Парарелно програмиране с Java

Аделина Алексиева - Петрова
aaleksieva@tu-sofia.bg

Java като многонишков език

- В C, C++, и т.н. е възможно да се програмира многонишково, чрез подходяща библиотека:
  - например **pthreads** библиотеката (POSIX).
- Многонишковите библиотеки предоставят един възможен на подход за паралелно програмиране на архитектура с обща памет(shared memory).
- Друг подход е използване на **OpenMP**, който използва директиви за компилатора, разглеждани по-късно.
- За разлика от тях (традиционните HPC) езици, Java интегрира нишките (threads) в основната спецификация на езика (*language specification)* по много по-силен начин.
  - Всяка Java Virtual Machine *ТРЯБВА* да поддържа нишки.
- Въпреки вградената поддръжка, това не прави автоматично многонишковото програмиране в Java *лесно*, тя помага за избягване на чести програмни грешки, и пази кода подреден.

Характеристики на нишките в Java

- Java предоставя набор от примитиви за синхронизация, базирани на концепциите за монитор (**monitor)** и условни променливи (**condition variable)** на Hoare.
- *Синтактично* разширение на езика Java за поддръжка нишките, напр.:
  - **synchronized** – ключова дума за синхронизиран метод.
  - **synchronized** израз.
  - **volatile** ключова дума.
- Други управления на нишките и синхронизации са изнесени в класа **Thread** и свързаните с него класове.
- Но съществуващите нишки имат по-голям ефект е спецификацията на езика и в реализацията на JVM.
  - Например модела на паметта в Java.

Създаване на нишки в програма на Java

- В Java изпълнението на всяка нишка се асоциира с инстанция на класа **Thread**. Преди стартирането на нова нишка трябва да се създаде нова инстанция на този клас.
- Класът **Thread** *имплементира* интерфейса **Runnable**. По този начин всяка инстанция на **Thread** има метод:

```
public void run() { . . . }
```

- Когато нишката се *стартира*, програмният код който се изпълнява е в тялото на метода **run()**.

Реализация

- Два начина:
  1. Наследяване на класа **Thread** и пренаписване на метода **run()**:

```
class MyThread extends Thread {
   public void run() {
      System.out.println("Hello from another thread") ;
   }
}
. . .
Thread thread = new MyThread() ;
```

  2. Създаване на отделен **Runnable** обект и предаването му като параметър на конструктора на **Thread**:

```
class MyRunnable implements Runnable {
   public void run() {
      System.out.println("Hello from another thread") ;
   }
}
. . .
Thread thread = new MyThread(new MyRunnable()) ;
```


Стартиране на Thread

- Създаването на инстанция на класа **Thread** не означава, че стартираме нишка.
- За да се направи това трябва да се извика метода **start()** за всяка инстанция:

```
thread.start() ;
```

Това действие извиква изпълнението на метода **run()** за конкретната нишка.

Пример

```
class MyThread extends Thread {
   MyThread(int id) {
      this.id = id ;
   }
   public void run() {
      System.out.println(“Hello from thread ” + id) ;
   }
   private int id ;
}
. . .
Thread [] threads = new Thread [p] ;
for(int i = 0 ; i < p ; i++)
    threads [i] = new MyThread(i) ;
for(int i = 0 ; i < p ; i++)
    threads [i].start() ;
```


Забележки

- Един начина за създаване и стартиране на **p** на брой нишки, които да се стартират едновременно.
- Изходът на програмата може да изглежда нещо от този вид (за **p = 4**):

  Hello from thread 3
  Hello from thread 4
  Hello from thread 2
  Hello from thread 1

  Разбира се тук не ви е гарантиран редът на изхода, защото всички нишки са конкурентни.
- Това може да е притеснително при много голям брой нишки (massive parallelism).

JVM и системни нишки

- Когато се стартира Java приложени, методът **main()** се изпълнява като една единствена нишка нар. *основна нишка*.
- В най-простия случай – ако методът не създава никакви нови нишки, тогава JVM е стартирана докато не приключи метода **main()** (основната нишка).
- Ако методът **main()** създава нови нишки, тогава JVM приключва когато *всички* създадени нишки се изпълнят.
- Съществуват и т. нар. системни нишки, които се изпълняват на заден фон едновременно с потребителските нишки. Тези нишки са наречени *daemon threads*— и те нямат свойството “keeping the JVM alive”.
- Обикновено потребителските нишки могат да създават системни нишки чрез метода **setDaemon()**.

Взаимно изключване (Mutual Exclusion)

10

Избягване на взаимно влияние (Interference)

- Всяко нетривиално многонишково приложение (или паралелизъм от тип споделена памет shared-memory-parallel) е важен въпрос.
- Най-общо взаимното влияние (или още - race condition) се появи, когато две нишки се опитват да работят върху една и съща променлива в едно и също време. Това се отразява с неверни/разрушени данни.
  - Това не винаги е така. Това зависи от точните инструкции. Тази неопределеност (non-determinism) е най-лошата характеристика на race conditions.
- Решение на проблема е като се въведе някакъв вид примитив за заключване . Това означава, че само една нишка може да отключи дадено заключване за дадено време. Конкурентните програми могат да бъдат написани така, че такива операции, върху дадена група от променливи се извършва само от тези нишки, които държат заключването, свързано с тази група.

11

Пример на Mutex (в C)

Thread A	Thread B
pthread_mutex_lock(&my_mutex) ;	
/* critical region */ tmp1 = count ; count = tmp1 + 1 ;	pthread_mutex_lock(&my_mutex) ;
pthread_mutex_unlock(&my_mutex) ;	Blocked
	/* critical region */ tmp2 = count ; count = tmp2 - 1 ;
	pthread_mutex_unlock(&my_mutex) ;

12

Монитори (Monitors)

- Java разглежда тези проблеми като адаптира версия на концепцията за монитор (*monitors* proposed by C.A.R. Hoare).
- *Всеки* Java обект се създава със собствено заключване (и всяко заключване е асоциирано с обект – тук няма начин да създаден изолиран mutex). В Java това заключване често се нарича мониторно заключване (*monitor lock).*
- Методите на класа могат да се обявят като *synchronized*.
- Заключването на обекта се взима при влизане в синхронизиран метод(**synchronized method**), и се освобождава при излизането му.
  - Статичните синхронизирани методи се различават.
- Нека приемем, че методите най-общо модифицират полетата на обекта върху който са извикани. Това води до естествено и систематично свързване между заключването и променливите, които то предпазва: а именно заключването охранява променливите, за които е закачено.
- Критична секция (*critical region)* става тялото на синхронизирания метод.

Пример

Thread A	Thread B
… call to counter.increment() …	

// body of synchronized method
tmp1 = count ;
count = tmp1 + 1 ;
	… call to counter.decrement() …
	Blocked

… counter.increment() returns …	

	// body of synchronized method
	tmp2 = count ;
	count = tmp2 - 1 ;

	… counter.decrement() returns …

Предимства и недостатъци

- Този подход помага използването на добри практики и прави многонишковото програмиране в Java по-лесно и с по-малко грешки, за разлика например от многонишковото програмиране да речем в езика C.
- Въпреки това:
  - Все още синхронизацията зависи от коректното определяне на критичните секции с цел избягване състояние на състезание.
  - Вътрешното свързване на заключването на обекта и неговите полета зависи от спазването на шаблони от конвенционалното програмиране в среда на ООП (които езика поддържа, но не задължава).
  - Чрез използването на синхронизирана конструкция (*synchronized construct)*, програмата може да се провали.
  - Има много други начини, по които да се въведе ситуация на "мъртва хватка", освен по невнимание да забравим да отключим дадено заключване!
- Паралелното програмиране е трудно, и предположението, че с Java то става някак си по-лесно, най-вероятно в началото да има доста грешни/неработещи правилно програми!

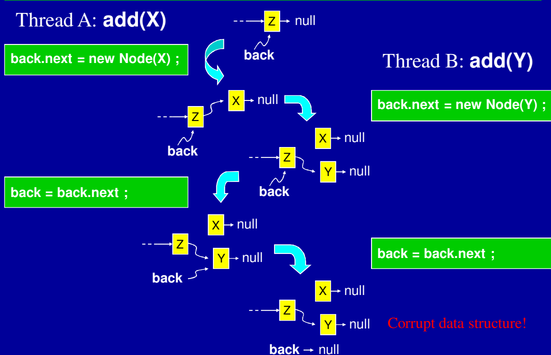
Пример: проста опашка

```
public class SimpleQueue {
    private Node front, back ;
    public synchronized void add(Object data) {
        if (front != null) {
            back.next = new Node(data) ;
            back = back.next ;
        }
        else {
            front = new Node(data) ;
            back = front ;
        }
    }
    public synchronized Object rem() {
        Object result = null ;
        if (front != null)  {
            result = front.data ;
            front   = front.next ;
        }
        return result ;
    }
}
```


Забележки

- Тази опашка е реализирана като свързан списък с указател към началото (първия елемент) на списъка и указател към края (последния елемент) на списъка.
- Метода **add()** добавя възел в края на списъка; метода **rem()** премахва елемент от началото на списъка.
- Класът **Node** има поле данни **data** (от тип **Object**) и поле **next** (от тип **Node**).
- Метода **rem()** връща **null** когато опашката е празна.
- Следващия слайд дава пример за това какво може да се обърка, без използването на концепцията за взаимно изключване (mutual exclusion). То приема, че две нишки добавят елемент към опашката;

Нужда от синхронизация на методи



Конструкция за синхронизиране

- Ключовата дума **synchronized** също може да се използва в *синхронизиран блок*, като синтаксиса е:

```
synchronized (object) {
    … critical region …
}
```

- Тук ***object*** е указател към който и да е обект. Синхронизираната команда първо взема заключването над този обект, после се изпълнява критичната секция, и после освобождава заключването.
- Най-общо може да се каже, че чрез синхронизирана команда може да се заключи който и да е обект, за да се охранява който и да е програмен код.

Производителност при използване на **synchronized**

- Очевидно заключването внася известно забавяне в изпълнението на синхронизиран метод.
- Много от помощните класове в Java (например **Vector**, и други) първоначално са написани като синхронизирани методи, с цел да се използват за многонишково програмиране.
- Това е преосмисляно като грешка. За това са добавени нови класове – например **ArrayList** обикновено нямат синхронизирани методи. Това е оставено на потребителя, когато е необходимо, да си напише класове – обвивка (wrapper classes).

Извън взаимното изключване (Mutual Exclusion)

- Взаимното изключване, предоставено в синхронизираните методи и команди е важен тип синхронизация.
- Съществуват и други интересни форми на синхронизация между нишки. Взаимното изключване само не е достатъчно за реализиране на по-общо взаимодействие между нишките (най-малкото не и ефективно).
- Нишките в POSIX, например предоставят втори вид синхронизация, наречен *condition variable* за реализация на синхронизация между нишките.
- В Java това е вътрешно – при дефинирането на обекта: всеки обект има ефективно една условна променлива, свързана с него.

Работещо изчакване (Busy Waiting)

- Един подход е да добавим метод, който се опитва да извлече данни, докато те се появят:

```
public synchronized Object get() {
    while(true) {
        Object result = rem() ;
        if (result != null)
            return result ;
    }
}
```

- Това работи, но може да е не ефективно, т.е. да се извършва постоянно метода **rem()** в безкраен цикъл, т.к. през това време машината може да се използва от другите нишки.
  - Това не винаги е така: понякога busy waiting е най-ефикасният подход.
- Друга възможност е използването на метода **sleep()** в цикъла, за да разпределяме определено време между операциите по извличане на информация. В такъв случай губим времето за реакция в този интервал.

22

Методи wait() и notify()

- Най-добрият подход е да се използват методите **wait()** и **notify()**. Те са дефинирани в класът **Object**.
- Типично извикването на метода **wait()** вкарва извикващия метод в състояние на изчакване, докато друга нишка не го събуди, чрез извикването на метод **notify()**.
- В нашия пример, ако опашката е празна, то метода **get()** ще извика **wait()**. Това ще блокира операцията **get()**. По-късно, друга нишка, ще извика метода **add()**, като вкара данни в опашката и **add()** ще извика метода **notify()** за да събуди "заспалата" нишка. Тогава метода **get()** ще се изпълни и ще върне стойност.

23

Малък пример

```
public class Semaphore {
    int s ;
    public Semaphore(int s) { this.s = s ; }
    public synchronized void add() {
        s++ ;
        notify() ;
    }
    public synchronized void get() throws InterruptedException {
        while(s == 0)
            wait() ;
        s-- ;
    }
}
```

24

Забележки I

- Вместо свързан списък имаме обикновен брояч, който иска стойността му винаги да е не-отрицателна
  - **add()** увеличава брояча.
  - **get()** намалява брояча, но ако брояча е нула се блокира докато друга нишка не увеличи брояча.
- Примерните данни са опростени, но конструкциите са същите, които се използват, ако трябваше да блокираме опашка.
- Може да се разпознае в тази реализация класически семафор (semaphore) – важен синхронизиращ примитив.

Забележки II

- **wait()** и **notify()** се използват само в синхронизирани методи.
- Методът **wait()** "паузира" извикването на нишката. Т.е. реализира заключване, където нишката се държи от обект.
- Няколко нишки могат да **wait()** паралелно на един и същи обект.
- Методът **notify()** "събужда" само една нишка от тези нишки. Ако няма чакаща нишка, то методът **notify()** не извършва нищо.
- Методът **wait()** хвърля изключение **InterruptedException**.

Друг пример

```
public class Barrier {
    private int n, generation = 0, count = 0 ;
    public Barrier(int n) { this.n = n ; }
    public synchronized void synch() throws InterruptedException {
        int genNum = generation ;
        count++ ;
        if(count == n) {
            count = 0 ;
            generation++ ;
            notifyAll() ;
        }
        else
            while(generation == genNum)
                wait() ;
    }
}
```

Забележка

- Този клас реализира т.нар **бариерна синхронизация** (barrier synchronization)— важна операция в паралелното програмиране в споделена памет.
- Тя синхронизира **n** процеса: когато **n** нишки правят обръщение към метода **synch()** първите **n-1** нишки са блокирани, докато последната нишка преминава бариерата.
- Метода **notifyAll()** е обобщение на метода **notify()**. Той събужда **всички** нишки, чакащи за този обект.
  - Много автори препоръчват използването на метода **notifyAll()** като по "сигурен" от метода **notify()**, и препоръчват винаги да се използва само **notifyAll()**.
- В нашия пример – генерираният номер означава текущата обща операция на бариерата: тогава остава само да се контролира цикъла while до метода **wait()**.
  - И този цикъл е нужен само за да се спази шаблона за стандартното използване на **wait()**- споменато по-рано.

Заключителни бележки за синхронизацията

- Илюстрирахме с два примера, че методите **wait()** и **notify()** позволяват различни интересни шаблони за синхронизация (или комуникация) на нишки да бъде реализирана.
- По някои показатели, тези примитиви са достатъчни за реализацията на общо паралелно програмиране – всеки шаблон от синхронизацията може да бъде реализиран на базата на тези примитиви.
  - Например, може лесно да се реализира изпращане на съобщение между нишки
- Това не означава, че това е всичко в синхронизацията: например за разширяема паралелна обработка някой може да иска по-ефективна имплементация.

Други характеристики на нишките в Java

Join

- **Thread** API има фамилия от **join()** оператори. Те имплементират друг прост подход за синхронизация, при която една нишка чака друга да приключи изпълнението си:

```
Thread child = new MyThread() ;
child.start() ;
… Do something in current thread …
child.join() ;    // wait for child thread to finish
```


Priority и Name

- Нишката има свойства **priority** и **name**, които могат да се дефинират в подходящи set методи, преди стартирането на нишката.

Sleeping

- Може да се накара нишката да "заспи" за някакъв интервал чрез метода **sleep()**.
- Тази операция обаче не е толкова мощна, колкото при **wait()**. Не е възможно друга нишка да я "събуди".

Забранени методи в класа Thread

- Има група методи в класа **Thread,** които поддържат т.нар. външен контрол на времето на живот на нишките (external “life-or-death” control).
- Тези методи никога не са били надеждни, и вече официално са маркирани като неизползваеми (“deprecated”). Използването им трябва да бъде избягвано.
- Ако трябва някоя да се прекъсне работата на стартирана нишка, **трябва явно да се укаже**, че нишката “слуша” за състояние в което може да се прекъсне.
- По-интересните забранени методи са::
  - **stop()**
  - **destroy()**
  - **suspend()**
  - **resume()**

Прекъсване на нишки

- Прекъсването става чрез извикването на метода **interrupt()** в нишката, която искаме да се прекъсне.
- Това прекъсване не става непременно: програмния код, на нишката която искаме да прекъснем трябва явно да провери дали не е бил прекъснат, например:

```
public void run() {
    while(!interrupted())
        … do something …
}
```

Тук метода **interrupted()** е статичен метод на класа **Thread,** който определя дали текущата нишка е прекъсната.
- Ако нишката изпълнява блокираща операция, например като **wait()** или **sleep()**, операцията може да завърши с изключението **InterruptedException**. Прекъснатата нишка трябва да хване това изключение и да се спре сама.
- Този механизъм зависи изцяло от реализацията на кода на нишката. Програмистът трябва да реши дали някоя нишка може да поддържа възможността да се прекъсва – много често това не се налага.

ThreadGroup

- Съществува механизъм за организирането на нишките в групи. Това може да е полезно за налагане на ограничения от гледна точка на сигурността, и кога една нишка може да прекъсне друга нишка.
- За повече информация виж документацията на класа **ThreadGroup**.

ThreadLocal

- Обект от класа **ThreadLocal** съхранява обекти, които имат различна локална стойност за всяка нишка.
- За пример – нека предположим, че имплементираме интерфейс за комуникация, базиран на MPI, който съпоставя на всеки MPI процес по една нишка в Java. Тогава може да решите че трябва да има статична променлива в класа **Comm**. Но тогава идва въпроса, как ще се извика метода **rank()** , за да върне различен ID за всяка нишка, и как би работило следното извикване:

```
int me = Comm.WORLD.rank() ;
```

- Един възможен подход е да съхраняваме ID на процеса в обекта за комуникация в т.нар. *локални за нишката променливи (thread local variable)*.
  - Друг подход е да използвате хеш-карта (hash map) която е индексирана чрез **Thread.currentThread()**.
- За подробности виж API на класа **ThreadLocal.**

Променливи от тип Volatile

- Нека предположим, че дадена променлива трябва да е достъпна от няколко нишки, но по някаква причина сте решили, че не искате програмата да се забавя от използването на ключовата дума synchronized.
- Най-общо Java не гарантира следното – при липса на заключващи операции насилствено да синхронизира на паметта – т.е. Стойностите на променливите, записани в една нишка, ще е видима в другите нишки.
- Но ако декларираме дадено поле като *volatile*:

```
volatile int myVariable ;
```

тогава се очаква JVM да синхронизира стойността на това поле във всяко локално за нишката поле или още т.нар. кеширано (cached) поле с копие на променливата със централнойто място за съхранение (и правейки го видимо за всички нипки) – тогава всеки път стойността на променливата ще се обновява.
- Точното значение на volatile променливата, и модела на паметта в Java по-общо е все още неясно виж за пример тук.
  "A New Approach to the Semantics of Multithreaded Java",
  Jeremy Manson and William Pugh,
  http://www.cs.umd.edu/~pugh/java/memoryModel/

java.util.concurent

- Future интерфейса допълващ съществуващият Callable интерфейс. Callable позволява да се стартира самостоятелна нишка, която да върне резултат при завършването си. Future допълва това, като позволява да получим референция към тази нишка. Така след като тя е стартирала може да изпълним блокираща get операция и да вземем резултатът й.
- Executor и ExecutorService. Те позволяват да се създаде басейн от нишки решаващи определена задача. Когато една от нишките приключи, тя не се унищожава, а се запазва в басейна. При пристигане на нова заявка изискваща паралелна обработка, директно и се дава една от свободните нишки. Предимството тук е, че се пести време от създаване и зачистване на нишки.

OpenMP и Java

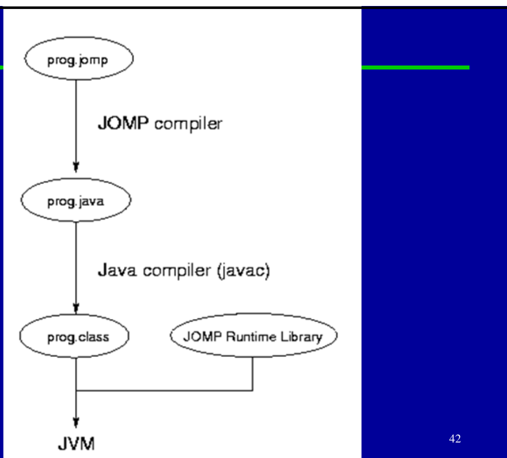
- *OpenMP* (**www.openmp.org**) е добре известен стандарт за паралелно програмиране.
- Съдържа набор от *директиви* и *библиотеки* включени в езика, обикновено Fortran или C/C++.
  - Директивите позволяват да се дефинират *parallel regions*, изпълнявани от нишки, и *work-sharing directives*, които дефинират как итерациите на цикъла се поделят между тези нишки и тяхната синхронизация от типа бариерна синхронизация или критичен регион.
- Съществува реализация на OpenMP за Java от Edinburgh Parallel Computing Center — нар., JOMP.

  http://www.epcc.ed.ac.uk/research/jomp/

Модел на изпълнение на JOMP

- Програмата на Java, която съдържа JOMP директиви трябва да е написана във файл с разширение **.jomp**.
- Този файл се подава на JOMP компилатора, който генерра криптиран **.java** файл, който включва runtime библиотеките необходими за управлението на нишките за обработка на паралелни региони, извършване на синхронизация и т.н.
- Този файл се компилира **javac**, за да се получи байткода.
- Изпълнението на резултата се стартира чрез комадата

  java -Djomp.threads=10 MyClass



JOMP Директиви

- JOMP директивите се синтактично Java коментари.
- В JOMP синтаксиса на *конструкцията* е:

  //omp *directive*
  Java-statement

  където *Java-statement* е блок от оператори **{ }**.
- Има JOMP директиви, които не са свързани с някакъв Java блок (например директивата **barrier**).

43

JOMP дирактивата: **parallel**

- Синтаксис:

  //omp parallel
  Java-statement
- Друг вариант на тази директива.:

  //omp parallel shared(*vars1*) private(*vars2*) …
  Java-statement

  Където ***vars1***, ***vars2*** са имена на променливи. Това контролира дали променливите са поделени или частни на нишките в паралелния регион.

44

JOMP директива: **for**

- Синтаксис на ***for***:

  //omp for
  for(*integer-type var* = *expr* ; *test-var* ; *incr-var*)
  Java-statement
- Конструкцията **for** трябва да е вложена в **паралелна** директива – итерациите се разделят между активните нишки.
- Компилаторът поставя бариерна синхронизация в края на всеки оператор for, която може да се изключи чрез клаузата ***nowait***:

  //omp for nowait …
  Java-for-statement
- Може да се управлява приоритета чрез клаузата ***schedule***:

  //omp for schedule(*mode*, *chunk-size*) …
  Java-for-statement

45

sections и *section* директиви

- Директивата *sections* се използва за да покаже броя на секциите които се изпълняват паралелно:

- Пример:

```
//omp sections [nowait]
{
    //omp section
    <code block>

    [//omp section
     <code block>]...
}
```

```
//omp parallel shared(a,b,c)
  {
//omp sections
     {
//omp section
       a.init();

//omp section
       b.init();
//omp section
       c.init();
     }
  }
```

JOMP директиви: **master, critical, barrier**

- Синтаксис на ***master***:

  //omp master
  Java-statement

- Синтаксис на ***critical***:

  //omp critical *[name]*
  Java-statement

- Синтаксис на ***barrier*** :

  //omp barrier

Всичките тези директиви се използват в паралелните секции.

Пример

```
public class Hello {
  public static void main (String argv[]) {
     int myid;
//omp parallel private(myid)
     {
     myid = OMP.getThreadNum();
     System.out.println("Hello from " + myid);
     }
  }
}
```

Пример

```
import jomp.runtime.*;

public class Hello {
   public static void main (String argv[]) {
      int myid;
      __omp_class_0 __omp_obj_0 = new __omp_class_0();
      try {
         jomp.runtime.OMP.doParallel(__omp_obj_0);
      }
      catch (Throwable __omp_exception) {
         jomp.runtime.OMP.errorMessage();
      }
   }
```

Паралелизация за клъстер

- PJ - MPI библиотека разработена за Java
- Parallel Java - (http://www.cs.rit.edu/~ark/pj.shtml)
- проф. Алан Камински от Рочестърския Институт за Технологии
- PJ се поддържа активно (последния към момента билд е от месец април 2010)
- съвместима със най-новите възможности в Java
- следва обектно ориентирания модел заложен в Java, а не разчита на декларативни техники

Пример

- За да използваме PJ трябва още в началото на програмата да инициализираме комуникационния обект Comm. Може да разберем свободните възли (size) и уникалния идентификатор на нашия възел (rank). Пример:

```
Comm.init(args);
Comm world = Comm.world();
int size = world.size();
int rank = world.rank();
```

Пример

- На всички възли от клъстера е нужно да имаме JAR-а на библиотеката.
- В главния възел имаме конфигурационен файл описващ останалите възли, както и вървящ PJ Scheduler – фонов процес, който трябва да стартира със зареждането на операционната система.
- Когато нашата програма използва PJ и бъде стартирана, тя се свързва с PJ Scheduler-а и разбира колко възела от клъстера са на разположение.
  - Ако такива няма, приложението се изпълнява само на един компютър.
  - В повечето случаи обаче, има възли на разположение и паралелната работа може да продължи както е дефинирана в алгоритъма.

Създаване на конфигурационен текстов файл на интерфейсния възел

cluster *<име-на-клъстера>*
logfile *<абсолютен-път-до-лог-файла-на-PJ-Scheduler>*
webhost *<хост-за-уеб-интерфейса-на-PJ-Scheduler>*
webport *<порт-за-уеб-интерфейса-на-PJ-Scheduler:8080>*
schedulerhost *<хост-където-PJ-Scheduler-слуша-за-конекции-от-клиентските-програми>*
schedulerport *<порт-където-PJ-Scheduler-слуша-за-конекции-от-клиентските-програми>*
frontendhost *<хост-на-който-процесите-на-главния-възел-се-свързват-с-процеси-на-задните-възли>*
backend <име-на-възела> <брой-cpu> <хост-на-възела> <път-до-JVM-на-този-възел> <classpath-запрограмите-стартирани-на-възела> <флагове-за-JVM...>
jobtime <максимално-време-което-отделна-задача-може-да-върви>

(mpiJava)

```
import mpi.* ;

  class Hello {
    static public void main(String[] args)
                     throws MPIException {

      MPI.Init(args) ;
      int my_rank; // Rank of process
      int source;  // Rank of sender
      int dest;    // Rank of receiver
      int tag=50;  // Tag for messages
      int myrank = MPI.COMM_WORLD.Rank() ;
      int      p = MPI.COMM_WORLD.Size() ;
    }
  }
```

Използвани и адаптирани ресурси

- Multithreaded and Shared-Memory Programming in Java, Instructor: Bryan Carpenter, Pervasive Technology Labs, Indiana University
- Programming Models for Parallel Java Applications, Mark Bull and Scott Telford, Edinburgh Parallel Computing Centre, Edinburgh, EH9 3JZ