файлът трябва да се затвори чрез MPI\_File\_close(). Функцията гарантира, че всички буферирани данни ще се запишат, както и нормалното съхраняване на файла.

Всеки отворен файл се идентифицира с MPI структурата MPI\_handler. Това е основният идентификатор за отворен файл в една MPI програма и почти всички функции за входно/изходни операции го изискват. Освен него, за всеки файл се поддържа и *файлов указател* (File pointer). Той играе ролята на *офсет* и показва къде ще се изпълни следващата операция, била тя писане или четене.

Един от най-интересните аспекти на паралелните входно/изходни операции в MPI е това, че даден процес може да вижда само част от файла, не целия. Трябва да обърнем внимание на факта, че основната единица за писане и четене в MPI не е байт, а някакъв елементарен тип *etype*. Той, заедно с *типа на файла* и *отместването* дефинират *изгледа* (view) на даден файл от даден процес. Файл-типът е тип, конструиран от etype-елементи. В него е позволено да има дупки, тоест някои etype-елементи са недостъпни в изгледа на определен процес. Файл-типът не се използва само за дадена порция от файла, той е активен от отместването до края му. Отместване е броят на байтове от началото на файла до нулевата позиция на файловия указател, откъдето започва и файловият изглед. По-долу са дадени примери и схеми описващи този модел на достъп.

Това са само някои от основните аспекти на MPI стандарта. В този текст няма как да включим абсолютно всички функции и механизми на MPI. По-подробна информация може да се намери в [MPI Standard] – основния доклад на MPI форума. В следващата глава ще разгледаме една примерна имплементация на стандарта.

# MPICH

MPICH [MPICH] е високо ефективна и широко преносима имплементация на MPI (версии: MPI-1, MPI-2 и MPI-3). Целта и е да предостави MPI имплементация, която да може да се използва от голям кръг компютърни машини с различни комуникационни възможности. Говорим за системи базирани на споделена памет или системи с високоскоростни мрежови технологии (10Gb Ethernet), това са само малка част от технологиите, за които е предназначена библиотеката. Операционните системи, които се поддържат са Linux, Mac и Windows. Макар, че целта на mpich е да не се изпълнява върху Windows. Последната официална версия за тази операционна система е MPI-2. От MPI-3 поддръжката и за Windows спира, възможно е да се компилира за нея, но не и без промени в компилационния процес. Но това не е от особено значение, малко “gird” системи ще се опитат да подкарат изчислителните си алгоритми върху Windows, по-вероятно е да се ползва версия на Linux, която е и фокуса на mpich.

Разработката и започва през 1992 с първата дефиниция на MPI. Базирана е на „Chameleon Portability System“ [Chameleon], от където произлиза и името MPICH (Message Passing Interface CHameleon).

## Процес мениджмънт (Hydra)

Hydra е система за управление на процеси. Тя позволява тяхното стартиране и управление, като за целта се използват демони като ssh, rsh, pbs, slurm и sage. Това са програми, позволяващи комуникация, дистанционно изпълнение на команди, връзка с други устройства, защита и много други. Те участват директно в подкарването на процеси по различните възли участващи в комуникацията.

Hydra е една от основните системи за mpich. Тя включва в себе си много и различни компоненти, като потребителски интерфейс, управление на процесите и много други, за които ще стане дума в следващите раздели (фиг. 4.1 [MPICH WIKI]).

### Потребителски интерфейс – UI (mpiexec)

Основната задача на този компонент е да събира информация от потребителя и да я предава по надолу във веригата. Информацията включва неща, като къде да стартира процесите, колко са, да чете стандартните изходи и входове и други. Повече информация затова каква информация се изисква, може да се получи чрез командата mpiexec -h. „mpiexec“ е конзолна програма, основната точка на потребителския интерфейс.

### Ядро за управление на ресурсите (Resource Management Kernel RMK)

Компонентът предлага възможност за връзка с ресурс менажери (като Torque, Moab или Cobalt). Пример за полза от този компонент е когато системата налага първо инициализиране на възел, а чак след това и стартиране на съответствена задача.

### Процес менажер

Както може да се види от фиг. 4.1 [MPICH WIKI], компонентът се изпълнява върху UI възела (възела инициализация или още job launching executable). Той се занимава с настройката и управлението на създадените процеси. Основните неща, който изпълнява са:

* Създаване и свързване на паралелни задачи.
* Достъп на информация относно паралелни задачи, както и достъп до информация отнасяща се за възлите в една MPI система.
* Обмен на информация между свързани процеси.
* Обмен на информация свързана с „MPI name pushing interface“.

Една програма може би е достатъчна да изпълнява всички показани дейности, но не е удобно и намалява портабилността. За това се ползват множество модули за различните дейности. Например обмен на информация между паралелни задачи може да се прави чрез mpiexec, докато обмен на информация между свързани процеси може да стане чрез писане в споделен файл. Така че това е интерфейс обединяващ набор програми и методи под един покрив.

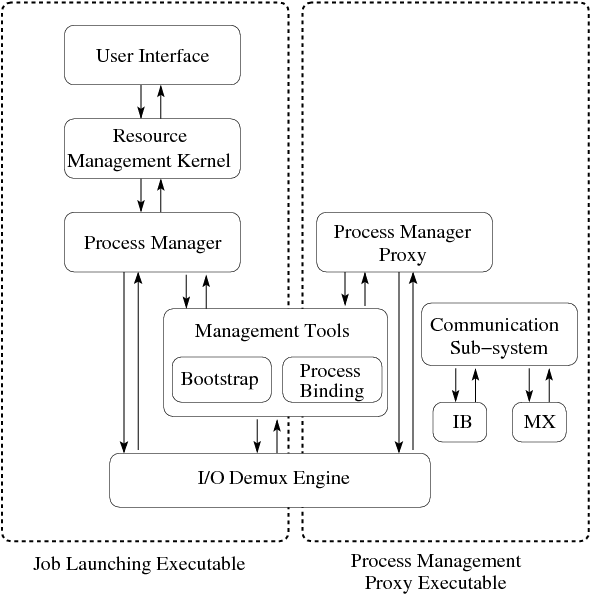
Основната имплементация, която се предлага е процес менажер интерфейс (Process Menadger Interface PMI). Той разполага с две версии, PMI-1 и PMI-2. В по-старите версии на MPICH, около версия 2, се използва PMI-1, докато в новите, над версия 3, може да се наблюдава използването на PMI-2.

### Процес мениджър прокси

Основната задача на проксито е да играе ролята на помощен агент за процес мениджъра. Той се изпълнява във всички възли без UI възела, като помага за създаване на процеси, тяхното почистване, I/O операции както и повечето от дейностите описани в **4.1.3**. Т.е. може да изпълнява всички задачи свързани с управлението на процесите и дори позволява създаването на друг процес мениджър прокси.

### „Bootstrap server“ (ssh, fork, pbs, slurm, sge)

В основата си това е компонент, който пряко участва в стартирането на процеси. Той използва външна програма, както е случая с ssh или вътрешен механизъм, както е в случая с fork (системна функция позволяваща изпълнени на процес от друг процес). Демоните описани в началото се използват точно в този компонент.

Фиг. 4.1

Архитектура на процес мениджъра (hydra).

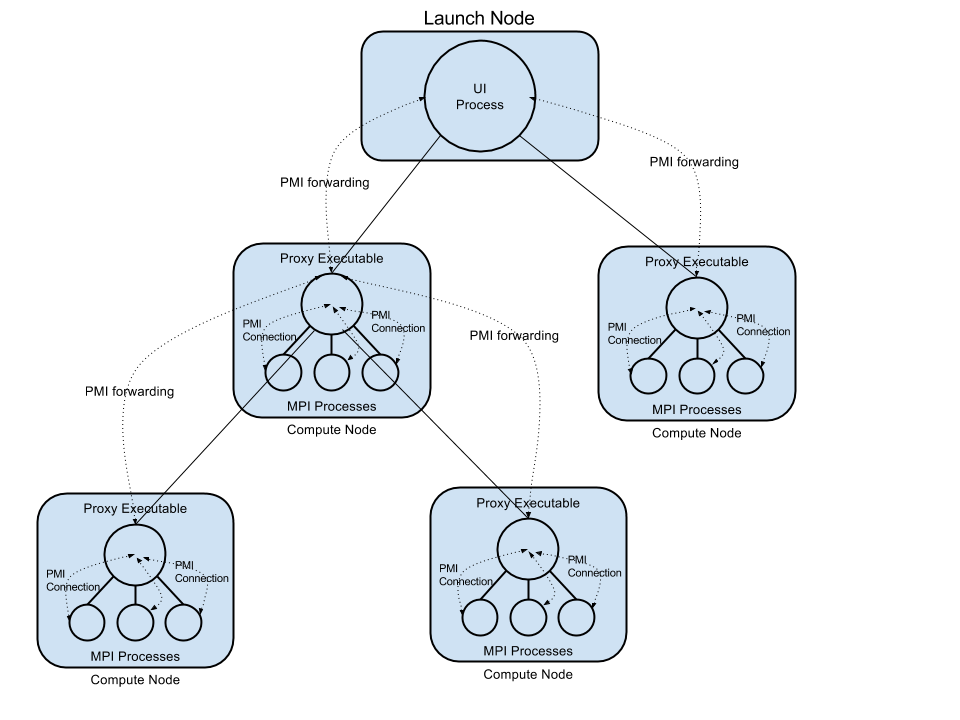
### „I/O demux engine“

Това не е компонент, който пряко помага за стартирането и управлението на процесите. Той е създаден за удобство. В него различните компоненти регистрират различни файлови дескриптори заедно с указатели на функции за определени събития като четене или запис. След като регистрацията се осъществи, компонентът започва да слуша за събития отнсящи се за функциии, които се изпълняват. С негова помощ се избягва необходимостта от стартиране на нишки, извършващи „polling“, т.е. изчакване на операцията да приключи.

## Поток на изпълнение

Всички горепосочени компоненти се включват тук. Точно UI интерфейсът е първият, който се изпълнява. Той взима цялата подадена му информация от потребителя и я подава надуло по веригата. Свързва се с RMK, който компонент се грижи за това да има достатъчно ресурси на разположение. От своя страна RMK си комуникира с процес мениджъра, като той се грижи непряко за създаването на процесите и тяхното управление. Аргументите подадени от потребителя биват преформатирани под формата на списък, след което се подават на процес мениджъра. Той от своя страна, с помощта на „bootstrap server“-а, помага за стартирането на прокси сървъри (процес мениджър прокси програми). Не е задължително всички те да са свързани пряко с UI процеса – проксито поддържа и йерархична структура, което дава възможност на различните възли да предават информация чрез PMI. Не е задължително подобна програма да се стартира от UI възела, възможно е да се стартира от друг прокси процес. Веднъж стартиран, всеки прокси сървър си предава входно/изходните дескриптори (stderr,stdout и stdin) на UI процеса, който се намира върху UI възела или по-точно възела отговарящ за стартирането на цялата система. Това става посредством „demux enginee“-a, който освен при прокситата се стартира и при UI процеса. На него той слуша за приключването на работата на всеки от тях. Обаче преди да приключи работата си той стартира процесите определени му за дадения възел. Общо взето UI възелът стартира UI процес, който стартира един или повече прокси програми, които от своя страна стартират процеси (фиг. 4.2 [MPICH WIKI] .).

След приключване работата на процесите, задачата по почистване и засичане на грешки е на процес мениджъра и неговите помощници, прокси програмите. Те гарантират това, че процесите и задачите ще бъдат стартирани, след което ще бъдат изпълнени и при приключване системата ще мине и ще освободи всички използвани ресурси. Те не гарантират за програмни грешки, мъртви хватки и други подобни грешки.

  
Фиг. 4.2

## Mpich архитектура

Архитектурата на всички подобни библиотеки е еднаква. Да вземем, например, OpenMPI и MPICH. Това са две имплементации на MPI. Няма голяма разлика между тях, както в имплементация така и в използване. Дори имат общи модули. Например, библиотеката hwloc (Hardware Locality). Това е библиотека помагаща за откриване на хардуерни ресурси в паралелни архитектури [Hwloc], която се използва и в двата проекта, OpenMPI и MPICH. Това означава, че повечето неща отнасящи се за MPICH са верни и за OpenMPI, както и за други подобни имплементации. Но това не означава, че те са съвсем еднакви, най-вероятно съществуват ключови аспекти, в който те се различават, което ги прави по-удобни в един и по-неудобни в друг случай.

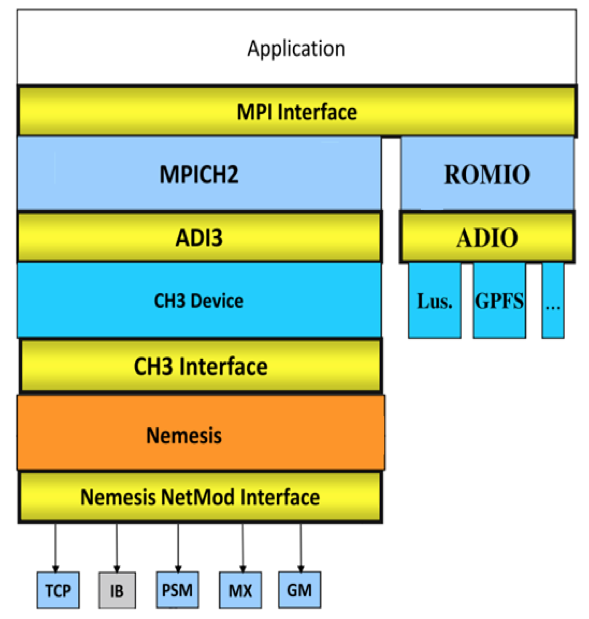
MPICH се базира на многослойна архитектура, т.е. различните компоненти имплементират различен аспект на абстракция и функция. Всички компоненти на MPICH2 може да видите на фиг. 4.3 [MPICH WIKI], като в следващите раздели са дадени кратки обяснения.

### Процеси, групи и виртуални връзки (Virtual Connections - VC)

Комуникацията между процеси в MPICH се осъществява чрез виртуални връзки (VC). Те се имплементират от CH3/CH4, за които ще стане дума по-долу. В много от случаите, връзките не се осъществяват, докато това не се изиска. По този начин се запазва мащабируемостта на системата и се намалява времето за стартирането ѝ. Всяка виртуална връзка се асоциира с група от процеси (PG). Всяка VC се идентифицира чрез групата процеси, в която участва и от ранга на тази група, като всяка една VC участва само в една PG. Както можете да се досетите имаме и група MPI\_COMM\_WORLD, която представлява всички процеси. Всяка друга група от процеси е някакво подмножество на MPI\_COMM\_WORLD, както е описано в стандарта.

Всяка група от процеси разполага с референтна таблица за виртуални връзки (virtual connection reference table). Тя представлява структура от числа и указатели към масив от виртуални връзки, индексирани по ранг. Освен с тях, структурата разполага и с брояч на референциите, който помага за тяхното управление. Самата тя позволява динамично създаване на групи и свързване на процеси към тях. Животът на всяка виртуална връзка започва с функции като MPIDI\_VC\_Init() или MPIDI\_CH3\_VC\_Init() и завършва с функции за прекратяване на връзката или MPIDI\_CH3\_VC\_Free()**.**

### Abstraction Device Interface (ADI3)

Фиг. 4.3

ADI3 е слой от MPICH отговарящ за комуникацията между процесите. Той имплементира набор от функции, който се използват за point-to-point и едностран­на комуникация. Той не е имплементация, а интерфейс. Самият той е абстрактен слой за MPICH, отговарящ за комуникацията. Самата кому­ни­кация се имплементира от друг слой. В MPICH2 това е Channels 3 (CH3), както може да видите на фигура 4.3 [MPICH WIKI], докато в MPI-3 това е Channels 4 (CH4). И двата слоя изпълняват една и съща функция – да имплементират ADI3 интерфейса.

### CH3

Това е слой имплементиращ няколко на брой комуни­кационни канала. Те отговарят за изпращане на данните между MPI процесите. В допълнение на това, някой от каналите предлагат и допълнителни функционалности, подобрения и други избираеми възможности, които могат да се използват за подобряване на производителността на комуникацията в канала. Различни канали могат да се изберат по време на компилация. На фиг. 4.3 е показан каналът Nemesis, но това не е единственият канал, който може да се избере. CH3 разполага и със сокет комуникационен канал, както и няколко, който не се поддържат вече. Това са shm (комуникационен канал, използващ споделена памет), ssm (комуникационен канал, включващ в себе си сокет комуникация и споделена памет), dllchan (канал, позволяващ динамично зареждане на канали) и, не на последно място, sctp (канал, използващ SCTP сокет комуникация [SCTP]).

### CH4

Това е по-нова и оптимизирана имплементация на ADI3 интерфейсa. С времето CH3 придобива нови оптимизации и хакове, започва да използва нови комуникационни технологии и мрежови архитектури, което се явява далеч от оригиналната идея на слоя. За това се появява CH4, следваща версия на CH3.

* CH3 и цялата комуникация на MPICH е базирана на **виртуални връзки** VC. С времето VC разработва функционалности, които не са съвсем на място и не разширяват мащаба на комуникацията.
* CH3 e комуникация, базирана на **активни съобщения**. Това изисква добавянето на допълнителна информация към потребителските данни. Тези данни помагат на системата да определи къде и за кого е съобщението, освен това дава и информация за типа му. Проблемът при такъв тип комуникация е, че информацията за съобщението и потребителските данни се намират на различни места, което забавя комуникацията. В новите версии на CH3 това се оптимизира и се разработват решения, които целят да оптимизират и подобрят този процес, но те са по-скоро временно решение отколкото постоянно.
* CH3 разполага с функционалност позволяваща на компоненти да **заместват функции** на други компоненти. Това се осъществява чрез указатели. Но този модел на имплементация се смята за неприемлив поради факта, че генерира голям брой инструкции, което прави крайния изпълним файл доста голям.
* CH3 разполага с **не толкова добра и оптимизирана споделена памет**. В много от случаите слоят прави ненужни замествания, което забавя и прави тромаво предаването на данни до мрежовия слой на компонента.
* CH3 разполага с **не толкова добър приложно-програмен интерфейс** за мрежовата комуникация. Това довежда до факта, че слоят ползва прости комуникационни функции за съобщения и не използва пълната функционалност и сила на мрежовите компоненти намиращи се на по-долно ниво.

Общо взето това са проблемите, които авторите се опитват да решат с CH4. Общо взето, посоките на сегашната версия на компонента е да подобри VC и да ги използва само там където е необходимо. CH4 ще предоставя структури, съдържащи всички данни за комуникация. Както и, не на последно място, по-добро и оптимизирано използване на мрежовите технологии и споделяне на памет.

### Канали за комуникация

Каналите за комуникация са най-ниското ниво от архитектурата на MPICH, те отговарят за същинското пренасяне на данни от процес до процес. На фиг. 4.3 [MPICH WIKI] е изобразен Nemesis. Той е един от основните канали използвани в CH3 компонента. Разполага с различни модули и начини за комуникация, които го правят доста бърз и оптимизиран. Освен с него, CH3 разполага и със sock. Това е канал, използващ сокети за комуникация.

Това са само два от каналите имплементирани за комуникация. В новата версия на CH, т.е. CH4, са въведени много нови и по-оптимизирани комуникационни канали, за които няма време и не е необходимо да са споменати тук. Повече информация може се намери на wiki страницата на MPICH [MPICH WIKI] или в самият код на MPICH [MPICH SOURCE].

### ROMIO

ROMIO [ROMIO] е имплементация на входно/изходните MPI операции (Глава 4 на стандарта). Имплементацията е напълно отделен компонент и не се разработва специално за MPICH. Напротив, тя се намира в набор от различни MPI имплементации като, MS-MPI (Microsft версията на MPI), Open MPI и други. Също така може да се види в супер компютъра на IBM, Blue Genе [IBM Blue Gеne].

Разбира се, има много повече информация, която може да разгледаме относно MPICH и всички подобни имплементации, но това не ни е целта. Целта ни е да предоставим интерфейс, който да може да се използва заедно с java. В следващата глава правим точно това – ще разгледаме начина, по който ще свържем имплементация на MPI, в нашия случай MPICH, с Java.

# JNI

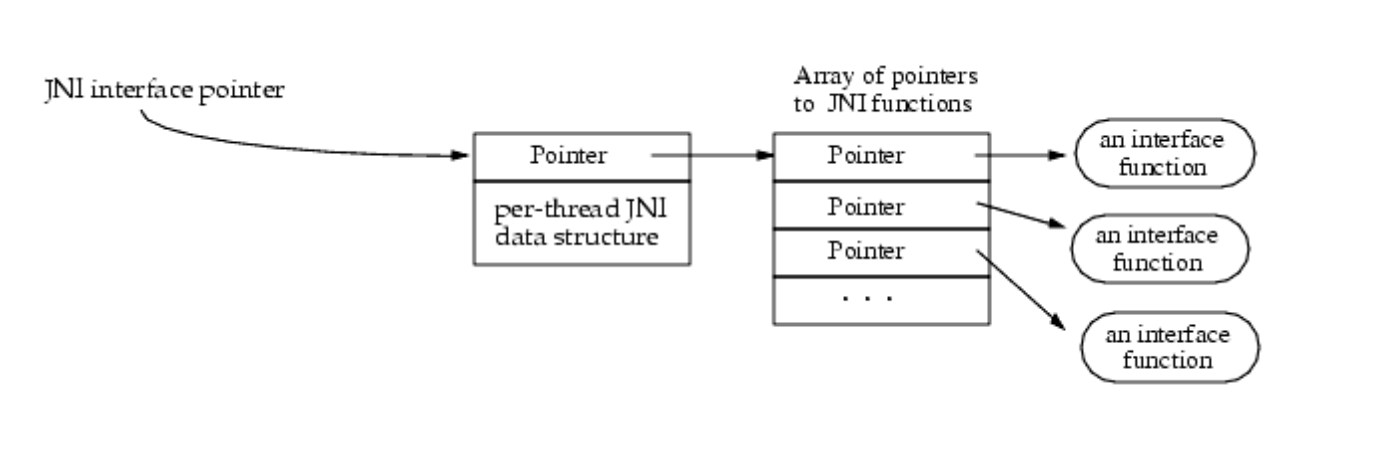
Главата е предназначена за читатели, които не познават JNI (Java Native Interface). Както споменахме в Глава 1, това е интерфейс който свързва C/C++ код с Java. С две думи, идеята му е да може C библиотека да се изпълнява от *java bytecode* без проблеми. Точно JNI се явява междинния слой между двете среди [JNI].

Идеята ни за разработка в подобна посока идва от първоначалната задача, а именно – да разработим MPI библиотека за Android. Дори и при съвсем нов подход, без използването на вече съществуваща имплементация на MPI, използването на JNI е неизбежно. Рано или късно ще се наложи комуникация между двата слоя, bytecode-а на java и native кода на C. Това е необходимо, защото комуникацията се извършва на най-ниско ниво. Освен това, някои оптимизации ще са далеч по ефективни, та дори и възможни, само на ниско ниво.

## Дизайн

C/C++ код, наричан за кратко *native* код, има достъп до Java VM (Virtual Machine) функционалност през JNI функции. Тези JNI функции са достъпни през указател. Той сочи към масив от указатели, които от своя страна сочат към въпросните функции. Основна полза от подобна таблица от указатели, е че пространството от имена се пази чисто. Друго удобство е това, че една VM може да предложи множество интерфейси, които да могат да се използват в различни случай. На фигура 5.1 [JNI] може да видите илюстрация на въпросната схема.

Поради факта, че Java VM поддържа многонишково изпълнение, такова трябва да поддържа и JNI. Това довежда до необходимостта отделните нишки, които имат връзка с C/C++ код, да използват отделни JNI интерфейсни указатели. Това обаче не означава, че зареждането на библиотека трябва да стане във всяка нишка. Пример 5.1 показва как една библиотека може да бъде заредена.

Фиг. 5.1

**Пример 5.1**

package pkg;

class Cls {

static {

System.loadLibrary(“pkg\_Cls”);

}

public native double f(int i, String s);

}

Извикването System.loadLibrary(String name) се използва точно за зареждане. Системата преобразува името на библиотеката спрямо стандарта за дадената ОС. Т.е. ако имаме something\_lib в Unix-подобна система, тогава VM ще търси в директориите със споделени библиотеки библиотека с име libsomething\_lib.so.

От страната на Java може да се види декларация подобна на

native double f(int i, String s);.

Точно с ключовата дума native се отбелязва метод, свързан с име на функция от C библиотека. От страна на C също се изисква по специална дефиниция и декларация на функцията предназначена да се извика от Java код. В пример 5.2 [JNI] може да се види подобна функция. След като библиотека е заредена, виртуалната машина проверява за съвпадение в символите, като търси за две имена. В първия случай търси за име съдържащо само пътя до метода и самото му име, тоест нещо подобно на Java\_packages\_class\_methodname. Във втория – освен пътя имаме описани и аргументите. Той се използва в случаите, в които имаме припокриване на методи (overlaping). В пример 5.2 може да се види подобна дефиниция на функция. Тези, както и други, правила не е нужно да се помнят. Java разполага с метод за генериране на декларациите необходими за C/C++. С добавянето на -h path към javac при компилиране, заглавните файлове ще бъдат генерирани автоматично така, че да съдържат всички необходими декларации.

**Пример 5.2 [JNI]**

JNIEXPORT jdouble JNICALL

Java\_pkg\_Cls\_f\_\_ILjava\_lang\_String\_2 (

JNIEnv \*env, /\* interface pointer \*/

jobject obj, /\* "this" pointer \*/

jint i, /\* argument #1 \*/

jstring s) /\* argument #2 \*/

{

/\* Obtain a C-copy of the Java string \*/

const char \*str = (\*env)->GetStringUTFChars(env, s, 0);

/\* process the string \*/

...

/\* Now we are done with str \*/

(\*env)->ReleaseStringUTFChars(env, s, str);

return ...

}

Функционалността на JNI е доста обширна. Интерфейсът има възможност за:

* преобразуване на данни;
* връзка, както от Java към C/C++, така и от C/C++ към Java;
* използване на java.nio.Buffer;
* използването на API функция.

## Типове

Най-сериозният проблем на общуването между C и Java програми са типовете данни. Повече от 50% от функциите са предназначени за обработка и преобразуване на данни. Това се налага поради липсата на съответствие между Java типове и C/C++ типове.

Пример за такова несъответствие е типът char. В Java той е с размер 16 бита поради факта, че има възможност за представяне UTF кодове. До като в C/C++ char е 8 битов. Трансформацията на типовете е показана в Таблица 1.

**Таблица 1.** Основни типове.

|  |  |  |  |
| --- | --- | --- | --- |
| **Java Type** | **Native Type** | **Description** | **Type signature** |
| boolean | jboolean | Unsigned 8 bits | Z |
| Byte | jbyte | Signed 8 bits | B |
| Char | jchar | Unsigned 16 bits | C |
| short | short | Signed 16 bits | S |
| Int | jint | Signed 32 bits | I |
| Long | jlong | Signed 64 bits | J |
| float | jfloat | 32 bits | F |
| double | jdouble | 64 bits | D |
| Void | void | N/A | V |

Съответните jboolean стойности са:

#define JNI\_FALSE 0

#define JNI\_TRUE 1

а пример за тип, описващ големина е:

typedef jint jsize;

Друг проблем е, че референциите към обекти, методи и полета трябва да могат да бъдат достъпни и от C/C++ код. Този достъп, обаче, изисква по-сложен механизъм, който може да се види в пример 5.3. В примера е показано създаването на два обекта.

**Пример 5.3**

jclass gJniStatusClass;

jmethodID gJniStatusGetStatus;

jmethodID gJniStatusConstructor;

jclass gJniMessageClass;

jmethodID gJniMessageConstructor;

jobject newJniMpiMessage(JNIEnv \*env,MPI\_Message\* msg,MPI\_Status\* stat,jboolean flag){

jobject status = (\*env)->NewObject(env,gJniStatusClass,

gJniStatusConstructorReq,stat->count\_lo,

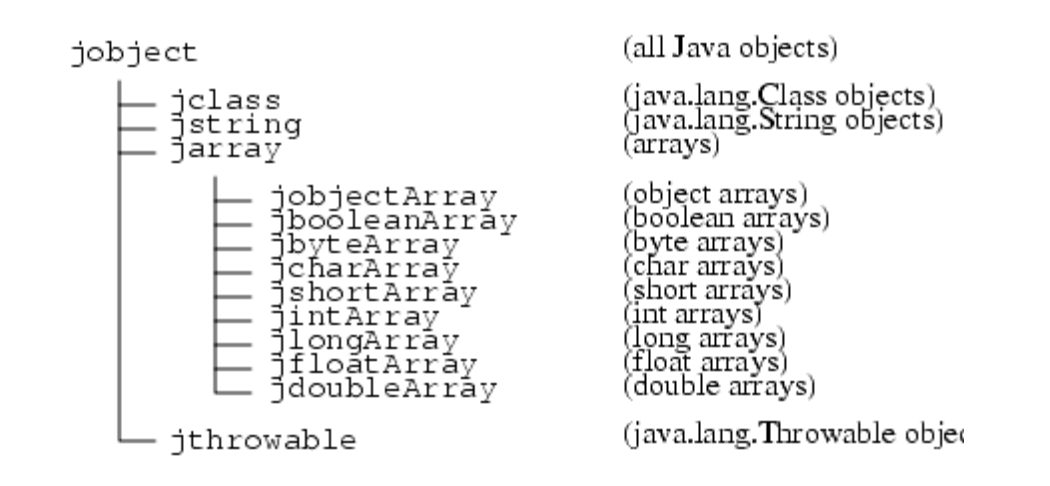
stat->count\_hi\_and\_cancelled,stat->MPI\_SOURCE,

stat->MPI\_TAG,stat->MPI\_ERROR,flag);

return (\*env)->NewObject(env,gJniMessageClass,gJniMessageConstructor, (jlong)\*msg,status);1

}

Освен jobject JNI дефинира и други често използвани обектни типове, които може да се видят на фиг. 5.2 [JNI].

Фиг. 5.2

## Достъп от C/C++ към Java

Вече разгледахме как Java-програма осъществява достъп до C-програма. Обратното се прави чрез JNI указател. Това е същият указател, даващ достъп до всички функции на интерфейса. В пример 5.2 това е envот тип JNIEnv*\**, като, освен функциите, има и няколко променливи, които помагат за осъществяване на достъпа:

struct \_jfieldID; /\* opaque structure \*/

typedef struct \_jfieldID \*jfieldID; /\* field IDs \*/

struct \_jmethodID; /\* opaque structure \*/

typedef struct \_jmethodID \*jmethodID; /\* method IDs \*/

jfieldID е идентификатор, използван в C за достъп до полета, намиращи се в Java обект. jmethodID е идентификатор, използван в C за достъп до методи, намиращи се в Java oбект. Тези два тип указатели показват на JNI какво точно да търси. Инициализацията им става чрез специални функции:

* Функцията jclass FindClass(JNIEnv \*env, const char \*name); се използва за намиране на клас.
* Функцията jmethodID GetMethodID(JNIEnv \*env, jclass clazz, const char \*name, const char \*sig); инициализира идентификатор на метод.
* Функцията jfieldID GetFieldID(JNIEnv \*env, jclass clazz, const char \*name, const char \*sig); инициализира идентификатор на поле.

Указателят sig е специален аргумент, приеман от всяка функция свързана с инициализация на jfieldID или jmethodID. Той описват сигнатурата и типа за достъпа, неща като:

* Какъв е типът данни, който връща методът.
* Дали полето е int, или double, или long ... .
* От какъв тип са аргументите на един метод.

За jfieldID, сигнатурата му е от вида *<type signature>*. За jmethodID, сигнатурата има вид (*<type signature>,<type signature>,...)<type signature>,* като обектите имат по-специфична сигнатура. Общо взето тя описва пътя до обекта с две допълнения. Накрая на сигнатурата се добавя “;“, докато в началото се добавя „L“. Представлява нещо подобно на L<package>/<package>..../<class name>;, например, Ljava/nio/ByteBuffer;.

Вече инициализирани, jfieldID и jmethodID могат да се използват за достъ до свойствата на един обект. Примерна функция за достъп е:

NativeType Call<type>Method(JNIEnv \*env,jobject obj,

jmethodID methodID, ...);

Функцията изпълнява метод, който връща стойност от тип <type>(всички в Таблица 1, включително и jobject. Примери 5.4, 5.5 и 5.6 описват примерен достъп до метод с име setStatus от клас mpi.Status.

**Пример 5.4**

cls = (\*env)->FindClass(env,"mpi/Status");

gJniStatusClass = (\*env)->NewGlobalRef(env,cls);

(\*env)->DeleteLocalRef(env,cls);

if(gJniStatusClass == NULL){

return JNI\_ERR;

}

gJniStatusSetStatus =(\*env)->GetMethodID(env,gJniStatusClass, "setStatus","(IIIIIZ)V");

Самият Java метод има вида показан в пример 5.5

**Пример 5.5**

private void setStatus(int count\_lo,int count\_hi\_and\_cancelled,

int mpi\_source,int mpi\_tag,int mpi\_error,boolean isReqFinish

){

this.count\_lo = count\_lo;

this.count\_hi\_and\_cancelled = count\_hi\_and\_cancelled;

this.mpi\_source = mpi\_source;

this.mpi\_tag = mpi\_tag;

this.mpi\_error = mpi\_error;

this.isRequestFinish = isReqFinish;

}

**Пример 5.6**

// cache Request class

cls = (\*env)->FindClass(env,"mpi/Request");

gJniRequestClass = (\*env)->NewGlobalRef(env,cls);

(\*env)->DeleteLocalRef(env,cls);

if(gJniRequestClass == NULL){

return JNI\_ERR;

}

gJniRequestHandler = (\*env)->GetFieldID(env,gJniRequestClass, "handler","J");

...

// Самото поле има вида

public class Request {

    private long handler;

...

}

## Функции

JNI има много и различни функции помагащи в различни аспекти от интерфейса и комуникацията между Java и C. Както видяхме в раздел **5.3**, достъп до методи и полета на един Java клас е напълно възможен, благодарение на подобни функции. Освен тях интерфейса разполага и с много други:

* Информация за версията:

jint GetVersion(JNIEnv\*)

* Операции с класове:

jclass DefineClass(JNIEnv\*,const char\*,jobject,cont jbyte\*,jsize)

jclass FindClass\*(JNIEnv\*,const char\*)

...

* Изключения (Exceptions):

jint Throw(JNIEnv\*,jthrowable)

jint ThrowNew(JNIEnv\*,jclass,const char)

...

* Управление на глобални и локални референции (разликата е само в обхвата на използване):

jobject NewGlobalRef(JNIEnv\*,jobject);

void DeleteGlobalRef(JNIEnv\*, jobject);

void DeleteLocalRef(JNIEnv\*, jobject);

...

* Слабо глобални референции (този вид глобални референции е по-специален от обикновените; когато една глобална референция се отбележи като слаба това означава, че референцията на Java обекта, намиращ се в C, е обект на разглеждане от системата за почистване на паметта (garbage collection))

jweak NewWeakGlobalRef(JNIEnv \*env, jobject obj)

void DeleteWeakGlobalRef(JNIEnv \*env, jweak obj)

* Операции с обекти:

jobject AllocObject(JNIEnv\*, jclass);

jobject NewObject(JNIEnv\*, jclass ,jmethodID, ...);

jclass GetObjectClass(JNIEnv\*, jobject);

...

* Достъп до методи (Call<type>Method е то този тип):

jmethodID GetMethodID(JNIEnv\*,jclass ,const char\*,const char\*);

**NativeType** Call**<type>**Method(JNIEnv\*, jobject, jmethodID, ...);

**NativeType** CallNonvirtual**<type>**Method(JNIEnv\*, jobject, jclass, jmethodID, ...);

...

* Достъп до полета (Get<type>Field):

jfieldID GetFieldID(JNIEnv\*,jclass,const char\*,const char\*);

**NativeType** Get**<type>**Field(JNIEnv\*,jobject,jfieldID);

void *Set***<type>***Field*(JNIEnv\*, jobject, jfieldID,**NativeType**);

...

* Достъп до статични методи:

jmethodID GetMethodID(JNIEnv\*,jclass,const char\*,const char\*);

**NativeType** Call**<type>**Method(JNIEnv\*,jobject,jmethodID,...);

**NativeType** CallNonvirtual**<type>**Method(JNIEnv\*, jobject,jclass, jmethodID,...);

...

* Извикване на статични полета:

jfieldID GetStaticFieldID(JNIEnv\*,jclass,const char\*,const char\*);

**NativeType** GetStatic**<type>**Field(JNIEnv\*,jclass,jfieldID);

void SetStatic**<type>**Field(JNIEnv\*,jclass,jfieldID,**NativeType**);

...

* Операции с низове:

jstring NewString(JNIEnv\*,const jchar\*,jsize);

jsize GetStringLength(JNIEnv\*,jstring);

const jchar \* GetStringChars(JNIEnv\*,jstring,jboolean\*);

void ReleaseStringChars(JNIEnv\*,jstring,const jchar\*);

jstring NewStringUTF(JNIEnv\*,const char\*);

...

* Операции с масиви:

jsize GetArrayLength(JNIEnv\*,jarray);

jobjectArray NewObjectArray(JNIEnv\*,jsize,jclass,jobject);

jobject GetObjectArrayElement(JNIEnv\*,jobjectArray,jsize);

**ArrayType** New**<PrimitiveType>**Array(JNIEnv\*,jsize);

**NativeType** \*Get**<PrimitiveType>**ArrayElements(JNIEnv\*,**ArrayType**, jboolean \*isCopy);

void Release**<PrimitiveType>**ArrayElements(JNIEnv\*,**ArrayType**,

**NativeType**\*,jint);

...

* Регистрация на „native“ функции (има два начина за обвързване на методи от Java с функции от C/C++; първия вече го показахме и него и ще използваме за имплментацията на **java-mpi**. Втория начин е чрез методи):

jint RegisterNatives(JNIEnv\*,jclass,const JNINativeMethod\*,jint);

jint UnregisterNatives(JNIEnv\*, jclass);

* Операции за наблюдение:

jint UnregisterNatives(JNIEnv\*,jclass);

jint MonitorExit(JNIEnv\*, jobject);

* „NIO“ поддръжка (това са функции позволяващи ползването на java.nio.Buffer):

jobject NewDirectByteBuffer(JNIEnv\*, void\*, jlong);

void\* GetDirectBufferAddress(JNIEnv\*, jobject);

jlong GetDirectBufferCapacity(JNIEnv\*, jobject);

* Възможност за ползване на **java reflection** (разработчиците могат да използват JNI за извикване на методи на Java или достъп до полета на Java; ако потребителят знае името и типа на методите или полетата. ***Java Core Reflection API*** функциите позволяват на програмистите да извършват операции с Java класове по време на изпълнение. JNI разполага с функции за преобразуване на **reflection** обекти в идентификатори):

jmethodID FromReflectedMethod(JNIEnv\*, jobject);

jobject ToReflectedMethod(JNIEnv\*, jclass,jmethodID,jboolean);

jobject ToReflectedField(JNIEnv\*,jclass,jfieldID,jboolean);

* Java VM интерфейс:

jint GetJavaVM(JNIEnv\*,JavaVM\*\*);

## АPI за извиквания (The Invocation API)

Идеята на този интерфейс е да се използва в обратна посока, а именно да позволи на една Java VM да се зареди в „native“ приложение. Възможностите му са да:

* Създава виртуална машина (JNI\_CreateJavaVM() и JNI\_DestroyJavaVM()).
* Свързва с вече работеща виртуална машина (AttachCurrentThread() и DetachCurrentThread()).

### JNI\_OnLoad

jint JNI\_OnLoad(JavaVM \*vm, void \*reserved);

Тази функция може да се имплементира от разработчика. Виртуалната машина извиква тази функция при зареждане на библиотеката (System..loadLibrary). JNI\_OnLoad() трябва да върне версията на JNI, която се изисква от „native“ библиотеката.

### JNI\_OnUnload

void JNI\_OnUnload(JavaVM \*vm, void \*reserved);

Тази функция е подобна на JNI\_OnLoad() с две малки разлики. Тя се изпълнява, когато класът заредил библиотеката бъде почистен от паметта.

Тези две функции не са нищо особено, но могат да се използват за неща, които искате да инициализирате при зареждане на библиотеката или съответно да изтриете при изтриване на класа заредил библиотеката.

### JavaVM

Това е указател, сочещ към таблицата с функции използвани от АPI функциите за извиквания.

typedef const struct JNIInvokeInterface \*JavaVM;

const struct JNIInvokeInterface ... = {NULL, NULL, NULL,

DestroyJavaVM, AttachCurrentThread, DetachCurrentThread,

GetEnv, AttachCurrentThreadAsDaemon

};

Може да се види, че липсват JNI\_DestroyJavaVM() и JNI\_CreateJavaVM(). Те са единствените функции, използвани директно, без участието на интерфейсен указател. В пример 5.7 може да се види примерно използване:

**Пример 5. 7** [ORACLE docs]

#include <jni.h> /\* where everything is defined \*/

...

JavaVM \*jvm; /\* denotes a Java VM \*/

JNIEnv \*env; /\* pointer to native method interface \*/

JavaVMInitArgs vm\_args; /\* JDK/JRE 6 VM initialization arguments \*/

JavaVMOption\* options = new JavaVMOption[1];

options[0].optionString = "-Djava.class.path=/usr/lib/java";

vm\_args.version = JNI\_VERSION\_1\_6;

vm\_args.nOptions = 1;

vm\_args.options = options;

vm\_args.ignoreUnrecognized = false;

/\* load and initialize a Java VM, return a JNI interface

\* pointer in env \*/

JNI\_CreateJavaVM(&jvm, (void\*\*)&env, &vm\_args);

delete options;

/\* invoke the Main.test method using the JNI \*/

jclass cls = env->FindClass("Main");

jmethodID mid = env->GetStaticMethodID(cls, "test", "(I)V");

env->CallStaticVoidMethod(cls, mid, 100);

/\* We are done. \*/

jvm->DestroyJavaVM();

Това е единствената част от JNI, която няма да използваме. Ще използваме само JNI\_OnLoad() и JNI\_OnUnload(), за да инициализираме нашия интерфейс.

## Опасности

JNI е опасен интерфейс. В много случай се препоръчва да се избягва и да се търсят алтернативни методи. Причината е в това, че интерфейсът е комуникационен слой между две коренно различни среди за програмиране. Има проблеми, които се създават при пренасяне на данните. Освен това, някои JNI функции прекратяват работата на „garbage collector“-а в някой случай, което прави тяхната употреба трудна и тромава, като освен това и забавя приложението.

# Java-Mpi спецификация

## Въведение

Java-Mpi е интерфейс позволяващ на Java процеси да комуникират помежду си. Той следва семантиката на MPI до толкова, доколкото това е възможно и Java го позволява. Това означава, че функционалност като P2P (Point to Point) и колективна комуникация са на разположение за ползване.

В следващите раздели ще дадем само спецификация на интерфейса. Имплементацията е описана в следващата глава.

## Point-To-Point Комуникация

Какво вече знаме MPI специфицира P2P комуникацията чрез изпращане на съобщения. Това прави и Java-MPI. Общо взето за всяка P2P функция от MPI може да намерим аналог в java-mpi. Една от основните в нашият интерфейс, и първата която ще отбележим, е MPI\_Send(). Тя има вида Comm.send(....).

public void send(Object buff,int offset,int count,

Datatype datatype,int dest,int tag,ByteBuffer objBuff)

Java-mpi също разполага и с останалите изпрощащи методи.

public void bsend(Object buff,int offset, int count,Datatype datatype, int dest, int tag,ByteBuffer objBuff) – MPI\_Bsend()

public void ssend(Object buff,int offset, int count,Datatype datatype, int dest, int tag,ByteBuffer objBuff) – MPI\_Ssend()

public void rsend(Object buff,int offset, int count,Datatype datatype, int dest, int tag,ByteBuffer objBuff) – MPI\_Rsend()

Абсолютно същият случай е и с не-блокиращите функции, MPI\_Isend, MPI\_Irsend и т.н. Те също са налични в java-mpi. Например:

public Request isend(Object buff,int offset,int count,Datatype datatype,int dest,int tag,ByteBuffer objBuff)

Разликата с класически MPI e, че нашият метод връща Request обект. Поради факта, че този вид функции са не блокиращи, се налага да се създаде метод, чрез който да се разбере дали операцията по получаване е приключила или не. Точно това е и ролята на Request обекта.

Класът Request имплементира методи като MPI\_Wait(), MPI\_Test() и т.н. Общо взето, той дава живот на всички функции от MPI изискващи като аргумент MPI\_Request. Подобна аналогия може да се забележи и по-нататък в описанието на java-mpi. Друг пример е MPI\_Send(). Тази функция изисква аргумент MPI\_Comm, следователно се намира в Comm.java. Друг пример е с MPI\_Win, който в Java е Win.java. Общо взето, това е логиката на целия интерфейс – опитваме се да остaнем максимално близо до обектно-ориентираната природа на MPI. В Таблица 2 може да се види пълният списък от обекти. А местоположението на методи, които ще са аналози на функциите от MPI, е избирано по две основни правила. Първото е – да съответстват на функцията използвана от дадения обект. Пример за това е MPI\_Send(), основния аргумент, който е необходим за извикването ѝ е комуникаторът MPI\_Comm. Другото правило е, Java класовете да съдържат методи, отговорни за тяхното управление и промяна.

**Пример 6.1**

...

Request req = MPI.COMM\_WORLD.isend(pArr,0,1,person,1,0,buff);

req.Wait();

}

else if(rank == 1) {

Person[] pArr = new Person[2];

Request req = MPI.COMM\_WORLD.irecv(buff,0,1,person,0,0);

Status stat = req.wait(pArr,0,1,person,buff);

// Do some operations!

}

MPI.Finalize();

...

Пример 6.1 показва не-блокираща комуникация - isend/irecv. Освен това, в примера може да се види и примерна комуникация между обекти, която ще опишем по-надолу в главата. Основното нещо, което искаме да покажем с този пример е начинът на комуникация. Както може да се види, потребителят използва обект за комуникация и в двата случая – и за изпращане, и за получаване. Това е COMM\_WORLD. След като вече е регистриран буфер за получаване с метода irecv, потребителят е длъжен, в някакъв момент от изпълнението на програмата, да извика метод за изчакване на съобщението или проверка на това, дали е получено. Това може да стане с:

public Status wait(Object[] objArr,int offset,

int count,Datatype type,ByteBuffer buff)

или

public Status test(Object[] objArr,int offset,

int count,Datatype type,ByteBuffer buff).

**Таблица 2.** Cъответствие на обекти и класове в MPI и Java-MPI.

|  |  |
| --- | --- |
| **C types/objects** | **Java classes** |
| MPI\_Comm | Comm/Intercomm/Intracomm/Graph/Cart |
| MPI\_Group | Group |
| MPI\_Datatype | Datatype/Byte/Int/Short….. |
| MPI\_File | File |
| MPI\_Status | Status |
| MPI\_Op | Op |
| MPI\_Request | Request |
| MPI\_Info | Info |
| MPI\_Win | Win |

MPI\_Bsend() и MPI\_Ibsend(), се използват заедно с MPI\_Buffer\_attach() и MPI\_Buffer\_detach(). Идеята е потребителя да предостави буфер, който да буферира съобщенията за изпращане:

for (i=0; i<m; i++) {

MPI\_Bsend(buf, n, MPI\_DOUBLE, ...);

}

Ако MPI не намери MPI\_Recv() в процеса получател, той ще се опита да буферира данните за изпращане. Когато буферът се препълни, MPI\_Bsend() ще върне грешка. След като MPI\_Bsend() има аналог, налага се и MPI\_Buffer\_attach() да има аналог. Именно затова добавяме функциите public static void buffer\_attach(ByteBuffer buff) и public static void buffer\_detach(ByteBuffer buff), намиращи се в MPI.java. Те са пълна аналогия на MPI\_Buffer\_attach() и MPI\_Buffer\_detach() с разликата, че в нашия случай ползваме директен „NIO“ буфер.

Поради липсата на указатели, в много случай използваме „NIO“ буфери за пренос на данни. Това многократно подобрява бързината, не се налага копиране на данни и е лесно за ползване. В Java го използваме, чрез java.nio.Buffer и всички негови наследници (java.nio.ByteBuffer, java.nioIntBuffer, ...), докато в C/C++ частта се използва като указател. Единственият недостатък на този метод е сравнително бавната алокация на буфера. Начинът, по който се заделя паметта, е чрез java.nio.ByteBuffer.allocateDirect(int size). Този метод заделя памет извън Java heap-a на една програма.

MPI\_Sendrecv() е функция обединяваща MPI\_Send() и MPI\_Recv() в едно извикване, а именно – получава от един процес и едновременно с това изпраща на друг. Тоест

if(rank == 0){

    MPI\_Send(sendBuffer,4,MPI\_INT,1,0,MPI\_COMM\_WORLD);

    MPI\_Recv(recvBuffer,4,MPI\_INT,1,0,MPI\_COMM\_WORLD,

MPI\_STATUS\_IGNORE);

}

else if(rank == 1){

    int buffer[4];

MPI\_Recv(recvBuffer,4,MPI\_INT,0,0,MPI\_COMM\_WORLD,

MPI\_STATUS\_IGNORE);

MPI\_Send(sendBuffer,4,MPI\_INT,0,0,MPI\_COMM\_WORLD);

}

може да бъде написан по този начин

if(rank == 0){

    MPI\_Sendrecv(sendBuffer,4,MPI\_INT,1,0,

recvBuffer,4,MPI\_INT,1,0,

MPI\_COMM\_WORLD,MPI\_STATUS\_IGNORE);

}

else if(rank == 1){

    int buffer[4];

    MPI\_Sendrecv(sendBuffer,4,MPI\_INT,0,0,

recvBuffer,4,MPI\_INT,0,0,

MPI\_COMM\_WORLD,MPI\_STATUS\_IGNORE);

}

и най-често се използва за изместване на информация в някаква посока.

Аналог на тази функция в java-mpi е методът намиращ се в Comm.java:

public Status sendrecv(Object sendbuff,int sendoffset,int sendcount,Datatype sendtype,int dest,int sendtag, ByteBuffer sendObjBuff,Object recvbuff,int recvoffset,int recvcount,Datatype recvtype, int source,int recvtag, ByteBuffer recvObjBuff)

MPI функциите MPI\_PROBE(), MPI\_IPROBE(), MPI\_MPROBE() и MPI\_IMPROBE() позволяват изпратено съобщение да се провери без всъщност да се получи. След това разработчикът може да избере метод, който да получи съответното съобщение. В java-mpi аналозите на тези функции се намират в Comm.java.

public Status probe(int source,int tag)

public Status iprobe(int source,int tag)

public Message mprobe(int source,int tag)

и т.н.

В MPI има така наречената „непроменлива“ (persistent) комуникация. Тя се осъществява с функциите MPI\_SEND\_INIT(), MPI\_BSEND\_INIT(), MPI\_SSEND\_INIT(), MPI\_START(), MPI\_RSEND\_INIT(), MPI\_RECV\_INIT()и MPI\_STARTALL(). Идеята на тези функции е да се оптимизират повтарящи се и непроменливи извиквания на функциите MPI\_Send() и MPI\_Recv(). Същите функции са на разположение и в java-mpi. Тук обаче имаме малка подробност – буфер аргументът ни може да е само директен „NIO“ буфер. Това е необходимо поради факта, че JNI интерфейса е тромав и опасен при използването на обикновен масив за тази функционалност. Основната причина, поради която специфицираме метода да използва само този метод за представяне на буфер е търсене на оптималност. Ако използваме обикновен масив, това ще изисква допълнителна имплементация, която ще е тромава и не оптимална. Така избягваме копирането на памет и запазване на глобални променливи. За това по-удобния и бърз начин е да се използва java.nio.Buffer. Методи за тази комуникация се намират в Comm.java:

public Request send\_init(Object buff,int offset,int count, Datatype type, int dest,int tag)

public Request recv\_init(Object buff,int offset,int count, Datatype type,int source,int tag)

и т.н

Може да се види, че персистентнаата комуникация връща Request обект, който отговаря за стартиране на комуникацията. А именно имплементира public void start().

В Таблица 3 е показан пълният списък от P2P MPI функции адаптирани за java-mpi. По-интересните аспекти и имплементационни проблеми идващи с тях вече бяха разгледани. Освен тях има и много други, който са разгледани в следващи раздели.

**Таблица 3.**

|  |  |
| --- | --- |
| **Java Method** | **MPI function** |
| public void send(Object buff, int offset, int count, Datatype datatype, int dest, int tag, ByteBuffer objBuff) | MPI\_Send |
| public void bsend(Object buff, int offset, int count, Datatype datatype, int dest, int tag, ByteBuffer objBuff ) | MPI\_Bsend |
| public void ssend(Object buff,int offset, int count, Datatype datatype, int dest, int tag,ByteBuffer objBuff) | MPI\_Ssend |
| public void rsend(Object buff, int offset, int count, Datatype datatype, int dest, int tag, ByteBuffer objBuff) | MPI\_Rsend |
| public Status recv(Object buff, int offset, int count, Datatype datatype, int source, int tag,ByteBuffer objBuff) | MPI\_Rcev |
| public Request isend(Object buff, int offset, int count, Datatype datatype, int dest, int tag, ByteBuffer objBuff) | MPI\_Isend |
| public Request ibsend(Object buff,int offset,int count, Datatype datatype,int dest,int tag,ByteBuffer objBuff) | MPI\_Ibsend |
| public Request issend(Object buff,int offset,int count, Datatype datatype,int dest,int tag,ByteBuffer objBuff) | MPI\_Issend |
| public Request irsend(Object buff,int offset,int count, Datatype datatype,int dest,int tag,ByteBuffer objBuff) | MPI\_Iresend |
| public Request irecv(Object buff,int offset,int count, Datatype datatype,int source,int tag) | MPI\_Irecv |
| public Status iprobe(int source,int tag) | MPI\_Iprobe |
| public Status probe(int source,int tag) | MPI\_Probe |
| public Request send\_init(Object buff,int offset,int count, Datatype type, int dest,int tag) | MPI\_Send\_init |
| public Request recv\_init(Object buff,int offset,int count, Datatype type, int source,int tag) | MPI\_Recv\_init |
| public Message mprobe(int source,int tag) | MPI\_Mprobe |
| public Status sendrecv(Object sendbuff,int sendoffset,int sendcount,Datatype sendtype,int dest,int sendtag, ByteBuffer sendObjBuff, Object recvbuff,int recvoffset,int recvcount,Datatype recvtype, int source,int recvtag, ByteBuffer recvObjBuff) | MPI\_Sendrecv |
| public Status sendrecv\_replace(Object buff,int offset,int count,Datatype datatype, int dest,int sendtag,int source,int recvtag,ByteBuffer objBuff) | MPI\_Sendrecv\_replace |

## Типове Данни

MPI разполага с голям и впечатляващ списък от вградени типове, като MPI\_INT, MPI\_CHAR, MPI\_UINT16 и т.н. Общо взето, всички характерни за C. Плюс това, MPI дава възможност на потребителя да дефинира динамично нови типове по време изпълнение на съответната програма. Java-MPI се опитва да следва този модел дотолкова доколкото това е възможно.

**Таблица 4.**

|  |  |
| --- | --- |
| **JAVA type** | **MPI type** |
| BYTE | MPI\_INT8\_T |
| SHORT | MPI\_INT16\_T |
| INT | MPI\_INT32\_T |
| LONG | MPI\_INT64\_T |
| FLOAT | MPI\_FLOAT |
| DOUBLE | MPI\_DOUBLE |
| CHAR | MPI\_UINT16\_T |
| BOOLEAN | MPI\_UINT8\_T |
| Java Object | - |

В Таблица 4 може да се видят Java типовете, имащи аналогични MPI типове. Това са всички основни типове за Java, които могат де се ползват от java-mpi. Разбира се, разликата в типовете на двата езика създава много и различни проблеми с комуникацията между двата езика. Начините, с който се опитваме да се справим с тях не са перфектни, но са начало.

Два от основните проблеми, с които ще започнем, като за начало, е разликата в указването на буфер, който ще се използва в MPI функция. В C, разработчикът спокойно може да изпълни Пример 6.2. Но в Java това не е възможно, или поне не съвсем. Имплементацията на метод позволяващ получаването и изпращането на само един елемент е възможно, но изисква използването на Java обекти съответстващи на съответните типове. Вместо да се използва int променлива, да се използва java.lang.Integer. Това е напълно възможно но не оптимално, за това сегашната имплементация на интерфейс не се заема да реши този проблем. Вместо това тя дава възможността за ползване само на три основни групи типове. Първият са едномерни масиви, вторият са директни „NIO“ буфери и не на последно място даваме възможност за използването на java обекти за комуникация.

**Пример 6.2**

int value;

MPI\_Send(&value,1,MPI\_INT,1,0,MPI\_COMM\_WORLD);

...

int value[40];

MPI\_Send(&value[30],10,MPI\_INT,1,0,MPI\_COMM\_WORLD);

В Пример 6.2, втората изпращаща функция използва индекс за определяне на това какво точно да изпрати. Този проблем го решаваме чрез допълнителен аргумент на всеки изпращащ или получаващ метод. Този аргумент отразява отместването, от което java-mpi трябва да започне да изпраща или получава данни.

Java дава голямо предимство, когато става дума за буфери. Както споменахме по-рано, езикът дава имплементация на буфери, които се запазват в „native” частта на едно приложение и това са “NIO” буфери. Тъй като „NIO“ буферите са коренно различни от обикновен Java масив, те изискват коренно различно управление. За да не се налага системата да определя какво е подадено, дали е масив или java.nio.Buffer, добавяме набор от допълнителни типове. Така вече не е необходимо системата да губи време да определя какво сме ѝ подали. В Таблица 5 може да се видят тези типове.

**Таблица 5.**

|  |  |
| --- | --- |
| **Java-mpi type** | **MPI type** |
| BYTE\_BUFFER(java.nio.ByteBuffer) | MPI\_INT8\_T |
| SHORT\_BUFFER(java.nio.ShortBuffer) | MPI\_INT16\_T |
| INT\_BUFFER(java.nio.IntBuffer) | MPI\_INT32\_T |
| LONG\_BUFFER(java.nio.LongBuffer) | MPI\_INT64\_T |
| FLOAT\_BUFFER(java.nio.FloatBuffer) | MPI\_FLOAT |
| DOUBLE\_BUFFER(java.nio.DoubleBuffer) | MPI\_DOUBLE |
| CHAR\_BUFFER(java.nio.CharBuffer) | MPI\_UINT16\_T |
| BOOLEAN\_BUFFER(java.nio.BooleanBuffer) | MPI\_UINT8\_T |

Всички типове в Таблица 5 трябва да се директни буфери. За да се създадат се използва java.long.ByteBuffer.allocateDirect. Това е методът на Java за създаване на директен „NIO“ буфер. Той се използва по-абсолютния начин, като и масивите.

Java-mpi дава възможност на потребителите да дефинират и свой собствени типове. Това се осъществява чрез Java Reflection – метод, или по-точно приложно-програмен интерфейс, на Java, позволяващ изследването или модифицирането на методи, полета, обекти, класове и интерфейси динамично, по време на изпълнението на една програма.

С други думи системата дава възможност за изпращане и получаване на обекти. За да работи този метод на комуникация, системата трябва да има начин, по който да разбере какви данни да вземе от обект и да ги изпрати. Конструкцията на един java-mpi тип се състой от два основни компонента, единия е самия клас, другия е Datatype обект.

Самият клас трябва задължително да дефинира сетери (setters) и гетъри (getters), които ще се ползват за достъп до полетата, определени за използване. Като те трябва да имат специална форма. За „set“ методите тя е void set**<Name>**(**PrimriveType**), докато „get“ методите имат формата **Type** get**<Name>**(). Като <Name> винаги трябва да започва с голяма буква. Не е задължително целият обект да се изпраща, т.е. възможно е само част от полетата да се определят за изпращане. Това се определя от Datatype обекта, втората част от целия метод за изпращане на обекти. Декларацията му става чрез един от конструкторите на обекта, а именно:

public Datatype(String[] names,Datatype[] types,int[] counts, Class<?> cls)

Аргументите му са, съответно:

* names, това е масив от низове съдържащи имената на полетата от класа, които участват в комуникацията. Общо взето те са същите описани по-горе като <Name>. В пример 6.3 е показано подробно тяхното използване.
* types, това са типовете на който съответстват имената. Като типовете могат да бъдат рекурсивно създадени, тоест възможно е един тип, дефиниран от потребителя, да съдържа поле за комуникация от друг тип, също дефиниран от потребител.
* counts, това е основният момент, в който се определя дали изпращаното е масив или само един елемент. При брой равен на 1 се приема, че е само а един елемент или поле от тип int, char, long и т.н. При брой по-голям от 1 се приема, че данните, който се подават, са масив и counts дефинира максималния му брой елементи. Системата ще даде грешка само при несъответствие за една стойност или масив, докато при неправилно определяне на максималната големина на масив ще замълчи, т.е. ще изреже или добави нули в съответния случай. Причината за добавянето на този аргумент е възможността да определим големената на обекта, който ще изпращаме. Това означава, че всеки обект ще има еднаква големина. Имайте предвид, че като основен тип все още не сме добавили java.long.String, следователно за сега трябва да се използва char[].
* cls, това е типът на обекта, [class name].class.

**Пример 6.3**

// execute point

static String[] h\_hapes = {"Id"};

static Datatype[] h\_types = {MPI.INT};

static int[] h\_counts = {1};

static Datatype here = new Datatype(h\_hapes, h\_types, h\_counts, Here.class);

static String[] names = {"Age","Name","Some","Here","H"};

static Datatype[] types = { MPI.INT,MPI.INT,MPI.LONG,here,here};

static int[] counts = {1,10,20,1,3};

static Datatype person = new Datatype(names, types,counts,Person.class);

public static void main(String[] args) {

        MPI.Init(args);

        Datatype.commit(person);

        Datatype.commit(here);

        ByteBuffer buff = Datatype.allocBuff(1024);

    int rank = MPI.COMM\_WORLD.rank();

    if(rank == 0) {

     Person p =new Person(new int[]{0x000000FF,0x000000FF,0x000000FF},0x55AA55AA);

     Here h = new Here();

     Here[] hs = new Here[3];

     hs[0] = new Here(); hs[0].setId(13);

     hs[1] = new Here(); hs[1].setId(14);

     hs[2] = new Here(); hs[2].setId(15);

     h.setId(12);

     p.setHere(h); p.setH(hs);

     long[] array = new long[] {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};

     p.setSome(array);

     // send the object

     Person pArr[] = new Person[1];

     pArr[0] = p;

     MPI.COMM\_WORLD.send(pArr,0,1,person,1,0,buff);

    }

    else if(rank == 1) {

        Person[] pArr = new Person[2];

        Status stat = MPI.COMM\_WORLD.recv(pArr,0,1,person,0,0,buff);

    }

    MPI.Finalize();

}

// Here type

public class Here {

    private int id;

    private int id2;

    public int getId() {return id;}

    public void setId(int id) {this.id = id;}

    public int getId2() {return id2;}

    public void setId2(int id2) {this.id2 = id2;}

}

// Person type

public class Person{

private int age;

private int[] name;

private long[] some;

private Here here;

private Here[] h;

public Here getHere() {return here;}

public void setHere(Here here) {this.here = here;}

public Person(){}

public Person(int[] name,int age){this.name = name;this.age = age;}

public int getAge() {return age;}

public void setAge(int age) {this.age = age;}

public int[] getName() {return name;}

public void setName(int[] name) {this.name = name;}

public long[] getSome() {return some;}

public void setSome(long[] some) {this.some = some;}

    public Here[] getH() {return h;}

    public void setH(Object[] h) {

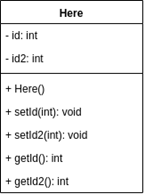
        this.h = new Here[h.length];

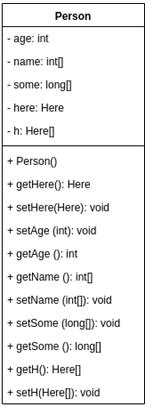
        for(int i = 0;i < this.h.length;i++) {

            this.h[i] = (Here)h[i];

  ...

Пример 6.3 започва с инициализация на потребителските типове, а именно





Фиг. 6.1

static Datatype here = new Datatype(h\_hapes, h\_types, h\_counts, Here.class);

static Datatype person = new Datatype(names, types,counts,Person.class);.

като самите класове имат вида, показана на фиг. 6.1

За да може съответните типове да се ползват те трябва да се обявят в системата. Т.е. да се изпълнят следните два реда:

Datatype.commit(person);

Datatype.commit(here);

Веднъж направена цялата тази процедура, типовете може да се използват за изпращане, получаване и от части за колективни операции.

В много от методите за P2P комуникация може да се намери допълнителен „java.nio.Buffer“ аргумент. Разбира се той може да се избегна, като вместо референция към него се подадете „null“. Идеята му е да се използва за прехвърляне на данните от един или повече обекти от java към C и обратно. Това се прави поради оптимизационни причини. Така или иначе тези данни трябва да се прехвърлят към C пространството, просто използването на този метод е по бърз и безполезен начин. Има два начина, по които може да се предостави „NIO“ буфер. Първия е, потребителя да го предостави чрез аргумент. Другия е да се зададе „null“ като аргумента използван за буфера, **ByteBuffer sendObjBuff, ByteBuffer recvObjBuff**, по този начин се оставя на системата да определи начина, по който ще се създаде „NIO“ буфер или изобщо начин, по който ще се прехвърлят необходимите данни. В пример 6.3 може да видите това.

Има една малка особеност при ползването на не-блокиращи опера­ции и потребителски типове. Преобразуването на комуникаци­онните данни в java обект не се прави от системата директно, а трябва да се извърши от потребителя. Методите, по които това може да стане са:

public Status wait(Object[] objArr,int offset,

int count,Datatype type,ByteBuffer buff)

или

public Status test(Object[] objArr,int offset,int count,Datatype type,ByteBuffer buff)

**Пример 6.4**

ByteBuffer buff = ByteBuffer.allocateDirect(1024);

Request req = MPI.COMM\_WORLD.ibcast(pArr,0,1,person,0,buff);

buff.position(0);

buff.limit(buff.capacity());

req.wait(pArr, 0, 1, person, buff);

При test метода, същинското преобразуване става, чак когато тестът е положителен, т.е. операцията е приключила. Това важи, както за P2P комуникации, така и за колективни комуникации.

## Колективна Комуникация

Както вече казахме, колективната комуникация включва в себе си процеси от една или повече групи, позволявайки им да извършват общи комуникационни операции. java-mpi дава аналози на повечето от тях, дори MPI\_Reduce и операции свързани с потребителски типове. В Таблица 6 са показани само част от функциите и методите от MPI, които имат своето място и в java-mpi.

**Таблица 6.**

|  |  |
| --- | --- |
| **Java-Mpi method** | **MPI function** |
| public void bcast(Object buff,int offset,int count, Datatype datatype,int root, ByteBuffer objBuff) | MPI\_Bcast |
| public void scatter(Object sendbuff, int sendoffset,int sendcount, Datatype sendtype,Object recvbuff, int recvoffset, int recvcount, Datatype recvtype, int root, ByteBuffer sendObjBuff, ByteBuffer recvObjBuff) | MPI\_Scatter |
| public void gather(Object sendbuff,int sendoffset,int sendcount, Datatype sendtype,Object recvbuff,int recvoffset,int recvcount, Datatype recvtype, int root, ByteBuffer sendObjBuff, ByteBuffer recvObjBuff) | MPI\_Gather |
| public void allgather(Object sendbuff, int sendoffset, int sendcount, Datatype sendtype, Object recvbuff, int recvoffset, int recvcount, Datatype recvtype, ByteBuffer sendObjBuff, ByteBuffer recvObjBuff) | MPI\_Allgather |
| public void alltoall(Object sendbuff,int sendoffset, int sendcount, Datatype sendtype,Object recvbuff, int recvoffset, int recvcount, Datatype recvtype, ByteBuffer sendObjBuff, ByteBuffer recvObjBuff) | MPI\_Alltoall |
| public void reduce(Object sendbuff,Object recvbuff, int sendoffset,int recvoffset,int count,Datatype datatype,Op op,int root, ByteBuffer sendObjBuff, ByteBuffer recvObjBuff) | MPI\_Reduce |
| public void allreduce(Object sendbuff,Object recvbuff,int sendoffset,int recvoffset, int count,Datatype datatype,Op op,ByteBuffer sendObjBuff, ByteBuffer recvObjBuff) | MPI\_Allreduce |
| public void scan (Object sendbuff,Object recvbuff,int sendoffset, int recvoffset, int count,Datatype datatype,Op op,ByteBuffer sendObjBuff, ByteBuffer recvObjBuff) | MPI\_Scan |
| public void barrier() | MPI\_Barrier |
| public void exscan (Object sendbuff,Object recvbuff,int sendoffset,int recvoffset, int count,Datatype datatype,Op op,ByteBuffer sendObjBuff, ByteBuffer recvObjBuff) | MPI\_Xscan |
| public Request ibcast(Object buff,int offset, int count,Datatype datatype,int root, ByteBuffer objBuff) | MPI\_Ibcast |
| public Request iscatter(Object sendbuff, int sendoffset,int sendcount, Datatype sendtype,Object recvbuff,int recvoffset, int recvcount, Datatype recvtype, int root, ByteBuffer sendObjBuff, ByteBuffer recvObjBuff) | MPI\_Iscatter |
| public Request igather(Object sendbuff,int sendoffset,int sendcount, Datatype sendtype, Object recvbuff,int recvoffset,int recvcount, Datatype recvtype,  int root,ByteBuffer sendObjBuff, ByteBuffer recvObjBuff) | MPI\_Igather |
| public Request iallgather(Object sendbuff,int sendoffset,int sendcount, Datatype sendtype, Object recvbuff,int recvoffset,int recvcount, Datatype recvtype,ByteBuffer sendObjBuff,ByteBuffer recvObjBuff) | MPI\_Iallgather |
| public Request ialltoall(Object sendbuff,int sendoffset,int sendcount, Datatype sendtype, Object recvbuff,int recvoffset,int recvcount,Datatype recvtype,  ByteBuffer sendObjBuff,ByteBuffer recvObjBuff) | MPI\_Ialltoall |
| public Request ireduce(Object sendbuff,Object recvbuff, int sendoffset, int recvoffset,int count,Datatype type,Op op,int root, ByteBuffer sendObjBuff, ByteBuffer recvObjBuff) | MPI\_ireduce |
| public Request iallreduce(Object sendbuff,Object recvbuff, int sendoffset, int recvoffset,int count,Datatype type,Op op, ByteBuffer sendObjBuff, ByteBuffer recvObjBuff) | MPI\_Iallreduce |
| public Request iscan(Object sendbuff,Object recvbuff, int sendoffset,int recvoffset,int count,Datatype type,Op op,ByteBuffer sendObjBuff, ByteBuffer recvObjBuff) | MPI\_Iscan |
| public Request iexscan(Object sendbuff,Object recvbuff, int sendoffset, int recvoffset,int count,Datatype type,Op op,ByteBuffer sendObjBuff, ByteBuffer recvObjBuff) | MPI\_Iexscan |

За съжаление не всички функции имат аналози. MPI функции като MPI\_SCATTERV(), MPI\_GATHERV(), MPI\_ALLTOALLW() и MPI\_REDUCE\_SCATTER() за сега не намират място в java-mpi. За наше щастие, употребата им може да се избегне. Тяхната функционалност е възможно да се имплементира чрез тези, които вече имаме налице. Това обаче не означава, че в бъдеще те няма да се адаптират, напротив с развитието на проекта все повече и повече MPI функционалности ще намират мястото си в Java-MPI.

Част от MPI функциите приемат операция, като аргумент. MPI\_Op е операция изпълнявана от функции като MPI\_Reduce. Те свеждат резултата от комуникацията до една стойност.

**Пример 6.5**

int rank; int size; int value; int recvBuff;

MPI\_Init(&argc,&argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD,&rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD,&size);

value = rank\*rank;

MPI\_Reduce(&value,&recvBuff,1,MPI\_INT,MPI\_SUM,0,MPI\_COMM\_WORLD);

if(rank == 0){

printf("Sum Values: %d\n",recvBuff);

}

MPI\_Finalize();

return 0;

Абсолютно същата функционалност предлага и java-mpi.

**Пирмер 6.6**

public static void main(String[] args) {

MPI.Init(args);

int[] sendbuff = new int[2];

int[] recvbuff = new int[1];

int rank = MPI.COMM\_WORLD.rank();

sendbuff[0] = rank;

MPI.COMM\_WORLD.reduce(sendbuff, recvbuff, 0, 0, 1,

MPI.INT, MPI.SUM, 0,null,null);

if(rank == 0){

System.out.println(recvbuff[0]);

}

MPI.Finalize();

}

Всички предефинирани операции са налични за ползване и дефинирани в java.mpi. MPI позволява също така и дефинирането на потребителски операции. Тук обаче те се дефинират малко по-сложно от това в MPI. Операции с прости типове, като int или short, могат да се дефинират чрез имплементация на Java interface. Те са: IBooleanOp, IByteOp, IShortOp, IIntOp и т.н. Операции от по-сложен характер, т.е. такива с обекти, могат да се дефинират чрез наследяване на абстрактния клас ObjectOp.

Начинът на дефиниране е много прост. След като се създаде клас, който съдържа в себе си операцията, т.е. се имплементира IIntOP, ... или се наследява ObjectOp, се подава в конструктора на Op класа public Op(Object operation,boolean isCommute). Този клас от своя страна отговаря за по-нататъшното управление, и до известна степен отговаря на MPI\_Op в MPI. За да е активна операцията тя трябва да се създаде. Това става с извикването на „create“ метода на обекта, public void create(Datatype type). В пример 6.7 може да се види пример за използването на въпросната функция.

**Пример 6.7**

static String[] h\_hapes = {"Id"};

static Datatype[] h\_types = {MPI.INT};

static int[] h\_counts = {1};

static Datatype her = new Datatype(h\_hapes, h\_types, h\_counts, Here.class);

public static void main(String[] args) {

MPI.Init(args);

Op op = new Op(new ObjectOp(her) {

@Override

public Object operation(Object a, Object b) {

Here \_a = (Here)a;

Here \_b = (Here)b;

// operation

...

\_b.setId(\_b.getId()+\_a.getId());

return \_b;

}

} , true);

Datatype.commit(her);

op.create(her);

int rank = MPI.COMM\_WORLD.rank();

Here[] hSend = new Here[10];Here[] hRecv = new Here[3];

hSend[0] = new Here();hSend[0].setId(rank);

ByteBuffer buff = ByteBuffer.allocateDirect(124);

ByteBuffer buff1= ByteBuffer.allocateDirect(124);

long start = System.nanoTime();

MPI.COMM\_WORLD.reduce(hSend, hRecv, 0, 0, 1, her, op, 0, buff,buff1);

long end = System.nanoTime();

Utils.sleepRand(500);

System.out.printf("%f\n", (end-start)/1000000.0);

if(rank == 0){

// operation

.....

В пример 6.7 може да се види в действие и потребителски тип и потребителска операция. Първото, което правим е да създадем операция, обект от тип Op. Той от своя страна съдържа друг обект имплементиращ същинската операция, IIntOp или ObjectOp.

**Пример 6.7**

Op op = new Op(new ObjectOp(her) {

@Override

public Object operation(Object a, Object b) {

Here \_a = (Here)a;

Here \_b = (Here)b;

// operation

}

} , true);

След като това е направено, можем да преминеме към самото създаване на операцията. Нейното обявяване в системата става чрез op.create(her);, като добавяме и съответния тип. След това се преминава към използване. Използването на потребителската операция става по абсолютно същият начин както и тези, предоставени от java-mpi. То обаче идва със забавяне, използването на потребителски дефинирани операции изисква доста комуникация между между java и C. В резултат, на което операции като Reduce стават доста тромави и бавни.

За всички останали операции, обаче, няма разлика. Те са специфицирани на база на P2P комуникационните методи. Тоест разликата между тях и колективните методи е в това, че колективните трябва да се извикат от всички процеси в един комуникатор (Comm.java). Това са методи като MPI\_Bcast(), MPI\_Gather(), MPI\_Scatter() и т.н.

## Групи, Контекст, Комуникатори

Вече обяснихме концепцията на MPI относно групи, комуникатори и контекст. Това което ще направим тук е да опишем това, което прави java-mpi, за да постигне подобна функционалност. Разликата между семантиката на MPI и семантиката на java-mpi е почти никаква – принципът е един и същи.

MPI функции като MPI\_Group\_size(), MPI\_Group\_rank() и всички подобни, отговарящи за достъп до определена информация на група се имплементират от Group.java.

**Таблица 7.**

|  |  |  |
| --- | --- | --- |
| **Java Object** | **Java-mpi method** | **MPI function** |
| Comm | public Group group() | MPI\_COMM\_GROUP |
| Group | public Group union(Group group) | MPI\_GROUP\_UNION |
| Group | public Group intersection(Group group) | MPI\_GROUP\_INTERSECTION |
| Group | public Group difference(Group group) | MPI\_GROUP\_DIFFERENCE |
| Group | public Group incl(int offset,int n,int[] ranks) | MPI\_GROUP\_INCL |
| Group | public Group excl(int offset,int n,int[]ranks) | MPI\_GROUP\_EXCL |

В Таблица 7 са изброени всички методи за създаване на група.

За комуникаторите подходът е абсолютно същият. В Comm.java могат да се намерят аналози на функции като: MPI\_COMM\_SIZE(), MPI\_COMM\_RANK() и MPI\_COMM\_COMPARE(). Конструкцията се осъществява чрез методите, описани в Таблица 8.

**Таблица 8.**

|  |  |  |
| --- | --- | --- |
| **Java Object** | **Java-mpi method** | **MPI function** |
| Comm | public Comm dup() | MPI\_COMM\_DUP |
| Comm | public Intracomm create(Group group) | MPI\_COMM\_CREATE |
| Intracomm | public Intracomm create(Group group,int tag) | MPI\_COMM\_CREATE\_GROUP |
| Comm | public Intracomm split(int color,int key) | MPI\_COMM\_SPLIT |
| Comm | public Intracomm split\_type(int split\_type, int key, Info info) | MPI\_COMM\_SPLIT\_TYPE |

MPI разполага със специални функции за управление на inter-комуникатори, затова с такива разполага и java-mpi. Те могат да се видят в Таблица 10.

**Таблица 10.**

|  |  |
| --- | --- |
| **Java method** | **MPI function** |
| public int size() | MPI\_COMM\_REMOTE\_SIZE |
| public Group remote\_group() | MPI\_COMM\_REMOTE\_GROUP |
| public void create(Intracomm local\_comm,int local\_leader, Intracomm peer\_comm, int remote\_leader, int tag) | MPI\_INTERCOMM\_CREATE |
| public Intracomm merge(boolean high) | MPI\_INTERCOMM\_MERGE |

Пример 6.8 показва един начин за създаване на група.

**Пример 6.8**

...

Group world\_group = MPI.COMM\_WORLD.group();

int n = 5;

int[] ranks = new int[]{1,3,5,7,9};

Group prime\_group = world\_group.incl(0, n, ranks);

Comm comm = MPI.COMM\_WORLD.create(prime\_group, 0);

int prime\_rank = -1, prime\_size = -1;

int comm\_result = -1;

if(!comm.isNull()){

prime\_rank = comm.rank();

prime\_size = comm.size();

comm\_result = MPI.COMM\_WORLD.compare(comm);

}

...

MPI кешира атрибути и информация, свързани с три основни типа обекти – комуникатори, прозорци (Windows) и типове данни. На този етап MPI кеширане не се поддържа от java-mpi.

## Виртуални Топологии

MPI разполага с декартова и граф топологии, както вече споменахме. java-mpi се опитва да се доближи максимално до тях, макар и това да не е от най-важните аспекти от интерфейса. Идеята е java-mpi да даде аналози на най-използваните MPI функции, за да може да видим дали те ще бъдат полезни за Java. Точно за това няма смисъл да се фокусираме върху функционалност, която дори може да не се използва.

Начинът, по който java-mpi ги имплементира е чрез допълнителни обекти наследяващи Intercomm класа. Те са Cart.java (Декартова топология), Graph.java (граф топология) и DistGraph.java (разпределен граф топология).

**Таблица 11.** **Декартови методи**

|  |  |  |
| --- | --- | --- |
| **Java-mpi method** | **Java class** | **MPI function** |
| public Cart cart\_create(int dims\_offset,int ndims,int[] dims, int per\_offset,int[] periods, boolean reorder) | Intracomm | MPI\_CART\_CREATE |
| public void coords(int rank,int offset,int maxdims, int[] coords) | Cart | MPI\_CART\_COORDS |
| public void get(int offsetdims,int maxdims, int[] dims,int offsetperiods,int[] periods,int offsetcoords,int[] coords) | Cart | MPI\_CART\_GET |
| public int source(int direction,int disp) | Cart | MPI\_CART\_SHIFT |
| public int dest(int direction,int disp) | Cart | MPI\_CART\_SHIFT |
| public Cart sub(int offset,int[] remain\_dims) | Cart | MPI\_CART\_SUB |
| public int dims() | Cart | MPI\_CARTDIMS\_GET |
| public static void dims\_create(int nnodes,int offset\_dims,int ndims,int[] dims) | MPI | MPI\_DIMS\_CREATE |

**Таблица 12. Граф методи**

|  |  |  |
| --- | --- | --- |
| **Java-mpi method** | **Java class** | **MPI function** |
| public Graph graph\_create(int offset\_nodes,int nnodes,int[] indexes, int offset\_edges,int[] edges, boolean reorder) | Intercomm | MPI\_GRAPH\_CREATE |
| public void get(int offsetindx,int maxindex, int offsetedges,int maxedges,int indx[], int edges[]) | Graph | MPI\_GRAPH\_GET |
| public int nnodes() | Graph | MPI\_GRAPHDIMS\_GET |
| public int nedges() | Graph | MPI\_GRAPHDIMS\_GET |
| public void neighbors(int rank,int offset,int maxxneighbors,int[] neighbors) | Graph | MPI\_GRAPH\_NEIGHBORS |
| public int neighbors\_count(int rank) | Graph | MPI\_GRAPH\_NEGHBORS\_COUNT |

DistGraph.java дава имплементация на разпределен граф. Методът, чрез който се създава обектът е public Intracomm dist\_graph\_create(…);.

public Intracomm dist\_graph\_create(int offsetsources, int n, int[] sources,int offsetdegrees,int[] degrees,

int offsetdestinations, int[] destinations,

int offsetweights, int[] weights,Info info,

boolean reordered)

Този метод се намира в Intracomm и е аналог на MPI\_DIST\_GRAPH\_CREATE(). По принцип, това не е единственият начин за създаване на такъв тип тополгия. Другият е MPI\_DIST\_GRAPH\_CREATE\_ADJACENT(), но за момента тази MPI функция няма аналог в java-mpi.

Друга MPI функция, която има аналог в java-mpi е MPI\_TOPO\_TEST(). Тя се намира в Intracomm класа, public int topo\_test(). Методът тества комуникатор за тополгия, възможните резултати от него се намират в съответен MPI клас.

....

/\*

In addition, there are 5 predefined window attributes that aredefined for every window

\*/

public static final int WIN\_BASE = 0x66000001;

public static final int WIN\_SIZE = 0x66000003;

public static final int WIN\_DISP\_UNIT = 0x66000005;

public static final int WIN\_CREATE\_FLAVOR = 0x66000007;

public static final int WIN\_MODEL = 0x66000009;

/\* for subarray and darray constructors \*/

public static final int ORDER\_C = 56;

public static final int ORDER\_FORTRAN = 57;

public static final int DISTRIBUTE\_BLOCK = 121;

public static final int DISTRIBUTE\_CYCLIC = 122;

public static final int DISTRIBUTE\_NONE = 123;

public static final int DISTRIBUTE\_DFLT\_DARG = -49767;

// topologies

public static final int GRAPH = 1;

public static final int CART = 2;

public static final int DIST\_GRAPH = 3;

...

## Управления на средата

java-mpi разполага с пълния набор от MPI функции за управление на средата, като изключим някои методи за прихващане на грешки (error handling). За работата те не са особено приоритетни, поради необходимостта да се провери дали изобщо ще може да се използва интерфейсът или не. Другият проблем, който те пораждат е необходимостта да променим до голяма степен имплементацията на MPI, кято използваме. Това не означава, че съвсем сме забравили за обработка на грешки. В повечето случай на фатални грешки системата прекратява работата си. Друг метод, за прихващане на грешки е чрез кодове връщани от MPI функции. Това са кодове като MPI\_ERR\_BUFFER, MPI\_ERR\_COUNT, MPI\_ERR\_TYPE, MPI\_ERR\_TAG и т.н – пълният им списък може да се види в mpi.h. Всеки метод от java-mpi връща подобен резултат под формата на изключение (exception). Те се имплементират в mpi.exception пакета. Някой от тях са mpi.exception.BufferException, mpi.exception.CountException, mpi.exception.TypeException и т.н.

**Таблица 12** **Методи за управление.**

|  |  |
| --- | --- |
| **Java-mpi method** | **MPI function** |
| public static void Init(String[] args) | MPI\_Init |
| public static int Init\_thread(String[] args,int required) | MPI\_Init\_thread |
| public static void Finalize() | MPI\_Finalize |
| public static String processor\_name() | MPI\_Processor\_name |

Също там могат да се види и функция за проверка версията на интерфейса:

public int java\_mpi\_interface\_version().

Функцията MPI\_ABORT(), която се използва за прекратяване на работата на процеси намиращи се в един комуникатор, също e имплементирана в java-mpi:

public void abort(int errorcode)

## UI компонент, mpiexec

Изпълнението на една java програма с java-mpi, също не се различава много от тази на MPI. Идеята е да се използва по абсолютно същия начин, както в MPI. Така се спестява необходимостта от нова имплементация на UI компонент. Използването на mpiexec програмата е почти същото, разликата идва от там, че не се изпълнява директно алгоритъма, а се изпълнява JVM, която изпълнява алгоритъма.

**Пример 6.15**

**mpiexec -n 10 java -Djava.library.path=/home/nikolay/workspace/mpich-install-java/lib test.Test**

Разбира се, mpiexec трябва да се изпълни така, че java да може да намери, както основния клас (този, който съдържа main метода), така и всякакви библиотеки, било то java-mpi или други. Пример 6.15 илюстрира точно това. В него се изпълняват 10 процеса. Oсновния клас е test.Test и всички необходими библиотеки се намират в папката /home/nikolay/workspace/mpich-install-java/lib.

## Информационен обект

Както може да се забележи, повечето методи, описани до сега, приемат Info обекти. Това е напълно нормално, тези обекти са налице и в MPI. Един такъв обект се съставя от два параметъра – ключ и стойност. Този аспект от MPI е напълно приложен и в java-mpi.

Цялата функционалност се предоставя чрез Info.java. Не всички функции са налице, но основните са тук и са показани в Таблица 13.

**Таблица 13.**

|  |  |
| --- | --- |
| **Java-mpi method** | **MPI function** |
| public void create() | MPI\_INFO\_CREATE |
| public void delete(String key) | MPI\_INFO\_DELETE |
| public Info dup() | MPI\_INFO\_DUP |
| public void free() | MPI\_INFO\_FREE |
| public String get(String key) | MPI\_INFO\_GET |
| public int nkeys() | MPI\_INFO\_NKEYS |
| public String get\_nthkey(int n) | MPI\_INFO\_GET\_NTHKEY |
| public void set(String key,String value) | MPI\_INFO\_SET |

## Създаване на процеси и тяхното управление

За MPI, mpiexec не е единствената входна точка. Стандартът разполага с още два начина за осъществяване на връзка между процеси. Това са динамично създаване на процеси и динамично осъществяване на връзка (клиент-сървър).

Динамичното създаване на процес се осъществява чрез MPI\_COMM\_SPAWN(). Аналогичен метод е на разположение и в java-mpi., намиращ се в Intracomm.java.

**Пример 6.9**

…

public static final int NUM\_SPAWNS = 2;

public static void main(String[] args) {

int np = NUM\_SPAWNS;

int[] errcodes = new int[NUM\_SPAWNS];

String[] \_argv = {

"-Djava.library.path=/home/nikolay/workspace/mpich-"

"install-java/lib", "test.Spawn",null };

MPI.Init(args);

Intercomm parentcomm = MPI.parent();

if (parentcomm.isNull()){

parentcomm = MPI.COMM\_WORLD.spawn("java", \_argv, np, MPI.INFO\_NULL, 0,0,errcodes);

System.out.printf("I'm parent.\n");

}

else{

System.out.printf("I'm spawned.\n");

}

System.out.flush();

MPI.Finalize();

}

Както може да се види в Пример 6.9, използването на метода е напълно идентично със семантиката на MPI. В Пример 6.10 може да се види аналогичен код използващ MPI.

**Пример 6.10**

#define NUM\_SPAWNS 2

int main( int argc, char \*argv[] )

{

int np = NUM\_SPAWNS; int errcodes[NUM\_SPAWNS];

MPI\_Comm parentcomm, intercomm = MPI\_COMM\_NULL;

MPI\_Init( &argc, &argv );

MPI\_Comm\_get\_parent( &parentcomm );

if (parentcomm == MPI\_COMM\_NULL){

MPI\_Comm\_spawn( "./spawn", MPI\_ARGV\_NULL, np, MPI\_INFO\_NULL, 0,MPI\_COMM\_WORLD, &intercomm, errcodes );

// in the parent

}

else{/\* in the child\*/ }

fflush(stdout);

if(intercomm != MPI\_COMM\_NULL) MPI\_Comm\_free(&intercomm);

if(parentcomm != MPI\_COMM\_NULL) MPI\_Comm\_free(&parentcomm);

MPI\_Finalize();

return 0;

}

Вижда се, че промените са много малки – главно в начина, по който предаваме аргументите на системата за изпълнение. За да не променяме драстично метода, по който една MPI имплементация (в MPICH) създава нов процес, използваме абсолютно същия метод, който използваме и в mpiexec. А именно, основната програма, която ще се изпълни е java, а в аргументите указваме на системата кой e класът и къде да го намери. Освен това, могат да се добавят и всякакви помощни флагове и аргументи.

В Пример 6.9 може да види още, че основната команда, която се изпълнява е java, “parentcomm = MPI.COMM\_WORLD.spawn("java",...“, а останалите аргументи, като флагове, класове и други, се добавят към масива с аргументи String[] argv. След като се подкарва новият процес, java-mpi разполага с public static Intercomm parent(). Този метод връща „inter“ комуникатор към бащиния си процесор, този от който е бил създаден или MPI\_COMM\_NULL ако няма такъв. Методът е анлогичен на MPI\_COMM\_GET\_PARENT.

Вторият метод за осъществяване на динамична комуникация е базиран на клиент-сървър методологията. И тук java-mpi дава пълна аналогия с MPI функциите. Всички методи се намират в mpi.java

**Таблица 15.**

|  |  |
| --- | --- |
| **Java-mpi methods** | **MPI functions** |
| public static String open\_port(Info info) | MPI\_OPEN\_PORT |
| public static void close\_port(String portname) | MPI\_CLOSE\_PORT |
| public static void publish\_name(String servicename,Info info,String portname) | MPI\_PUBLISH\_NAME |
| public static void unpublish\_name(String servicename,Info info,String portname) | MPI\_UNPUBLISH\_NAME |
| public static String lookup\_name(String servicename,Info info) | MPI\_LOOKUP\_NAME |

Освен това, Intercomm добавя и метода public Intercomm accept(String portname,Info info,int root), който осъществява същинската връзка и е аналогия на MPI\_COMM\_ACCEPT().

**Пример 6.11**

public static void main(String[] args) {

String port1,port2;Intercomm comm1,comm2;

int[] data = new int[10];

MPI.Init(args);

int size = MPI.COMM\_WORLD.size(); int rank = MPI.COMM\_WORLD.rank();

if (size < 3){

// need more than 3 processes

MPI.Finalize();

return;

}

if (rank == 0){

// opening the ports

port1 = MPI.open\_port(MPI.INFO\_NULL);

port2 = MPI.open\_port(MPI.INFO\_NULL);

// taking the port names

char[] \_port1 = port1.toCharArray();

char[] \_port2 = port2.toCharArray();

// send a port name to the process 1

MPI.COMM\_WORLD.send(\_port1, 0, \_port1.length, MPI.CHAR, 1, 0, null);

// send the port name to the process 2

MPI.COMM\_WORLD.send(\_port2, 0, \_port2.length, MPI.CHAR, 2, 0, null);

// accepting a connection between process 0 and 1

comm1 =MPI.COMM\_SELF.accept(port1, MPI.INFO\_NULL, 0);

// accepting a connection between proces 0 and 2

comm2 =MPI.COMM\_SELF.accept(port2, MPI.INFO\_NULL, 0);

MPI.close\_port(port1); MPI.close\_port(port2);

// send data to process 1 via port connection

data[0] = 1; data[1] = 2; data[2] = 3;

comm1.send(data, 0, 3, MPI.INT, 0, 0, null);

// send data to proces 2 via port connection

data[0] = 4; data[1] = 5; data[2] = 6;

comm2.send(data, 0, 3, MPI.INT, 0, 0, null);

System.out.flush();

comm1.disconnect();comm2.disconnect();

}

else if (rank == 1){

// process 1 connection

char[] \_port1 = new char[1028];

MPI.COMM\_WORLD.recv(\_port1, 0, 1028, MPI.CHAR, 0, 0, null);

port1 = new String(\_port1);

comm1=MPI.COMM\_SELF.connect(port1, MPI.INFO\_NULL, 0);

comm1.recv(data, 0, 3, MPI.INT, 0, 0, null);

// some operation ...

comm1.disconnect();

}

else if (rank == 2){

// process 2 connection

char[] \_port2 = new char[1028];

MPI.COMM\_WORLD.recv(\_port2, 0, 1028, MPI.CHAR, 0, 0, null);

port2 = new String(\_port2);

/\* make sure process 1 has time to do the connect before this process attempts to connect \*/

try{ Thread.sleep(3000);}

catch(Exception e){ e.printStackTrace(); }

comm2=MPI.COMM\_SELF.connect(port2, MPI.INFO\_NULL, 0);

comm2.recv(data, 0, 3, MPI.INT, 0, 0, null);

comm2.disconnect();

}

MPI.COMM\_WORLD.barrier();

MPI.Finalize();

}

Пример 6.11 точно показва клиен-сървър комуникацията. Като за начало, тестовият алгоритъм трябва да се изпълни върху минимум 3 процеса. Процесът с ранг 0 играе ролята на разпределител и отваря два порта. Тези два порта ще са връзка с процеси 1 и 2. Тъй като MPI не разполага с метод за изпращане на портове, използвана е send/recv комуникация, за да може процеси 1 и 2 да научат имената (идентификаторите) на отворените от процес 0 портове. След като цялата тази процедура е изпълнена, се преминава към същинското осъществяване на връзка, а именно

comm1 = MPI.COMM\_SELF.accept(port1, MPI.INFO\_NULL, 0);

и

comm1 = MPI.COMM\_SELF.connect(port1, MPI.INFO\_NULL, 0);

В резултат на което, се получава „inter” комуникация между процеси 0 и 1, както и между процеси 0 и 2. Веднъж създадени, комуникаторите се използват по стандартният начин.

## Едностранна комуникация

Едностранната комуникация обхваща MPI\_Win обекта. В java.mpi е предоставен абсолютно съшия аналог с изключение на буфера. Той отново е само java.nio.Buffer базиран. Това засега е единственият начин за директна комуникация между двете среди (C и Java). Разбира се, и тук не всички функции са адаптирани за java-mpi, а само най-важните предоставящи основна функционалност. В Таблица 14 може да се видят повечето от тях.

**Таблица 14. Методи на Win класа.**

|  |  |
| --- | --- |
| **Java-mpi method** | **MPI function** |
| public void fence(int \_assert) | MPI\_WIN\_FANCE |
| public void get(Object base,int origin\_offset, int origin\_count, Datatype origin\_type, int target\_rank, long target\_disp, int target\_count, Datatype target\_type) | MPI\_GET |
| public void put(Object base, int origin\_offset, int origin\_count, Datatype origin\_type,int target\_rank, long target\_disp, int target\_count, Datatype target\_type) | MPI\_PUT |
| public void flush(int rank) | MPI\_WIN\_FLUSH |
| public void flush\_all() | MPI\_WIN\_FLUSH\_ALL |
| public void flush\_local(int rank) | MPI\_WIN\_FLUSH\_LOCAL |
| public void flush\_local\_all() | MPI\_WIN\_FLUSH\_LOCAL\_ALL |
| public Info info() | MPI\_WIN\_GET\_INFO |
| public String name() | MPI\_WIN\_GET\_NAME |
| public void name(String name) | MPI\_WIN\_SET\_NAME |
| public void lock(int lock\_type, int rank, int \_assert) | MPI\_WIN\_LOCK |
| public ByteBuffer shared\_query(int rank,ByteOrder order) | MPI\_WIN\_SHARED\_QUERY |

Създаването на Win обект може да стане по няколко начина, от които зависи и по нататъшната употреба на обекта. Всеки Win обект започва своя живот от „intra“ комуникатор. Методи се имплементират от Intracomm.java класа.

Първия начин е чрез public Win win\_create(ByteBuffer base, int unit\_dips,Info info), аналог на MPI\_WIN\_CREATE(). Особеността при него, е че потребителят предоставя паметта, която ще се ползва от системата. Операциите не са всички, но са достатъчно, за да определим дали семантиката на MPI за RMA комуникация е подходяща и за java-mpi.

Следващия метод е public Win win\_allocate(int size, int dist\_unit, Info info). При него е достатъчно да се подаде големина на буфера. Системата управлява това къде точно ще се отдели памет за него. Методът е аналог на MPI\_WIN\_ALLOCATE(). И последния е public Win win\_allocate\_shared(int size, int disp\_unit, Info info), аналог на MPI\_WIN\_ALLOCATE\_SHARED(). Това е метод, използван за заделяне на споделена памет между процеси. Това води до по-различен метод на достъп до паметта от предишните два метода.

Стандартната MPI синхронизация е налице и в java-mpi. Тоест използва се MPI\_WIN\_FENCE. В Пример 6.12 е показана подобна синхронизация.

**Пример 6.12**

WinObject.fence(0);

// operations e.c get, put ...

WinObject.fnece(0);

. . .

**Пример 6.13**

public static void main(String[] args) {

MPI.Init(args);

int wrank = MPI.COMM\_WORLD.rank();

int wsize = MPI.COMM\_WORLD.size();

Intеrcomm comm\_node = MPI.COMM\_WORLD.split\_type(MPI.COMM\_TYPE\_SHARED, 0, MPI.INFO\_NULL);

int sm\_rank = comm\_node.rank();

int sm\_size = comm\_node.size();

Info win\_info = new Info();

win\_info.create();

win\_info.set("alloc\_shared\_noncontig", "true");

Win sm\_win = comm\_node.win\_allocate\_shared(4, 4, win\_info);

win\_info.free();

sm\_win.lock\_all(0);

sm\_win.sync();

comm\_node.barrier();

ByteBuffer[] buf = new ByteBuffer[sm\_size];

if(sm\_rank!=0){

for (int i=0; i<sm\_size; i++) {

buf[i] = sm\_win.shared\_query(i, ByteOrder.BIG\_ENDIAN);

if(i==sm\_rank) {

// modification

buf[i].putInt(0, sm\_rank+100);

}

}

}

else{

// Master just watches it.

}

sm\_win.unlock\_all();

if(sm\_rank!=0){

for (int i=0; i<sm\_size; i++) {

if(buf[i]!=null)

// get the value

int v = buf[i].getInt(0);

int s = buf[i].capacity();

}

}

sm\_win.free(); comm\_node.free();

MPI.Finalize();

}

Пример 6.13 показва начин за използване на Win.java и методи определени за RMA операции. Примерът започва с реда

Intеrcomm comm\_node = MPI.COMM\_WORLD. split\_type(MPI.COMM\_TYPE\_SHARED,0, MPI.INFO\_NULL);

Това е ред, изпълняващ метод аналогичен на MPI\_COMM\_SPLIT\_TYPE(), въведен в MPI-3. Идеята на split\_type() е да се остави на системата да реши как да раздели входния комуникатор. Разделянето се базира на зададен тип. В нашия случай той е MPI.COMM\_TYPE\_SHARED, който разделя входния комуникатор на под-комункатори. Резултатните комуникатори може да извършват операции със споделена памет. Следващата стъпка е да създадем Win обекта и това става с тези редове:

Info win\_info = new Info();

win\_info.create();

win\_info.set("alloc\_shared\_noncontig", "true");

Win sm\_win = comm\_node.win\_allocate\_shared(4, 4, win\_info);

В тях може да се види и пример за ползване не Info обект.

Самите операции, в нашия случай shared\_query, се извършват между lock\_all и unlock\_all.

sm\_win.lock\_all(0);

// operations

sm\_win.unlock\_all();

## Входно/Изходни операции

Входно/изходните операции са последната реализирана функционалност, която ще разгледаме. Всички необходими методи се имплементират в File.java. В него се намират аналог на операцията MPI\_FILE\_WRITE:

public void write(Object buff,int offset,int count,Datatype datatype,Status stat,ByteBuffer objBuff)

както и аналог на операцията MPI\_FILE\_READ:

public void read(Object buff,int offset,int count,

Datatype datatype,Status stat,ByteBuffer objBuff)

Отварянето на файла става през public File file\_open(String fname,int amode,Info info), аналог на MPI\_FILE\_OPEN(). Предназначението, за което се отваря файла се специфицира от полета намиращи се в MPI.java.

..

/\*\* file flag modes \*/

public static final int MODE\_RDONLY = 2; /\* ADIO\_RDONLY \*/

public static final int MODE\_RDWR = 8; /\* ADIO\_RDWR \*/

public static final int MODE\_WRONLY = 4; /\* ADIO\_WRONLY \*/

public static final int MODE\_CREATE = 1; /\* ADIO\_CREATE \*/

public static final int MODE\_EXCL = 64; /\* ADIO\_EXCL \*/

/\* ADIO\_DELETE\_ON\_CLOSE \*/

public static final int MODE\_DELETE\_ON\_CLOSE = 16;

/\* ADIO\_UNIQUE\_OPEN \*/

public static final int MODE\_UNIQUE\_OPEN = 32;

public static final int MODE\_APPEND = 128; /\* ADIO\_APPEND \*/

...

Комбинация между предназначенията също е възможна с „|“. Например,

MODE\_WRONLY | MODE\_CREATE.

Това, което е добавено в MPI е възможността за предоставяне на изглед на файла. Т.е. всеки процес да вижда различни порции от файла. Тук са и основните проблеми за java-mpi. Това се получава, заради голямата разлика между C и Java. В Java няма как да предоставим абсолютно същото представяне на нови типове данни, както в MPI, които са и основната на MPI файловия изгледа. За сега предоставяме нещо, което е до известна степен подобно, но то е непълно и тромаво. Затова не може да кажем, че е аналогия на MPI функциите, изпълняващи същото. За момента интерфейсът дава известна информация за това дали е използваем в java и къде са основните проблеми.

Посоката към, която сме се ориентирали е използването на допълнителни класове, наследяващи Datatype.java, като те ще описват самият файлов изглед. Досега имаме само един Subarray.java, създаването му се осъществява със следните два метода

public Subarray(int ndims,int[] sizes, int[] sub\_sizes, int[] starts)

public void create(Datatype old\_type,int order).

Самата имплементация за създаването на изглед е описана в **Глава 7**.

**Пример 6.14**

public static void main(String[] args) {

MPI.Init(args);

int rank = MPI.COMM\_WORLD.rank();

int size = MPI.COMM\_WORLD.size();

File fh = MPI.COMM\_WORLD.file\_open("/home/nikolay/workspace/trash/test123",

MPI.MODE\_CREATE|MPI.MODE\_RDWR, MPI.INFO\_NULL);

int[] sizes = new int[1]; sizes[0] = size;

int[] subsizes = new int[1]; subsizes[0] = 1;

int[] starts = new int[1]; starts[0] = rank;

Subarray filetype = new Subarray(1, sizes, subsizes, starts);

filetype.create(MPI.BYTE, MPI.ORDER\_C);

filetype.commit();

fh.view(0, MPI.BYTE, filetype, "native", MPI.INFO\_NULL);

byte[] buff = new byte[10];

buff[0] = (byte)rank; buff[1] = (byte)(rank+0x1);

Status stat = new Status();

fh.write(buff, 0, 2, MPI.BYTE, stat, null);

System.out.println(stat.mpi\_error);

fh.write(buff, 0, 2, MPI.BYTE, null, null);

System.out.println(stat.mpi\_error);

filetype.free();

fh.close();

MPI.Finalize();

}

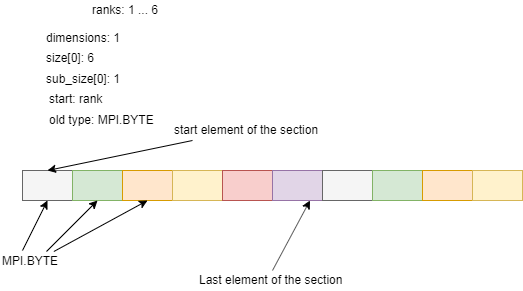
Пример 6.14 започва с отваряне на файл /home/nikolay/workspace/trash/test123 с флаг MPI.MODE\_CREATE|MPI.MODE\_RDWR (създаване на нов файл, и достъп за писане и четене). След което се преминава към описание и създаване на файловия изглед.

Subarray filetype = new Subarray(1, sizes, subsizes, starts);

filetype.create(MPI.BYTE, MPI.ORDER\_C);

filetype.commit();

fh.view(0, MPI.BYTE, filetype, "native", MPI.INFO\_NULL);

  
Фиг. 6.2

На фигура 6.2 може да се види резултатният изглед. Файлът се разделя на секции с големина равна на броя на процесите в COMM\_WORLD. Всеки от тези процеси вижда само един елемент от цялата секция, като той се намира с изместване равно на неговия ранг. След като секцията приключи тя започва отначало, като отново изместванията, големината и борят на елементите е същия. За сега аргументът отнасящ се към боря на измеренията не се подържа, той е предназначен за бъдещи разработки. Стойността му трябва винаги да е 1.

## Обобщение

В тази глава разгледахме, какво точно представлява java-mpi интерфейсът. Определихме в основи, от какви класове и методи ще се състои. Разбира се, не всички методи и функционалности са разгледани, но с тези, които описахме, ще е възможно да дадем отговор на въпроса, дали java е наистина подходящ език за MPI стандарта.

## Подобен софтуер

**Java-MPI** не е първият софтуер, който се опитват да имплементира MPI подобен интерфейс. Североизточенят център за паралелни архитектури, Университет Сиракуза издава публикация с подобен интерфейс, с име **mpiJava 1.2** [mpiJava 1.2]. Точно по него се създават и двата продукта имплементиращи го**, MPJ Express [MPJ Express]** и функционалността на **Open MPI да се използва с java**. В основата си тези две разработки също базират своята работа на JNI, т.е използват връзка между java и C/C++.

* mpiJava 1. 2

MPI стандарта търпи многократни преработки и множество промени, добавяне на нова функционалност и така нататък. Дори в момента се развива MPI версия 4.0. Докато mpiJava не е развиван от доста време насам, версията 1.2 съответства на MPI 2. Макар и стар, интерфейса е много по-надежден и сигурен е за употреба от java-mpi, поради много по-задълбоченото му тестване и използване.

* Open MPI с Java

Open MPI е библиотка имплементираща MPI стандарта [Open MPI]. Архитектурата и е много подобна на MPICH. За разлика от MPICH, Open MPI разполага с горепосочения интерфейс, mpiJava 1.2.

* MPJ Express

Това е проект имплементиращ mpiJava 1.2. Архитектурата му е доста подобна на нашата. За разлика от интерфейса на Open MPI, който изисква задължително своята собствена библиотека, MPJ Express не е задължена да използва точно определена библиотека. Тя може да бъде използвана както с Open MPI, така и с MPICH, което дава известна свобода.

Една от основните причина поради, която създаваме java-mpi e обновяване на този тип интерфейс. Както видяхте по-горе, той не е единствения софтуер използващ mpi през java. Но е единственият софтуер, който за основа използва mpich. Да, MPJ Express също има възможността да използва mpich, но не и по оптимален начин.

Java-MPI и mpiJava могат да използват някаква форма на потребителски типове, но те са корено различни. mpiJava използва MPI подобен метод на конструиране на нови типове, който от совя страна не е съвсем естествен за Java. Докато Java-MPI разработва метод, който е съвсем естествен за Java, а именно сериализация, но съвсем различен от MPI. Това дава своите плюсове и минуси при самата му употреба.

Няма да навлизаме в детайли относно разликите:

- Потребителски типове. Java-MPI разполага с сериализиращ метод на комуникация, докато mpiJava разполага с MPI подобен метод.

- Оползотворяването на java.nio.Buffer е на много по-високо ниво в java-mpi.

- Java-MPI разполага с едностранна комуникаци, макар и не оптимална.

- Java-MPI разполага с по-добро управление на средата.

- Java-MPI разполага със “spawn” функционалност, **точка 6.10**

- Java-MPI разполага с паралални входно/изходни операции

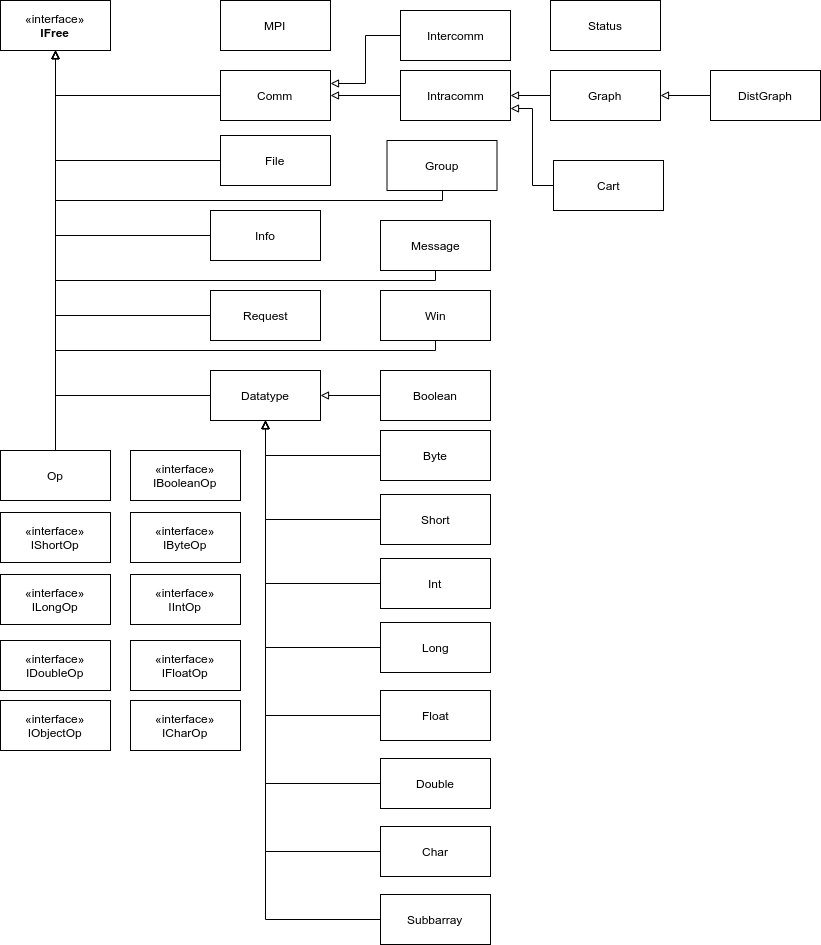
- mpiJava е по-стабилния интерфейс от двата.

В крайна сметка, причините са обновяване на интерфейса и имплементация на нови функционалностти.

Друга причина поради, която създаваме java-mpi е възможността му да се използва в android системи, но тази разработка изисква повече от само една дипломна работа. Основния проблем, който трябва да се реши преди това да е възможно е оптималното интегриране на mpich в android, което повреме на моето изследване завърши с неуспех.

# Имплементация

В предишната глава разгледахме в основи това, какво трябва да представлява java-mpi или поне какво целим да представлява за момента. Време е да опишем и нейната имплементацията. Един от основните моменти, който трябва да подчертаем отново е, че това не е имплементация започваща от нулата. Както казахме в уводната глава, за основа ще използваме mpich. Това ни дава бързина в имплементирането на основните функционалности, бързина на изпълнение, добра основа и отлична стабилност.

  
Фиг 7.1

## Обекти

В предишната глава вече разгледахме съответствието между Java обекти и MPI обекти. Имплементациятя също подържа такова съотвествие. Т.е. на всеки MPI обект съответства java-mpi обект. Интерфейса го постига чрез идентификатори, те може да са само числа, какъвто е MPI\_Comm или адреси, какъвто е MPI\_File, но във всички случай е от тип long за Java. Той играе ролята на референция и абсолютно всеки MPI обект, съответстващ на Java обект, разполага с такъв идентификатор. С други думи, java-mpi не имплементира комуникатор, той просто го използва наготово от mpich.

Примери 7.1 и 7.2 показват точно това. В тях може да се видят въпросните идентификатори под името „handler”. Повечето обекти в java-mpi не могат да се създават от само себе си, в смисъл, няма как да се създадат с просто извикване на конструктори. Има два варианта – или се създават от вече създаден обект, или чрез процес на регистриране в системата (commit метод или нещо подобно). Подобен е случаят с File.java, Пример 7.2. Единственият метод за създаване на MPI файл през java-mpi е public File file\_open(…), който се намира в друг обект, а именно Comm.java.

**Пример 7.1**

...

public class Comm {

protected long handler;

// null handler

public static final long null\_handler = 0x04000000;

public Comm(long handler) {

this.handler = handler;

}

...

**Пример 7.2**

...

public class File {

private long handler;

public File(long handler){

this.handler = handler;

}

// MPI\_File\_get\_amode

public int amode(){

return \_amode(this.handler);

}

private static native int \_amode(long fh);

// MPI\_File\_set\_atomicity

public void atomicity(boolean flag){

\_atomicity(this.handler, flag);

}

В много от случаите, идентификаторът представлява число. Но се срещат случаи, в който това не е точно така. За да може да ползваме MPI\_File, идентификаторът му в mpi e цяла структура. Когато се налага, за идентификатор се взема адресът на структурата, която се използва от C. MPI обектът се запазва в динамична или статична памет, след което се инициализира и, вместо да се изпраща цялата структура, се изпраща само нейният адрес до Java ниво. За обратно използване се следва точно обратния метод, използва се адреса, за да се достъпи съответната структура. Това дава многократно по-бързо изпълнение. Прехвърлянето на обекти е трудоемка и бавна операция, затова се опитваме да я избягваме колкото е възможно повече.

В Пример 7.3 е показна интерфейсната функция за връзка с file\_open метода. В нея може да се види заделянето на паметта за MPI\_File структурата, инициализирането ѝ и нейното връщане като адрес.

**Пример 7.3**

/\*

\* Class: mpi\_Intracomm

\* Method: \_file\_open

\* Signature: (JLjava/lang/String;IJ)J

\*/

JNIEXPORT jlong JNICALL

Java\_mpi\_Intracomm\_\_1file\_1open(

JNIEnv \*env, jclass class, jlong comm, jstring fname,

jint amode, jlong info)

{

int result;

const char \*name;

MPI\_File \*file = malloc(sizeof(MPI\_File));

if(file == NULL){

jniOutOfMemoryException(env,"malloc");

return -1;

}

name = (\*env)->GetStringUTFChars(env,fname,NULL);

if(name != NULL){

result = MPI\_File\_open((MPI\_Comm)comm,name,amode, (MPI\_Info)info,file);

if(result != MPI\_SUCCESS){

mpiFileException(env,"MPI\_File\_open");

}

}

(\*env)->ReleaseStringUTFChars(env,fname,name);

return (jlong)file;

}

Единственият недостатък на този подход е необходимостта от изчистващ механизъм. Начинът, който сме предложили е чрез метода free. Всеки обект имплементира IFree.java интерфейс, който задължава да се имплементира public void free();

package mpi;

public interface IFree {

public void free();

}

Точно free методът е отговорен за изтриване на всички ресурси, ползвани от съответния му обект.

В спецификацията не го споменахме ясно, но java-mpi имплементира повечето, ако не всички, най-важни предефинирани обекти на MPI. Става дума за обекти като MPI\_COMM\_WORLD, MPI\_BYTE,MPI\_INT, MPI\_MAX и прочие. Инициализацията им става по време на зареждане на java-mpi библиотеката от страна на java. Всеки един от тези обекти може да се намери в MPI.java.

## Mpich промени

Имплементацията java-mpi не е чиста версия на MPI имплементация. В нашия случай имаме MPICH, към която сме добавили няколко промени. Те са свързани с това, MPICH да може да изпълнява Java функции, тъй като това не е директно възможно от JNI. Функционалността, към която спада това, е изпълнение на потребителски операции, „reduce-функционалността”. В главата за JNI споменахме за това, как C кодът може да направи обратна връзка с Java, но не дадохме пример. Тук е мястото да го направим.

**Пример 7.4**

/\*

\* Class: mpi\_op\_Op

\* Method: \_create

\* Signature: (Ljava/lang/Object;ZJ)J

\*/

JNIEXPORT jlong JNICALL

Java\_mpi\_op\_Op\_\_1create(

JNIEnv \*env, jclass class, jobject obj, jboolean isCommute,jlong type)

{

tJniOp\* op;

long long unsigned int op\_addres = 0;

// allocate a operation handler

op = malloc(sizeof(tJniOp));

if(op == NULL){

jniOutOfMemoryException(env,"malloc");

return -1;

}

// crete object callback

op->obj = (\*env)->NewGlobalRef(env,obj);

if(op->obj == NULL){

return -1;

}

op\_addres = (long long unsigned int)op;

MPII\_Op\_create\_java(jniOpFunc,isCommute,&op->mpi\_op,op\_addres);

return (jlong)op;

}

Първото, с което започваме е създаването на операция. Това става чрез функцията Java\_mpi\_op\_Op\_\_1create. Това съответства на \_create метода, намиращ се в mpi.op.Op класа. Създаването на операцията започва със заделяне и инициализация на tJniOp обект, който е структура от вида:

typedef struct {

MPI\_Op mpi\_op; // mpi operation

jobject obj;

}tJniOp;

След това, алгоритъмът продължава с взимане на дефинираната операция и конвертирането ѝ до глобална референция. От страна на C, операцията се вижда чрез java обект, jobject obj, докато от страна на java тя е под формата на интерфейс.

// crete object callback

op->obj = (\*env)->NewGlobalRef(env,obj);

Превръщането на операция от локална в глобална се прави поради факта, че JNI не запазва никакви локални променливи и данни след излизане от функция. За да може да се използва като глобална и да се кешира е необходимо да се създаде глобална инстанция. Обектът (op->obj) съдържа и самата операция, която ще се изпълни. Т.е. обектът имплементиращ интерфейс от типа IByteOp.java, IIntOp.java и т.н.

След като се създаде глобална инстанция, системата преминава към същинското инициализиране. Това е и частта, която не е официално към mpich. Функцията, която създава операцията е променена версия на

int MPIR\_Op\_create\_impl(MPI\_User\_function \* user\_fn, int commute, MPI\_Op \* op),

която се използва от mpich за създаване на обикновена C операция. В нашата система тя има вида от Пример 7.5.

**Пример 7.5**

int MPII\_Op\_create\_java(MPII\_User\_function\_java \* java\_fn, int commute, MPI\_Op \* op,

long long unsigned int type\_pointer)

{

MPIR\_Op \*op\_ptr;

int mpi\_errno = MPI\_SUCCESS;

op\_ptr = (MPIR\_Op \*) MPIR\_Handle\_obj\_alloc(&MPIR\_Op\_mem);

/\* --BEGIN ERROR HANDLING-- \*/

if (!op\_ptr) {

mpi\_errno =

MPIR\_Err\_create\_code(MPI\_SUCCESS, MPIR\_ERR\_RECOVERABLE,

"MPII\_Op\_create\_java", \_\_LINE\_\_, MPI\_ERR\_OTHER,

"\*\*nomem", "\*\*nomem %s", "MPI\_Op");

goto fn\_fail;

}

/\* --END ERROR HANDLING-- \*/

op\_ptr->language = MPIR\_LANG\_\_JAVA;

op\_ptr->kind = commute ? MPIR\_OP\_KIND\_\_USER : MPIR\_OP\_KIND\_\_USER\_NONCOMMUTE;

op\_ptr->function.java\_function = (void (\*)

(const void\*, void \*,const int\*, const MPI\_Datatype\*,long long unsigned int)) java\_fn;

op\_ptr->java\_pointer = type\_pointer;

MPIR\_Object\_set\_ref(op\_ptr, 1);

MPIR\_OBJ\_PUBLISH\_HANDLE(\*op, op\_ptr->handle);

#ifdef MPID\_Op\_commit\_hook

MPID\_Op\_commit\_hook(op\_ptr);

#endif

fn\_exit:

return mpi\_errno;

fn\_fail:

goto fn\_exit;

}

Аргументите на тази функция са:

* java\_fn: Указател към функция. Обратна връзка за java-mpi.
* commute: true ако операцията е комутативна или flase ако не е.
* op: MPI указател „handler“ на операцията.
* type\_pointer: указател към нашия тип, съдържащ операцията (обекта).

Основната задача на тази функция е да инициализира MPIR\_Op структура и да я подаде за публикуване.

Може да се забележи, че използваме функция, а не извикваме директно операцията. Операцията се извиква през функцията поради това, че не искаме mpich директно да зависи от JNI. За да може C да извиква java метод, той се нуждае от JavaVM указател или поне от JNIEnv указател, който може да се вземе от JavaVM. Друго, от което се нуждаем, за да може да направим извикването е идентификатор на метода, който ще изпълняваме. Той се взима по-подобен начин показан в Пример 7.6.

**Пример 7.6**

gJniObjectOpCallback = (\*env)->GetMethodID(env,cls,"\_operation",

"(Ljava/nio/ByteBuffer;Ljava/nio/ByteBuffer;)V");

Веднъж публикувана, достъп до операцията може да се получи чрез MPI\_Op променливата, която подадохме като аргумент. Няма да даваме детайли за самото изпълнение на MPI\_Reduce, но ще покажем кода, в който се извиква операцията:

#ifdef HAVE\_JAVA\_BINDING

if(is\_java\_uop){

op\_ptr->function.

java\_function(inbuf,inoutbuf,&count,&datatype,op\_ptr->java\_pointer);

} else

#endif

{...}

Същинското извикване става чрез:

void jniOpFunc (const void\* inbuff, void \* inoutbuff,

const int\* len, const MPI\_Datatype\* datatype,

long long unsigned int op);

Тя извършва цялата работа по определяне на типове, операция, предоставяне на буфери и други. В крайна сметка се изпълнява определената операция благодарения на функция от типа **<type>**\_op\_call. В Пример 7.7 е показана фунцкцията object\_op\_call.

**Пример 7.7**

void object\_op\_call(JNIEnv\*env,jobject obj,const jbyte\* in,jbyte\* inout,int size)

{

jobject in\_buff = (\*env)->NewDirectByteBuffer(env,(void\*)in,size);

jobject inout\_buff = (\*env)->NewDirectByteBuffer(env,inout,size);

// call the method

(\*env)->CallObjectMethod(env,obj,gJniObjectOpCallback,

in\_buff,inout\_buff);

(\*env)->DeleteLocalRef(env,in\_buff);

(\*env)->DeleteLocalRef(env,inout\_buff);

}

Това е една от най-сложните операции, тъй като изисква създаване на буфери и отговаря за операции свързани с обекти. След като се създадат въпросните буфери, те се използват като аргументи за операцията.

**Пример 7.8**

/\*\*

\* Object Operation

\*/

public abstract class ObjectOp {

private Datatype type;

public ObjectOp(Datatype type){

this.type = type;

}

public abstract Object operation(Object a,Object b);

private void \_operation(ByteBuffer a\_buff,ByteBuffer b\_buff){

Object a = this.type.constructFromByteByffer(a\_buff);

Object b = this.type.constructFromByteByffer(b\_buff);

Object c = this.operation(a, b);

b\_buff.position(0);

this.type.setToByteBuffer(c, b\_buff);

}

}

В пример 7.8 е показан абстрактният клас за обектни оперции. В него се изпълнява алгоритъм за реконструкция на двата обекта и тяхното използване в public abstract Object operation(Object a,Object b);. В резултат на операцията се създава нов обект, който се конструира като ByteBuffer обект и се подава на b\_buff за обновяване. Подобна е и процедурата с по-просите типове – int, long и т.н. Основното предимство пред операциите с обекти е, че те не изискват допълнителна логика за конструкция и реконструкция. Това става при пренасяне от C в Java. И затова, те са само Java интерфейси.

Проблемът, който описахме по-горе, а именно, че mpich не може да извика директно JNI функции е причината, поради която за сега не имплементираме хващането на грешки.

Това е един от основните недостатъци на java-mpi, когато стане дума за портативност. Ако искаме да имаме функционалности от рода на хващане на грешки или “reduce”, java-mpi и имплементацията върху, която се пише трябва да са свързани. Разбира се, това не е важно за абсолютно всички функции на MPI. Прости P2P или колективни функции, та дори и RMA операции, не изискват подобна зависимост. Това означава, че потребителят може да конфигурира java-mpi интерфейса така, че да може да използва повече от една имплементация. Но това може да стане само на цената на премахнати функционалности.

## Типове

В Таблица 4 вече показахме всички типове поддържани от java-mpi, както и всички нейни MPI съответствия. Виижда се, че самото прехвърляне на данните от Java и C не е лесна задача. Цялата логика по управление на типовете става чрез структурата JNI\_Datatype\_struct. В Пример 7.9 може да се види цялостната ѝ декларация.

Самата структура има множество предназначения и функции. Главната от които е, да определя какъв тип от данни е спрямо MPI и Java. Тя съдържа, също така, и методи за взимане на референция към данните или тяхното копиране във временен буфер.

**Пример 7.9**

typedef struct JNI\_Datatype\_struct {

// true if the type is Buffer e.c ByteBuffer,IntBuffer..

jboolean isBufferType;

MPI\_Datatype mpi\_type;

fBufferControl getBuffAddr;

fBufferControl relBuffAddr;

int size;

// true blocking functions

fBufferControl copyBuffer;

fBufferControl setBuffer;

jboolean isCustom;

} tJniType;

Аргументи:

* IsBufferType: Индикатор на това дали типа е от вид „java.nio.Buffer“, каквито са всички потребителски типове.
* mpi\_type: Идентификатор на съответния MPI тип.
* getBufferAddr/relBufferAddr: Това са указатели към функции използвани за достъп до данните. За основните типове има две възможности да се вземат като референция или да се копират във временен буфер.
* size: Големина на типа, в байтове.
* copyBuffer/setBuffer: Това са указатели към функции извършващи копиране на данните, при тях не се използва рефериране на данните.

На java ниво няма голяма логика, управляваща типовете данни. Единствената такава се отнася главно за потребителските типове. Това което се използва за достъп до tJniType структурата е нейният адрес. Т.е. за индикатор на типа, който и да е, се използва за адреса на управляващата му структура. Всеки тип разполага с такава структура, било то инициализирана в началото или по време на изпълнението. Тя извършва същинската работа по достъп на данните.

## Потребителски типове

Следващия интересен аспект на java-mpi е използването на обекти. Java разполага с начин за сериализация на обекти, a именно Java Serialization. Това е метод имплементиран от Java, за преобразуване на java обекти в масив от данни и обратно. Един от проблемите, заради който не можем да го ползваме, е невъзможността да определим големината на обекта. Друга причина, поради която избягваме този метод, е свободата. Проблемът на този метод е, че той преобразува цял обект с всички негови данни, което го прави по-голям от необходимото. Методът, който предлагаме два много повече свобода на избора на данните и прави обема на данните за комуникация далеч по-малък. Той е базирн на „java reflection“. Всичко, което е необходимо, за да работи механизмът, е да се имплементират „getter/setter“ методи на всички полета, които ще участват в комуникацията. Освен това, класът трябва да разполага с конструктор без параметри. След което е необходимо само да се следва механизма за създаване на потребителски тип, показан в предишната глава.

Вече показахме процеса на създаване на потребителски тип, но какво наистина се случва под повърхността. Основният и най-важен метод е \_commit от класа mpi.datatype.Datatype. Той извършва същинското създаване на типа.

**Пример 7.10**

JNIEXPORT jlong JNICALL Java\_mpi\_datatype\_Datatype\_\_1commit\_\_I(

JNIEnv \*env,jclass class, jint size)

{

int result;

tJniType \*custom\_type;

jlong handler;

MPI\_Datatype new\_type = 0;

// allocate the type

custom\_type = malloc(sizeof(tJniType));

if(custom\_type == NULL){

jniOutOfMemoryException(env,"malloc");

return -1;

}

// configure the type

result = MPI\_Type\_contiguous(size,MPI\_BYTE,&new\_type);

if(result != MPI\_SUCCESS){

mpiErr(env,result,"MPI\_Type\_contiguous");

free(custom\_type);

return -1;

}

// create the type

result = MPI\_Type\_commit(&new\_type);

if(result != MPI\_SUCCESS){

mpiErr(env,result,"MPI\_Type\_commit");

free(custom\_type);

return -1;

}

// init the custom type

custom\_type->mpi\_type = new\_type;

custom\_type->isBufferType = JNI\_TRUE;

custom\_type->getBuffAddr = getAddrDirectBuffer;

custom\_type->isCustom = JNI\_TRUE;

// the custom type uses direct buffer so it doesn't need

// release function

custom\_type->relBuffAddr = NULL;

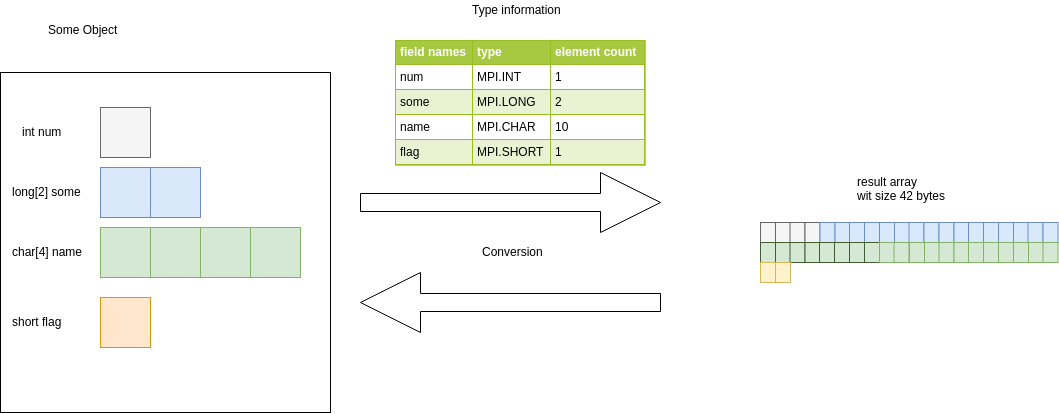
custom\_type->size = size;

return (jlong)custom\_type;

}

Пример 7.10 показва функцията, съответстваща на mpi.datatype.Datatype.\_commit. Тя извършва всички работи по инициализацията на новия тип. Т.е. определяне големината на тип, регистрация и конфигурация на новия тип и не на последно място, регистрирането на функции, който ще се използват за достъп до данните.

Едно от най-интересните неща тук е начинът, по който java-mpi предоставя данните на mpich. Както може да се види от Пример 7.10, при създаване на нов тип от страна на jni-mpi се създава и нов тип от срана на MPI библиотеката. Това ни помага да държим данните като едно цяло, да не ги разделяме дори и на ниско ниво. Новият MPI тип е произ­воден на MPI\_BYTE, с определена големина предоставена от потребителя. Той е точно съответствие на новия тип създаден от потребителя и се използва само с ново-регистрирания клас.

  
Фиг. 7.2

На фиг 7.2 може да се види и начинът, по който се преобразуват данните от java обект до mpi тип (проста последователност от байтове). Идеята е проста, преди да се започне каквото й да е било преобразуване, потребителят трябва да регистрира новия тип. Това става чрез подаване на всички необходими данни описани в предишната глава. След като това е направено, системата следва последователността от имена (аргументи), предоставени при създаването на типа. Това е илюстрирано в таблицата на фиг. 7.2. Всяко колона представлява един от трите масива, които трябва да се подадат на системата, за да се опише дадения тип. След това java-mpi взима всеки от параметрите, преобразува го до байтове и го съединява с останалите в масив. Големината на масива е равна на големината на типа. Така обратната процедура по преобразуване на данните от масив в обект е също толкова проста – jni-mpi отново следва подадените му данни от потребителя за определения тип. Използвайки тях, той спокойно може да следва обратния процес. Цялото взимане и вкарване на данни в java обект става точно чрез “java reflection“.

Използването на новия тип се осъществява като при всеки предефиниран, с една малка разлика. За него е необходим java.nio.Buffer. Той играе ролята на преводач от Java към C и обратното. Тъй като е много по-скъпо да се използва обикновен масив за преноса на данни, java-mpi използва подобен механизъм за пренос. Като може би единствената оптимизация, която може да се направи на този етап, е по-оптимално използване на тези буфери, тъй като създаването им е сравнително скъп процес.

Самото преобразуване се извършва чрез следните два метода:

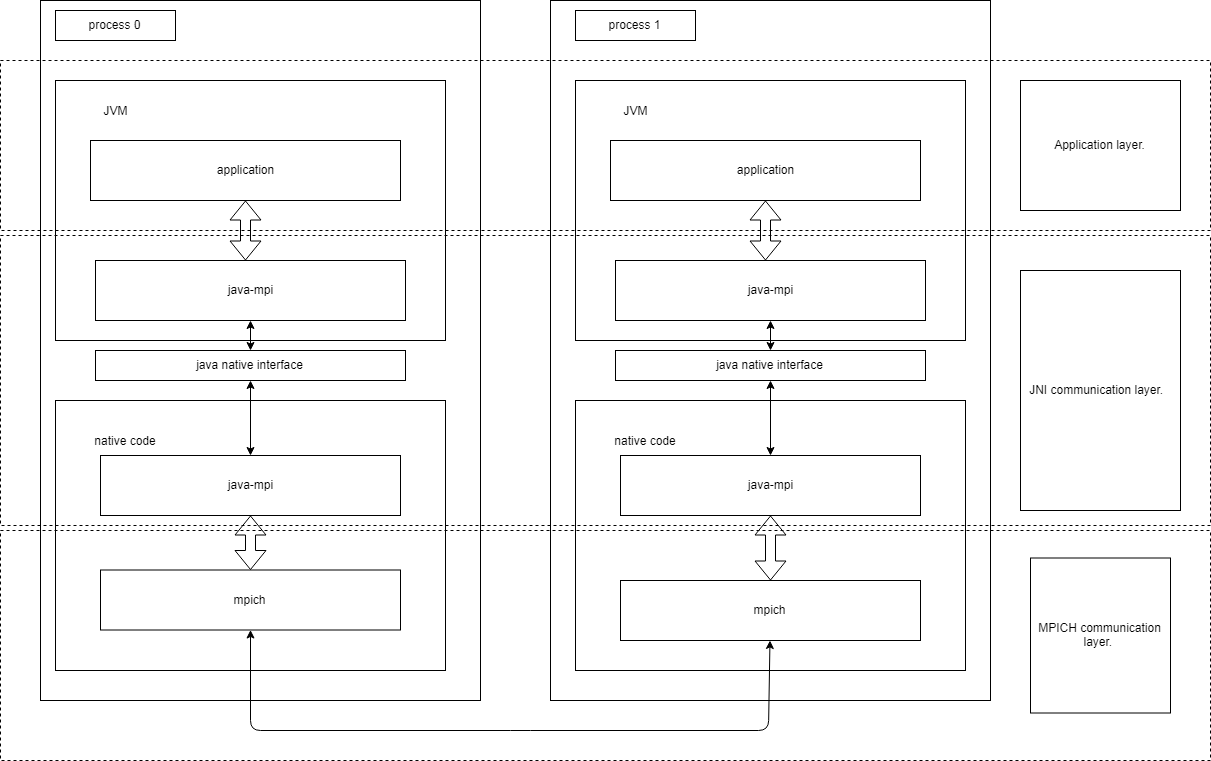
public void setToByteBuffer(Object obj,ByteBuffer buff)

и

public Object constructFromByteByffer(ByteBuffer buff)

като преобразуването е рекурсивно, както споменахме.

## Комуникации



Фиг. 7.3

Всяка комуникация между процеси преминава през няколко нива, докато стигне от едното до другото адресно пространство. MPICH изпълнява същинската комуникация, но данните, които той трябва да обработи не са директно достъпни за него. Те се намират в приложния слой на системата. Те се вземат от java-mpi (частта, която се намира в Java) и се предават на “native” частта чрез JNI. Оттам вече, данните се поемат от java-mpi (частта, намираща се в C) и се предават на MPICH. От своя страна MPICH извършва самата комуникация между процесите. Основното ниво, към което ние се ориентираме е JNI комуникационното ниво, MPICH комуникационното ниво използваме в относително оригинален вид. На фиг 7.3 може да се види цялостната архитектура на проекта по слоеве.

### Методи за достъп на данни от Java към C и обратно

Има три основни механизма, по който данни преминават от Java към C – копиране, референция и използване на “nio” буфер. Всеки от тях има своите предимства, които ги правят удобни за ползване в определени случай, както и минуси.

* Копиране

Идеята му е да прехвърля данни от Java към C чрез копиране, тоест използва се временен буфер от C страната. Този подход може да е един от най-бавните, но пък е най-сигурен. При него не се налага да се спира работата на системата за почистване на памет, нито пък копираните данни да са в опасна секция. Процедурата, която се следва, при прехвърляне на данни от единия край към другия, е да се създава втори буфер. Този буфер се намира в “native” частта на системата, в него се копират данните и се дават за ползване от mpich. Точно това е временният буфер, който споменахме. Ако е необходимо, буферът, намиращи се в Java, се обновява и обратно.

**Пример 7.11**

// Double

void getDoubleRegion(

JNIEnv \*env,jobject obj,jint start,jint len,void\*\* buff,int type\_size,int mode)

{

(\*env)->GetDoubleArrayRegion(env,obj,start,len,\*buff);

}

void setDoubleRegion(

JNIEnv \*env,jobject obj,jint start,jint len,void\*\* buff,int type\_size,int mode)

{

(\*env)->SetDoubleArrayRegion(env,obj,start,len,\*buff);

}

В Пример 7.11 са показани функциите отговарящи за копирането на double масиви.

* Референция

Това е механизъм, базиран на две функции предоставени от JNI интерфейса – GetPrimitiveArrayCritical и ReleasePrimitiveArrayCritical. Те предоставят референции към примитивни масиви, намиращи се в Java. Това ги прави доста опасни, защото за да се получи достъп до тях трябва да се спре системата за почистване на памет. Но предимство при тях е, че не се налага копиране на данни.

За да е безопасно използването им се препоръчват няколко правила, който трябва да се следват от разработчиците:

* Не може да се извикват JNI функции между изпълнението на GetPrimitiveArrayCritical и ReleasePrimitiveArrayCritical.
* GetPrimitiveArrayCritical винаги е следвана от ReleasePrimitiveArrayCritical, двете функции обособяват началото и края на критична секция.
* Не трябва да се изпълнява код, който има опасността да блокира или чака дълго време.

**Пример 7.12**

void getCriticalSection(

JNIEnv \*env,jobject obj,jint start,jint len,void\*\* buff,int type\_size,int mode)

{

int capacity;

capacity = (\*env)->GetArrayLength(env,obj);

\*buff = (\*env)->GetPrimitiveArrayCritical(env,obj,NULL);

if(\*buff == NULL){

jniOutOfMemoryException(env,"JNI Exception");

}

if((len+start) > capacity){

(\*env)->ReleasePrimitiveArrayCritical(env,obj,

\*buff,JNI\_ABORT);

jniIndexOutOfBoundsException(env,

"Error Offset/Capacity!");

buff = NULL;

return;

}

\*buff = ((jbyte\*)\*buff)+(type\_size\*start);

}

void relCriticalSection(

JNIEnv \*env,jobject obj,jint start,jint len,void\*\* buff,int type\_size,int mode)

{

\*buff = ((jbyte\*)\*buff)-(type\_size\*start);

(\*env)->ReleasePrimitiveArrayCritical(env,obj,\*buff,mode);

}

Поради голямата им опасност, тези две функции се използват само в някои случаи.

* При блокиращи изпращании.
* При взимане на масиви, които не отговарят за комуникация и не се нуждаят от задължително обновяване.

В случай на не блокиращи комуникации или комуникация с обекти се ползват останалите два случая. Но поради опасността, която дават самите функции, проектът дава право на избор. Всички функции и методи използващи референция могат да преминат автоматично на копиране. Избора става по време на компилация на java-mpi.

* “NIO” буфер

Java.nio.Buffer (java.nio.ByteBuffer, java.nio.IntBuffer …)

Това е метод, който ползваме при имплементации изискващи асинхронен достъп до памет. За пример може да дадем „Persistent Communication Requests“ комуникацията на MPI – send\_init, recv\_init и т.н. При тези методи, данните, които се изпращат или получават, се променят динамично от потребителя. Следователно няма да е практично да се ползва копиране или референция. Други случаи, при който се използва този метод за прехвърляне на данни е използването на буферни типове данни (BYTE\_BUFFER, INT\_BUFFER,...) или комуникация с обекти.

За Java достъпът е ясен – чрез наследниците на java.nio.Buffer (java.nio.ByteBuffer, java.nio.IntBuffer). Докато от гледна точка на C достъпът се получава чрез GetDirectBufferCapacity и GetDirectBufferAddress.

**Пример 7.13**

// direct address buffer region

void getAddrDirectBuffer(

JNIEnv \*env,jobject obj,jint start,jint len,void\*\* buff,int type\_size,int mode)

{

int capacity;

capacity = (\*env)->GetDirectBufferCapacity(env,obj);

\*buff = (\*env)->GetDirectBufferAddress(env,obj);

if(\*buff == NULL){

jniUndefinedMemException(env,

"JNI GetDirectBufferAddress");

return;

}

if(((len+start)\*type\_size) > capacity){

jniIndexOutOfBoundsException(env,

"Error Offset/Capacity!");

buff = NULL;

return;

}

\*buff = ((jbyte\*)\*buff)+(type\_size\*start);

}

### Блокиращи методи за комуникация между два процеса

Блокиращите методи са особено лесни за имплементиране. Те могат да работят и с трите типа на достъп, а именно референция, копиране и “nio” буфер. При използването на референция приемаме, че MPI функциите, които използваме за комуникацията не могат да блокират и по време на критичната секция не изпълняваме никакви JNI функции.

### Неблокиращи методи

Неблокиращите операции работят на база на копиране или чрез java.nio.Buffer. Естествено е, че при втория случай пренасянето на данни става в пъти по-лесно и бързо. Докато първият случай предоставя възможност за комуникация на примитивни масиви. Принципът е – ако е изпращащ метод, той да се копира и запазва в „native“ частта на java-mpi. В случаите, когато е получаващ, тогава се създава буфер, като големината му е равана на това, което потребителят очаква да получи. След което, получаващият буфер се обновява.

Цялата тази логика се осъществява със структура, която държи всички буфери, измествания и други данни помагащи на системата да си върши работата, Пример 7.14.

**Пример 7.14**

// jni mpi request handler

typedef struct {

MPI\_Request request;

tJniType\* type[JNI\_IMAX\_BUFFERS];

// region variables

int offset[JNI\_IMAX\_BUFFERS];

int count[JNI\_IMAX\_BUFFERS];

// buffers

jobject buffers[JNI\_IMAX\_BUFFERS];

void\* address[JNI\_IMAX\_BUFFERS];

jboolean isPersistentRequest;

jboolean isSendRequest;

} tJniRequest;

Буферите са повече отколкото е необходимо, поради някои колективни комуникации, които обикновено използват повече от един буфер. Принципът е сравнително прост и ефективен. При инициализирането на неблокираща операция, винаги се създава tJniRequest променлива, която в Java структурата съответства на Java Request обект. Комуникацията между тях се осъществява чрез адреси. След като променливата е създадена се преминава към инициализация и създаване на необходимите буфери. Създадените буфери не са обикновени, tJniReqeust разполага с глобални променливи

jobject buffers[JNI\_IMAX\_BUFFERS];,

определени за обновяване на данни. Те дават достъпа до масивите намиращи се в Java.

## „persistent“ комуникация

Вече споменахме този вид комуникация в раздел 6.2. Тя включва в себе си методи като send\_init и recv\_init. Те работят само с един тип данни – буферни типове BYTE\_BUFFER, SHORT\_BUFFER, .... От където следва и, че имплементацията им не е особено сложна. В Пример 7.15 може да се види имплементацията на send\_init метода.

**Пример 7.15**

JNIEXPORT jobject JNICALL

Java\_mpi\_Comm\_\_1send\_1init(

....

if(!type->isBufferType){

mpiErrType(env,"Unsupported datatype!");

return NULL;

}

request = calloc(1,sizeof(tJniRequest));

if(request == NULL){

jniOutOfMemoryException(env,"calloc");

return NULL;

}

//create the request

// set datatype for the request

request->type[JNI\_ISEND\_BUFFER] = type;

request->isPersistentRequest = JNI\_TRUE;

type->getBuffAddr(env,buff,offset,count,

&request->address[JNI\_ISEND\_BUFFER],type->size,0);

result=MPI\_Send\_init(request->address[JNI\_ISEND\_BUFFER],count,

request->type[JNI\_ISEND\_BUFFER]->mpi\_type,dest,tag, (MPI\_Comm)comm,&request->request);

if(result != MPI\_SUCCESS){

mpiErr(env,result,"MPI\_Send\_init");

jniMpiRequestFree(env,request);

return NULL;

}

return (\*env)->NewObject(env,gJniRequestClass,

gJniRequestConstructor,(jlong)request);

}

Задачите, които методът извършва са две. Първата е да създаде Request обект, с който да се управляват по-нататъшните операции. Другата е да изпълни MPI\_Send\_initфункцията.

**Пример 7.16**

// MPI\_Send\_init

// Note: The function accepts only Direct Buffers!

public Request send\_init(Object buff,int offset,int count, Datatype type,int dest,int tag){

sendRecvInit(buff);

return \_send\_init(buff,offset,count,type.getHandler(),

dest, tag, this.handler);

}

private static native Request \_send\_init(

Object buff,int offset,int count,long datatype,int dest,int tag,long comm);

След като се инициализира операция за изпращане или получаване, промяна или достъп до данните става директно от буфера, инициализиран в началото, т.е. Object buff. Изпълнението на операцията стават директно чрез методи намиращи се в Request обекта - start, Wait и free.

## Обобщение

Основната идея на java-mpi е да даде достъп на Java до C функциите на MPICH. Интерфейсът се опитва да преодолее бариерата между двата езика. Пълно описание на всяка функция и метод няма как да бъде включено в този текс, а и не е необходимо. Операциите с файлове, като read и write са много подобни на send и recv. За това в тази глава дадохме по-важните методи и моменти на интерфейса. В кода може да се намерят примери за абсолютно всеки един метод, неговото ползване и имплементация.

# Тестове

## Увод

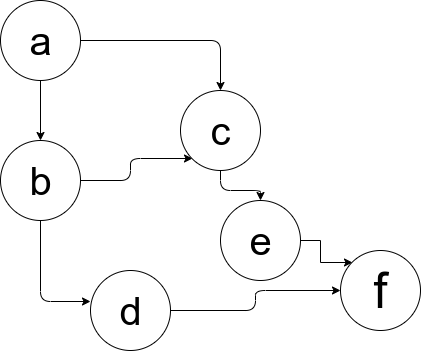
Тестовете, който изпълняваме целят не да ни покажат изправността на системата, а да ни покажат, че тя е използваема и има смисъл да се продължава с нейната разработка. Те се разделят на два сета. В първият са т.н. димни тестове („smoke tests“). Докато вторият тип тестове, които изпълняваме, са за използваемост и производителност.

## Smoke Tests

Димни тестове или тестове за проверка на производството са набор от сравнително прости тестове, целящи да покажат изправността на всички важни функционалности от системата. Резултата от тях показват дали системата е стабилна и дали има смисъл да се преминава към по-обстойно тестване.

Тестовете, който се изпълняват са базирани на всяка основна функционалност. Тоест всеки тест тества два или повече метода. Всеки от тях започва с MPI.Init и завършва с MPI.Finlize, кото между тях се намира същинският алгоритъм. В повечето случай той е прост, целящ да тества една или повече функционалности. Пълен списък с тестове може да се види в приложените към работата файлове.

## Максимален път в Даг

  
фиг. 8.1

Това е тест целящ да покаже потенциала на java-mpi. Той е базиран на алгоритъм за намиране на максимален път в даг (ориентиран ацикличен граф). На фиг. 8.1 може да видите пример на подобен граф.

Алгоритъмът, който използваме е стандартен. Той се базиран на динамично програмиране. Нека V = {a,b,c,d,..} са върховете на дага и Е = {1,2,3,4,..} ребрата му. За да намерим най-дългият път P, разделяме основната задача на под задачи, тоест първо търсим най-дългият път започващ във връх Vi и след това ги сравняваме за да намерим най-дългият между тях P(V) = {a,b,c,d,..}. Това ни позволява лесно да разделим задачата на отделните процеси. Това което правим е – всеки процес получава под-сет от върхове Vj={V(j\*avrg)+i}, където j е ранга, avrg е средно колко на брой върха се падат на процес и i е порденият връх. Като разбира се има случай, в които avrg не е точно, тоест броя на върховете не се дели точно на броя на процесите. В такива случай последния процес взима останалите върхове.

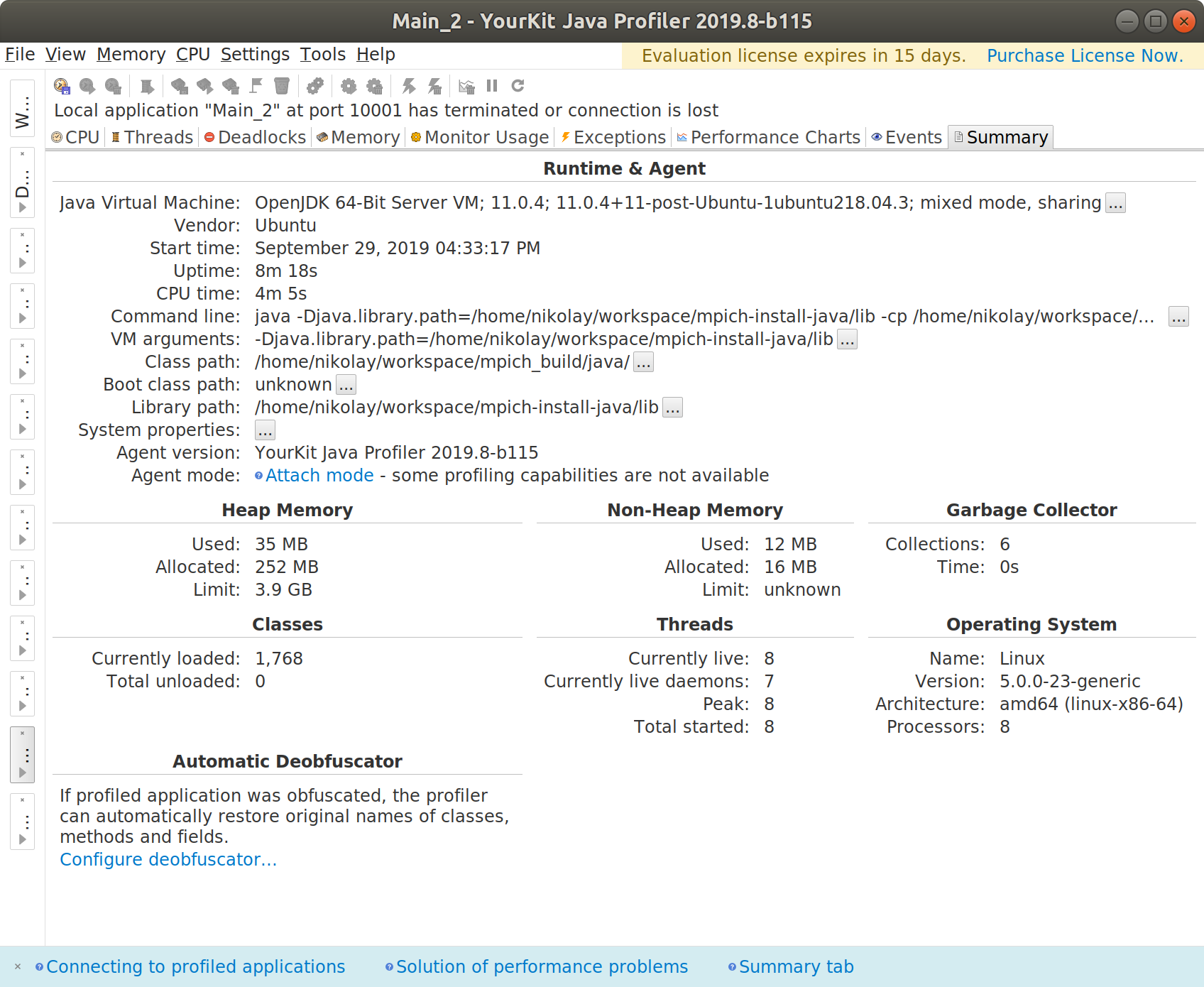
Самият алгоритъм за намиране на единичен най-дълъг път изглежда по следния начин. Започваме с топологичен анализ, като върховете ги запазваме в стек S. След което за Vi търсим най-дългият път, който може да бъде изваден от дага, това го правим с BFS(Bread First Search – алгоритъм за обхождане на граф).

Характеристики:

* 10 процеса;
* машина System 76, модел Galago Pro, процесор Intel Core i7-8565u, RAM 16 GB 2400MHz [System76];
* ОС Ubuntu 18.04 LTS (64-bit).

При **806** върха и **159267** ребра. Средното време на изпълнение на алгоритъма **е 0.245 секунди**, CPU време, като машината достига максимално натоварване. На фиг. 8.2 може да се види снимка на обобщение от изпълнението на инструмент за анализиране на Java процеси и програми [YourKit]. Тук трябва да се има предвид, че подобни инструменти могат да анализират само един java процес, тоест това, което се виждате на фиг. 8.2 е работата само на един процес при повтарящ се алгоритъм.

Резултатите от подобен тест са впечатляващи, макар и MPI комуникацията да е сведена до минимум.

  
Фиг. 8.2

# Заключение

В началото на дипломната работа си поставихме за цел да разработим среда за паралелно програмиране между Java процеси в Android. За целта имплантирахме Java библиотека, интерфейс използващ MPICH като основа. Идеята на цялата разработка беше да се състави Java интерфейс, който позволява MIP семантична разработка на паралелни алгоритми. В заключение мога да кажа, че интерфейсът е успешен и е използваем до толкова, до колкото Java позволява.

Макар и немного, разработените тестове показват, че използването на подобен интерфейс е удобно и оптимално за стандартите на Java. Разбира се има и няколко основни проблема, които са налице.

* MPI предлага множество имплементации, което позволява голяма портативност на приложенията написани на този стандарт. Това обаче не се отнася за нашия проект. Имплементацията на някой функции, както и по-нататъшно оптимизиране налагат java-mpi библиотеката да е обвързана пряко с MPI библиотеката.
* Проблема с типовете данни не е малък, дори сегашната имплементация не е перфектна. Основния, от които може да се посочи е изпращането на обекти. Решението, което даваме е сравнително оптимално и удобно, но изисква още работа и развитие.
* Степента на интегриране на различни Java структури за данни е слаба, да не кажа никаква. Тоест повечето структури като Map и Satck намиращи се в java.util не могат да се използват.

Възможно най-оптималната комуникация, която може да се постигне е чрез директни буфери, java.nio.Buffer. Те предлагат оптимален и бърз начин за пренос на данни между C и Java с минимален, та дори никакво копиране. Което бих казал, че е предпочитаният от мен начин на комуникация.

В крайна сметка, идеята е добра. MPI семантиката дава нови хоризонти за Java. Позволява бърза и проста IPC комуникация, без да се налага разработчика да навлиза в детайли и усложнения.

# Използвана литература

* [Android Developers] <https://developer.android.com/>
* [CH4] CH4 overall designe, <https://wiki.mpich.org/mpich/index.php/CH4_Overall_Design>]
* [Chameleon] Chameleon, <https://www.mcs.anl.gov/~lusk/oldpapers/ibmsysj/node7.html>]
* [Grid Computing] Soha Maad 2012, https://www.intechopen.com/books/grid-computing-technology-and-applications- widespread-coverage-and-new-horizons
* [JNI] Java Native Interface(JNI),

<https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html>

* [horovod] horovod, <https://github.com/horovod/horovod>
* [hwloc] hwloc,

<https://github.com/pmodels/hwloc/tree/c2e5db2a47e0e43f2cb46cc308e6a6585e329314>

* [IBM Blue Gene] IBM Blue Gene,

[https://www.ibm.com/ibm/history/ibm100/us/en/icons/bluegene](https://www.ibm.com/ibm/history/ibm100/us/en/icons/bluegene/)

* [JNI] Java Native Interface Specification

<https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html>

* [Liu] Liu, Feipeng, Android Native Development Kit Cookbook, 2013.
* [MPI Forum] MPI Forum, <https://www.mpi-forum.org/>
* [MPI Standard] A Message-Passing Interface Standard Version 3.1 by Message Passing Interface Forum, June 4, 2015, <https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf>
* [MPICH] MPICH, <https://www.mpich.org/>
* [MPICH SOURCE] MPCH source code, <https://github.com/pmodels/mpich>
* [MPICH WIKI] MPCH wiki, [https://wiki.mpich.org](https://wiki.mpich.org/)
* [Open MPI] Open MPI, <https://www.open-mpi.org/>
* [ROMIO] ROMIO, <https://www.mcs.anl.gov/projects/romio/>
* [SCTP] SCTP linux man page: <https://linux.die.net/man/7/sctp>
* [Stevens] Stevens, W. Richard, Unix Network Pro­gramming, Vol. 2, UNIX network programming interprocess comunications, 1998.
* [System76] System76 Galago Pro, <https://system76.com/laptops/galago>
* [YourKit] YourKit, <https://www.yourkit.com/docs/java/help/running_profiler.jsp>
* [Orange Pi] Orange Pi <https://www.aliexpress.com/item/32969503384.html?spm=2114.search0302.3.1.23bf4d31HZwxZ0&ws_ab_test=searchweb0_0,searchweb201602_0,searchweb201603_0,ppcSwitch_0&algo_pvid=1695a30b-0a87-43eb-bc1e-3e742040003a&algo_expid=1695a30b-0a87-43eb-bc1e-3e742040003a-0>
* [ORACLE docs] Oracle, <https://docs.oracle.com/>
* [MPJ Express] MPJ Express, <http://mpj-express.org/>
* [mpiJava 1.2] Bryan Carpenter, Geoffrey Fox, Sung-Hoon Ko, Sang Lim, Notheast Parallel Architecture Center, Syracuse Universiety, https://www.open-mpi.org/papers/mpi-java-spec/mpiJava-spec.pdf