A parallel programming language extension for Java

Patrick Viry, LzLabs

geecon 2023
Kraków, 19-21 April

geecon

sli.do #geecon

Bio: everything about languages

I have worked on..

- Language theory and semantics (PhD on term rewriting and concurrency)
- DSL (domain specific languages) design and implementation
- Parsing techniques, program analysis
- Tooling support (IDEs)
- Source-to-source transformation
- Migration of legacy applications

lzlabs.com/careers
patrick.viry@lzlabs.com

Extending general purpose languages (GPLs)

Domain-specific languages (DSLs) tend to grow into General Purpose Languages (GPLs)

Ateji : a defunct software startup to provide DSLs as extensions to GPLs

A dozen experiments, two products :

- OptimJ (logical variables, constraints, objectives, …)
- Ateji PX (parallel programming) ← this talk

Models of parallelism (1/2)

Threads
- a hardware-level concept, not a high-level language construct
- most multi-threaded programs are buggy

Edward A. Lee, "*The Problem with Threads*", Berkeley Technical Report No. UCB/EECS-2006-1.

Task parallelism
- fork/join (Unix), Intel Thread Building Blocks, …
- Dedicated languages such as Cilk and Erlang. The basic parallel operator "spawning" a task.

Message passing
- MPI (scientific computing)

Models of parallelism (2/2)

OpenMp / OpenACC

- Data parallelism
- Write sequential code in C/C++, then add parallel directives

CCS / Pi-calculus

- Academic (process algebras), Occam
- Compositional operators ← *that's the main idea*

Ateji PX is based on composing parallel branches. This is essential for writing structured and analyzable code, that programmers and tools can reason about. It also makes parallel code more intuitive, closer to the way we think about and understand the notion of parallelism.

It is based on a sound theoretical foundation and can easily emulate all the above models.

Parallelism at the language level

a = a+1 ; b = b+1

first increment a, then increment b

a = a+1 || b = b+1

increment a and increment b, in no particular order

“Compositional”

(ie. NOT “start a thread” or “spawn a task”)

What we can express with the compositional parallel operator

Patrick Viry, LzLabs

geecon 2023
Kraków, 19-21 April

Data parallelism

Add quantifiers to the parallel composition operator (generators and filters)

```
[
    || (int i:N, int j:N, if i+j<N) matrix[i][j]++;
]
```

Special case : “Parallel for” to easily parallelize existing sequential code.

```
for(int i : I) {
    ...
}
```

||

```
for||(int i : I) {
    ...
}
```

Recursive parallelism

Each recursive call creates parallel branches.

```
int fib(int n) {
    if(n <= 1) return 1;
    int fib1, fib2;
    // recursively create parallel branches
    [
        || fib1 = fib(n-1);
        || fib2 = fib(n-2);
    ]
    return fib1 + fib2;
}
```

Speculative parallelism

```
int[] sort(int[] array) {
    [
        || return mergeSort(array);
        || return quickSort(array);
    ]
}
```

Concise and intuitive, isn't it ?

Terminating branches properly with non-local exits (return, exceptions, break) is terribly difficult to get right when using threads or tasks.

Parallel reductions

Values

```
int sumOfSquares = `+ for||(int i : N) (i*i);
// sumOfSquares = 0*0 + 1*1 + ... + (N-1)*(N-1)
```

Collections

```
Set<String> s = set() for||(Person p : persons) p.name;
// s = { p1.name, ..., pN.name }
```

Streams are an operational description, this is an algebraic description : `+ and set() are monoids
Ateji PX also has algebraic collections… but that’s for another time.

Performance

Implemented using native threads (because that's the hardware model) –> similar performance
Annotations to finely control parallelism

Easier to focus on the algorithm, to debug, to experiment different ways of slicing a problem

```java
final int nThreads = System.getAvailableProcessors();
final int blockSize = I / nThreads;
Thread[] threads = new Thread[nThreads];
for(int n=0; n<nThreads; n++) {
  final int finalN = n;
  threads[n] = new Thread() {
    void run() {
        final int beginIndex = finalN*blockSize;
        final int endIndex = (finalN == (nThreads-1))?
                             I :(finalN+1)*blockSize;
        for( int i=beginIndex; i<endIndex; i++) {
          for(int j=0; j<J; j++) {
            for(int k=0; k<K; k++) {
              C[i][j] += A[i][k] * B[k][j];
    }}}}};
    threads[n].start();
}
for(int n=0; n<nThreads; n++) {
  try {
    threads[n].join();
  } catch (InterruptedException e) {
    System.exit(-1);
  }
}
```

Parallel Matrix Multiplication with Java threads

```java
for||(int i : I) {
    for(int j : J) {
        for(int k : K) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Parallel Matrix Multiplication with Ateji PX

Message passing

Sending and receiving messages

```
// declare a channel visible by both branches, and instanciate it
Chan<String> chan = new Chan<String>();
[
    // send a value over the channel
    || chan ! "Hello";
    // receive a value from the channel, and print it
    || chan ? s; System.out.println(s);
]
```

Non-determinism

Read two values from two channels, in no particular order:

```
[ in1 ? Value1; || in2 ? Value2; ]
```

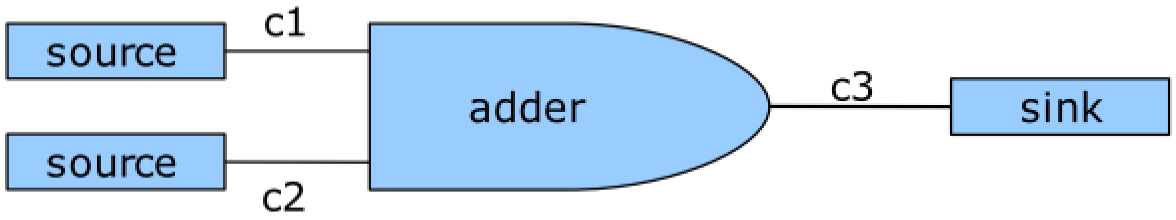
Either read a value from chan1 and print it, or read a value from chan2 and print it.

```
select {
    when chan1 ? v : println("1: " + v);
    when chan2 ? v : println("2: " + v);
}
```

Typical of a server accepting connections.
Similar to the Unix select() system call and the Java NIO Selector API.

Data-Flow, Actors, ...

```
void adder(Chan<Integer> in1, Chan<Integer> in2,
                Chan<Integer> out) {
    for(;;) {
        int value1, value2;
        [ in1 ? value1; || in2 ? value2; ];
        out ! value1 + value2;
}}
```



Distributed parallelism

Patrick Viry, LzLabs

Remote computing

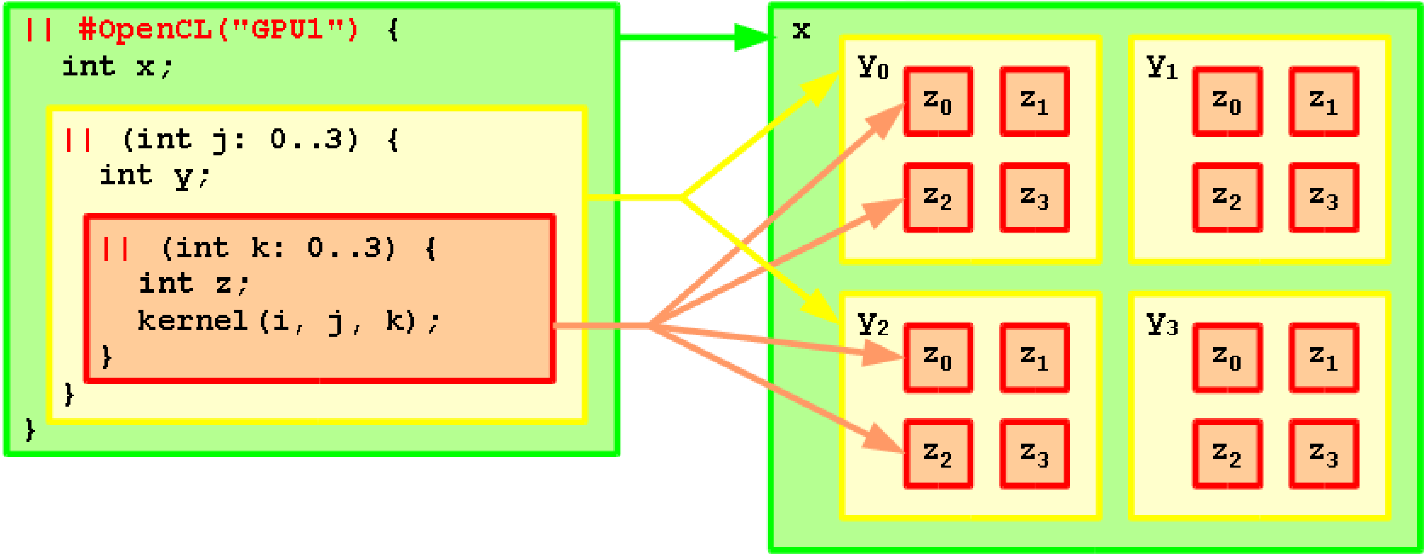
Locate branches with the #IP annotation

Exchange data via message passing with ! and ? operators.

```
// declare a channel visible by both branches, and instanciate it
Chan<String> chan = new Chan<String>();
[
    // send a value over the channel
    || #IP("192.168.0.2") chan ! "Hello";
    // receive a value from the channel, and print it
    || #IP("192.168.0.3") chan ? s; System.out.println(s);
]
```

GPU computing

- Locate branches with the #OpenCL annotation
- Only a small subset of Java. Same code works on CPU and GPU.
- Explicit message passing with ! and ? operators.
- GPU memory hierarchy as lexical scope.



Correctness of parallel programs

Patrick Viry, LzLabs

geecon 2023
Kraków, 19-21 April

Threads

- Notion of “thread safety”
- Code correctness requires intricate thinking and inspection of the whole program

Compositional operators

We can prove correctness at the point of composition.

- Local
- Specific
- Can be (somewhat) automated

Based on a theoretical model

- Pi-calculus : all our operators are defined there
- When not sure about the expected behavior, refer to pi-calculus

Takeaways / Food for thought

- Remember one word : **"Compositional"**

- Concise and intuitive
- As efficient as threads + blocking (more if we consider developer productivity)

- Language extensions are great but difficult :
  - Adoption
  - Tooling support
  - Compatibility between different extensions

- Implementation will benefit from project Loom / JEP 425 (lightweight fibers / continuations vs. heavy threads – currently in preview)

Learn more

Lots of code samples on github.com/PatrickViry/Ateji

- Implementation of the synchronization core available
- Tooling support out of date

These ideas apply to most languages, not just Java
You're welcome to reuse / extend

Contact patrick.viry@lzlabs.com