An ATEJI Whitepaper

Ateji PX for Java

"Parallel programming made simple"

by Patrick Viry

```
[ hello(); || world(); ]
```

Summary

Ateji® PX for Java™ introduces parallelism at the language level, extending the sequential base language with a small number of parallel primitives. This makes parallel programming simple and intuitive, easy to learn, efficient, provably correct and compatible with existing code, tools and development processes.

www.ateji.com

Table of Contents

Parallelism at the language level....3
Language extensions....3
The road to performance....4
The problem with threads....5
Ateji PX features....6
An overview of Ateji PX language constructs....7
  Task parallelism....7
  Data parallelism....8
  Parallelizing legacy code....9
  Recursive parallelism....10
  Speculative parallelism....10
  Parallel reductions....11
Ateji PX constructs for distributed parallelism....12
  Sending and receiving messages....12
  Data-flow, Streams, Actors and MapReduce....13
  Expressing non-determinsm....14
  Selecting....14
  And more....14
Compatibility....15
Thinking parallel....15
Tool support....16
Further reading....16

Parallelism at the language level

Traditional programming languages compose statements sequentially, typically separating them with the "`;`" symbol :

```
a = a+1 ; b = b+1
first increment a, then increment b
```

Figure 1: sequential composition (hypothetical sequential language using "`;`" as separator)

In contrast, a parallel language can compose statements without imposing any ordering on them. In Ateji PX, this is denoted by the "`||`" symbol:

```
a = a+1 || b = b+1
increment a and increment b, in no particular order
```

Figure 2: parallel composition (hypothetical parallel language using "`||`" as separator)

Most mainstream programming languages do not allow the parallel composition of statements. Alternatives had to be developed, typically in the form of threading libraries that make hardware-level or OS-level threading primitives accessible to the application developper. These ad-hoc approaches remain difficult to use because they still rely on a sequential language.

A parallel language does not only make life easier for the programmer, it also enables the compiler to understand parallel programs and provide services such as :

- detection of bugs related to parallelism,
- high-level code optimization,
- automatic adaptation to different target architectures (multi-core, grid, cloud, etc.).

Language extensions

Even though parallel programming boils down to a language problem, developpers don't want to learn a new language, retrain their team, buy new tools and throw away their source code.

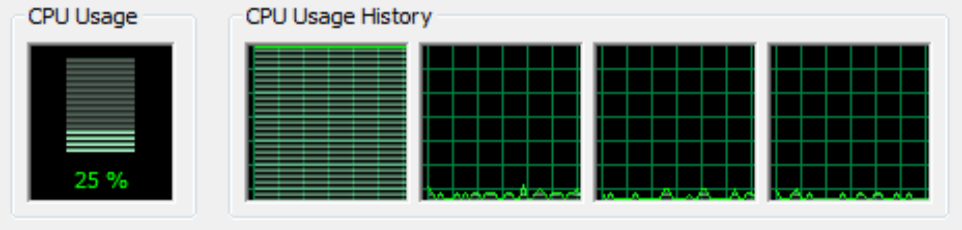
This is where language extensions come into the picture. Ateji PX builds upon existing languages, such as Java, for providing compatibility with existing sequential code, tools and training.

The extension provides only parallelism-related aspects, with the addition of a couple of syntactic constructions that can be learned in a few hours for those already proficient in the base language. No need to learn yet another way to write 1+2.

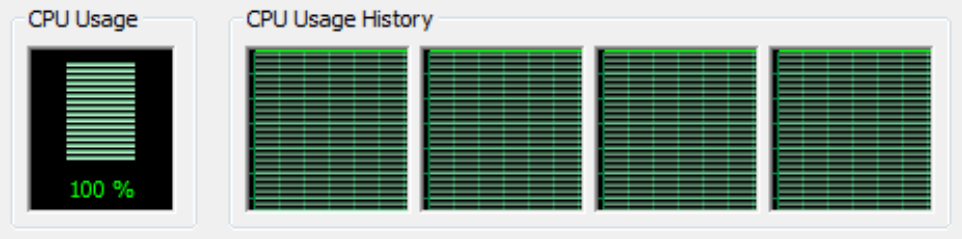
Ateji PX is implemented by source-to-source translation to the base language. This ensures compatibility with existing development tools and avoids technology lock-in (programmers can always switch back to the base language should they wish to do so). This also helps keeping the base language lean and standard.

The road to performance

Performance is the major reason for embracing parallel programming (but not the only one, read on). A sequential program at full power on a multi-core computer will use only one core, regardless of the available hardware:

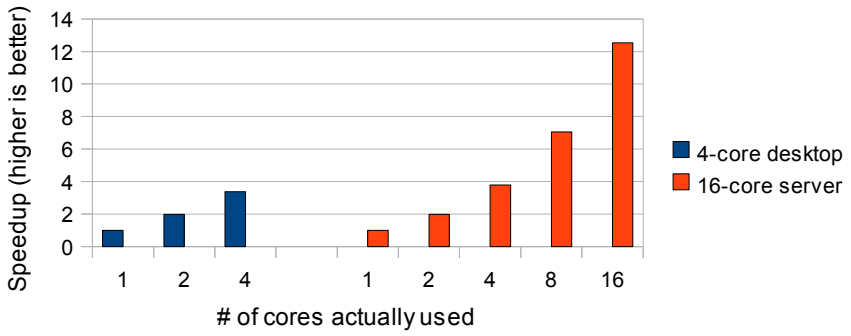


A parallel program at full power on the same hardware will use all available cores:



But parallelizing programs is far from being a trivial task without appropriate tools. Multi-threaed programs based on thread or task libraries are well known for crashing in unexpected ways, being hard to test and debug, and difficult to evolve (see section “The problem with Threads”, page 5).

With Ateji PX, parallelizing an application can be as simple as inserting a “||” operator in the source code. The whitepaper “*Matrix Multiplication with Ateji PX*”[1] illustrates this process on a standard example, achieving a 12.5x speedup on a 16-core server:



Ateji PX perfoms most of its job at compile-time and adds little or no overhead at run-time. The programmer is freed from irrelevant technical details and can more easily concentrate on performance improvement. Experimenting with different parallelization strategies requires little changes in the source code.

1 Available at http://www.ateji.com/multicore/whitepapers.html

The problem with threads

The concept of thread is a low-level hardware concept promoted by multi-core processors. Naturally the first programs for those processors were written using simple libraries providing direct access to the hardware concept.

But threads are very ill-suited as a programming concept. This has been extensively studied and described in "*The Problem with Threads*"[2], a Berkeley technical report by Edward A. Lee. In short:

- threads are a hardware-level concept, not a practical abstraction for programming
- threads do not compose
- code correctness requires intricate thinking and inspection of the whole program
- most multi-threaded programs are buggy

Just think about it: adding a single *method* call to `Thread.start()` radically changes the behavior of a whole program! Against such black magic, development tools are of little help.

On the opposite, the parallel programming model of Ateji PX is composable, intuitive (close to our intuition of what parallelism "is") and based on a sound mathematical foundation.

Another problem with threads is developper productivity. The whitepaper "*Matrix Multiplication with Ateji PX*"[3] shows an example of the same code, written using Java threads (left) and using Ateji PX (right). They have the same performance, but the threaded version is clogged with technical details irrelevant to the algorithm, takes time to write, is hard to read, understand and maintain.

```
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

```
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

For the most part, Java task frameworks exhibit the same problems (they basically just make explicit the scheduler behind threads).

2 Technical Report No. UCB/EECS-2006-1, http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf
3 Available at http://www.ateji.com/multicore/whitepapers.html

Ateji PX features

Parallelism made simple and intuitive

Ateji PX offers parallel programming in a clear and concise syntax, close to our intuition of what parallelism is. Parallel programming becomes accessible to all application programmers.

Compatible, based on standards

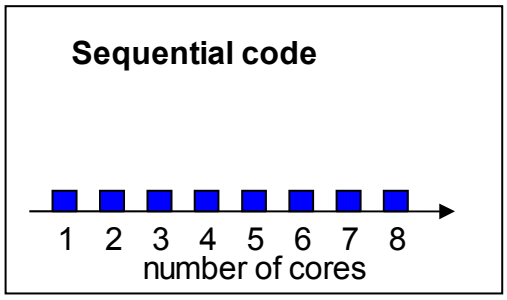
Source-to-source translation enables direct compatibility with current languages, tools and development processes. Ateji PX for Java is integrated within the Eclipse IDE.

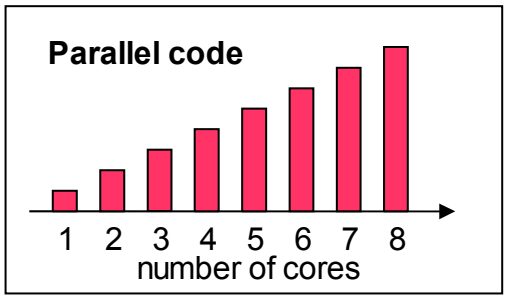
Easy learning, easy staffing

No need to learn a new language or new tools. Ateji PX adds a few additional syntactic constructs to a familiar language, and is integrated within a standard IDE. Experience shows that users typically feel at ease after one day.

Multiple models of parallelism

Not all parallel applications are alike. Some are best described by decomposing problems in the data domain or in the task domain, using shared-memory or message passing. Ateji PX covers all of them with one single tool.





Performance

Ateji PX makes the best use of available hardware, independently of the number of cores. Benchmarks demonstrate performance on par with specialized libraries. The programmer is freed from irrelevant details and has more time for concentrating on performance issues.

Parallel programs that actually work

There is no point having a fast program if it crashes or provides wrong results. Ateji PX is designed upon a sound theoretical model, which helps the programmer and the compiler to better understand parallel code.

Parallelization of existing applications

Parallelizing applications typically consists in simply adding a few "||" symbols in the source code. No need to rewrite, translate or instrument legacy code.

Future-proofing

Since the Ateji PX compiler knows about parallelism, it is able to map the same source code to different hardware, protecting investment in source code as hardware evolves.

An overview of Ateji PX language constructs

Task parallelism

Statements or blocks of statements can be composed in parallel using the "||" operator inside a parallel block, introduced with square brackets:

```
[
    || a++;
    || b++;
]
```

Figure 3: Two parallel branches

or in short form:

```
[ a++; || b++; ]
```

Figure 4: Two parallel branches (short form)

Each parallel statement within the composition is called a *branch*. We purposedly avoid using the terms "task" or "process" which mean very different things in different contexts.

Accesses to shared memory from different parallel branches must be protected as usual. However, since Ateji PX is based on a sound theoretical foundation, it is possible to guarantee at compile-time that parallelism does not introduce new bugs. Ateji is working on an interactive verifier that will help the programmer locate potential problems and ensure the correct behavior of parallel programs. Watch www.ateji.com for availability.

Task parallelism is provided by various libraries (the fork()/join() system calls in Unix, the forthcoming fork/join library in Java7, Intel Thread Building Blocks, etc) and by dedicated languages such as Cilk[4] (based on C/C++) and Erlang[5] (a purely functional language).

The basic parallel operator in Cilk and Erlang is "spawning" a task. In contrast, 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.

---

4 http://www.cilk.com/
5 http://www.erlang.org/

Data parallelism

Branches in a parallel composition can be *quantified*. This is used for performing the same operation on all elements of an array or a collection:

```
[
    // increment all array elements in parallel
    || (int i : N) array[i]++;
]
```

Figure 5: A quantified parallel branch

The equivalent sequential code would be:

```
// increment all array elements one after the other
for(int i : N) array[i]++;
```

Figure 6: Equivalent sequential code

Quantification can introduce an arbitrary number of generators (iterators) and filters. Here is how we would update the upper left triangle of a matrix

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

Figure 7: Multiple generators and filters

Data parallelism is familiar to HPC (High-Performance Computing) experts, and typically used for engineering application such as large-scale simulations. A popular tool for data parallel programming is OpenMP[6], consisting in a collection of C/C++ preprocessor pragmas that can express for example that a sequential for loop should be run in parallel:

```
#pragma omp parallel
for(int i=0; i<N; i++) array[i]++;
```

*Unlike Ateji PX, OpenMP requires the programmer to first write sequential code (in this example, "*first handle the case i=0, then increment i, then handle the case i=1, and so on*"), only to tell the compiler that these sequentiality constraints are actually meaningless (in this example, "*handle all cases in whatever order*"). The underlying C compiler has no knowledge about parallelism and is unable to tell if a parallel annotation is likely to introduce problems or not.*

The basic OpenMP constructs for parallelism have a simple and straightforward expression in Ateji PX in terms of parallel compositions or reductions.

---

6 http://openmp.org/

Parallelizing legacy code

The parallel-for syntax:

```
for||(int i : N) array[i]++;
```

Figure 8: Parallel For

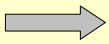
is roughly equivalent to a parallel block which contains exactly one quantified branch:

```
[
    || (int i : N) array[i]++;
]
```

Figure 9: Equivalent to the Parallel For above

Parallel-for is convenient for parallelizing legacy sequential code : simply add a "`||`"operator after the `for` keyword and the sequential loop becomes parallel.

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



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

Figure 10:Parallelizing an existing for loop

Note the convenient notation `int i : N` for iterating `i` over `0 .. N-1`, that can be used indifferently in a sequential or parallel for loop.

Ateji PX takes care of all technical details such as termination, index splitting, local variables, non-local exits and exceptions. Handling all these issues by hand would be verbose, time-consuming and error-prone.

For computational intensive programs, it is often possible to identify a handful of for loops that can be parallelized and provide good parallel speedup (see for instance the whitepaper "*Matrix Multiplication with Ateji PX*").

The transformed code is immediately usable when branches (loop iterations) do not share state. Otherwise, accesses to shared data must be protected using standard techniques such as locks and synchronized methods. Because parallelism is a language construct, based on a sound theoretical foundation, the interactive verifier for Ateji PX will be able to help the programmer identifying all accesses that may require protection, making sure that introducing parallelism does not introduce bugs.

Recursive parallelism

Parallel branches can be created recursively. This is often used in divide-and-conquer algorithms. Here is a simple example computing the Fibonacci series in parallel (this is a naive algorithm for the purpose of exposition):

```
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;
}
```

Figure 11: Recursive parallelism

Speculative parallelism

Parallelism is speculative when branches are created without knowing at the time of creation if the result they produce will actually be needed or not. A basic example consists in launching two different sort algorithms in parallel, and returning from the computation as soon as one of them returns:

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

Figure 12: Speculative parallelism

The two return statements implement non-local exits out of the parallel composition. As soon as one of the branches returns, all other branches are interrupted and their result is discarded. Other non-local jump or exit statements such as `break`, `continue` and `throw` behave similarly.

Note that this behavior is particularly difficult to get right when using threads, tasks or even dedicated "futures" libraries: it involves sharing state between branches, passing and returning values, possibly handling multiple exceptions, making local variables and parameters visible, and terminating branches properly.

Parallel reductions

Parallel reductions form a special case of data parallelism. The term “reduction” is used in parallel programming with the meaning of “reducing” a collection of data into a single value, for instance computing the sum of all elements of an array.

Ateji PX provides a concise and efficient notation for parallel reductions. Here is the parallel sum of all squares from 0 to N-1 :

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

Figure 13: Parallel reduction (sum)

Having parallel reductions at the language level is important for many reasons:

- clarity and brievity: hand-coding the computation would be long and error-prone (try it)
- expressiveness: a Java for loop is a statement, while in most cases an expression is required
- efficiency: compilers are able to perform high-level code optimizations and in some cases generate vectorized instructions (vector processors can compute a whole sum in few cycles)

Without the “`||`” operator, the computation is performed sequentially. Reduction expressions are useful even when run sequentially, as they provide a concise and intuitive syntax for many common expressions.

The same notation can also be used to build collections:

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

Figure 14: Parallel reduction (set)

Set or multiset comprehensions, as in the example above, provide the expressive power of SQL queries integrated in the language (in the spirit of LINQ[7] and PLINQ, with different technological choices).

OpenMP also provides parallel reductions, but in the form of preprocessor directives applied to for loops. Unlike Ateji PX, there is no notion of reduction at the language level, only a hint for the compiler to generate parallel code:

```
#pragma omp parallel for reduction(+:sumOfSquares)
for(i=0; i<N; i++)
    sumOfSquares = sumOfSquares + i*i;
```

---

7 http://en.wikipedia.org/wiki/Language_Integrated_Query

Ateji PX constructs for distributed parallelism

While multi-threaded programs on multi-core processors can rely on shared memory for communication between threads, this is not the case for other hardware architectures based on distributed memory such as grids, meshes, and most future-generation parallel processors. When shared memory is not available, parallel branches can communicate by exchanging messages if the program has been written in a message-passing style.

Ateji PX provides message-passing at the language level. This enables the compiler to map distributed programs to various target architectures and write code that is independent of any given library.

Even when considering only multi-core hardware, programs written in message-passing style tend to have less bugs (no data races, no locks), scale better (less traffic on the shared bus), and are ready for the hardware architectures of the future.

With Ateji PX, a source code written in message-passing style will also run without modifications on computer clusters, MPI-based supercomputers, across a network, and in the Cloud. A distributed version of Ateji PX is in preparation, where parallel branches can be run at remote locations. Watch www.ateji.com for announcements.

Message-passing at the language level also enables a simple expression of a wide range of parallel programming paradigms, including data-flow, stream programming, the Actor model, and the MapReduce algorithm.

Sending and receiving messages

Ateji PX provides two syntactic constructs dedicated to message passing:

- Send a message over a channel: **`chan ! value`**
- Receive a message from a channel: **`chan ? value`**

Channels can be synchronous, efficiently implementing rendez-vous synchronization, or buffered. They can be mapped to I/O devices such as files and sockets.

In the example below, two parallel branches communicate through the channel `chan`:

```
// 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);
]
```

Figure 15: Exchanging messages between branches

Data-flow, Streams, Actors and MapReduce

The notion of "process" is typically represented in Ateji PX as a method taking channels as arguments, and whose body is an infinite loop sending and receiving data on these channels.

Here is for instance an adder, that repeatedly reads an integer on each of its input channels, and writes their sum on its output channel:

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

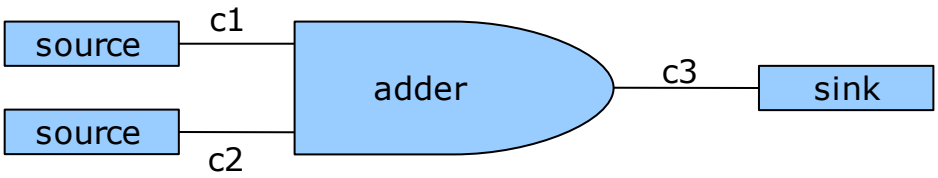
Figure 16: An adder

An adder must be composed with other processes in order to be useful. We need at least two source processes that output values on the input channels of the adder, and a sink that inputs values from the output channel of the adder. The composition requires three channels c1, c2, c3 that must have been declared and instanciated. Here is the general form:

```
[
    || source(c1); // generates values on c1
    || source(c2); // generates values on c2
    || adder(c1, c2, c3);
    || sink(c3);   // read values from c3
]
```

Figure 17: A data-flow program using an adder

Graphically, this parallel composition may be represented as follow:



This simple pattern in Ateji PX implements a number of common parallelism idioms:

- **data-flow** programming, where channels are typically synchronous
- **stream** programming, where channels are typically buffered
- the **actor** model, where each process has a single input "mailbox"
- **MapReduce**, where channels carry tuples

Expressing non-determinism

Note the parallel reads in the code of the adder (figure 16 above):

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

Figure 18: Non-deterministic read

This code reads two values from two channels, *in no particular order*. This is simply impossible to express in a sequential language such as Java, where the programmer is forced to order statements sequentially.

Selecting

Another form of non-determinism is the choice between alternatives. This is implemented in Ateji PX by the `select` statement.

The following code will *either* read a value from chan1 and print it, *or* read a value from chan2 and print it. Once a choice has been made, all other alternatives are discarded:

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

Figure 19: Selecting

This kind of non-deterministic choice is typical of a server accepting connections. It is similar to the Unix select() system call and the Java NIO Selector API.

And more...

You can learn more about the syntax constructs of Ateji PX by downloading the language documentation from www.ateji.com.

Compatibility

On shared-memory architectures such as multi-core chips, the parallel constructs of Ateji PX for Java are compatible with the Java standard libraries for parallelism such as threads, locks and concurrents collections.

Ateji PX parallel constructs are ultimately transformed into calls to a threading (on shared-memory systems) or a message-passing (on distributed systems) run-time. In this sense, threads can be seen as the hardware-level assembly language implementing the high-level parallel constructs suitable as a programming abstraction.

For compatibility purposes, the Ateji PX compiler outputs plain Java source code rather than JVM byte-code (it is still a full compiler, not a simple code generator). This makes it possible to interact with all development tools that require Java source code as input.

The generated source code is standard, readable and maintainable. It is primarily meant to be used by tools, not by humans, but also serves as a secure fallback solution should developers wish to switch back theirs projects to plain Java programming.

Thinking parallel

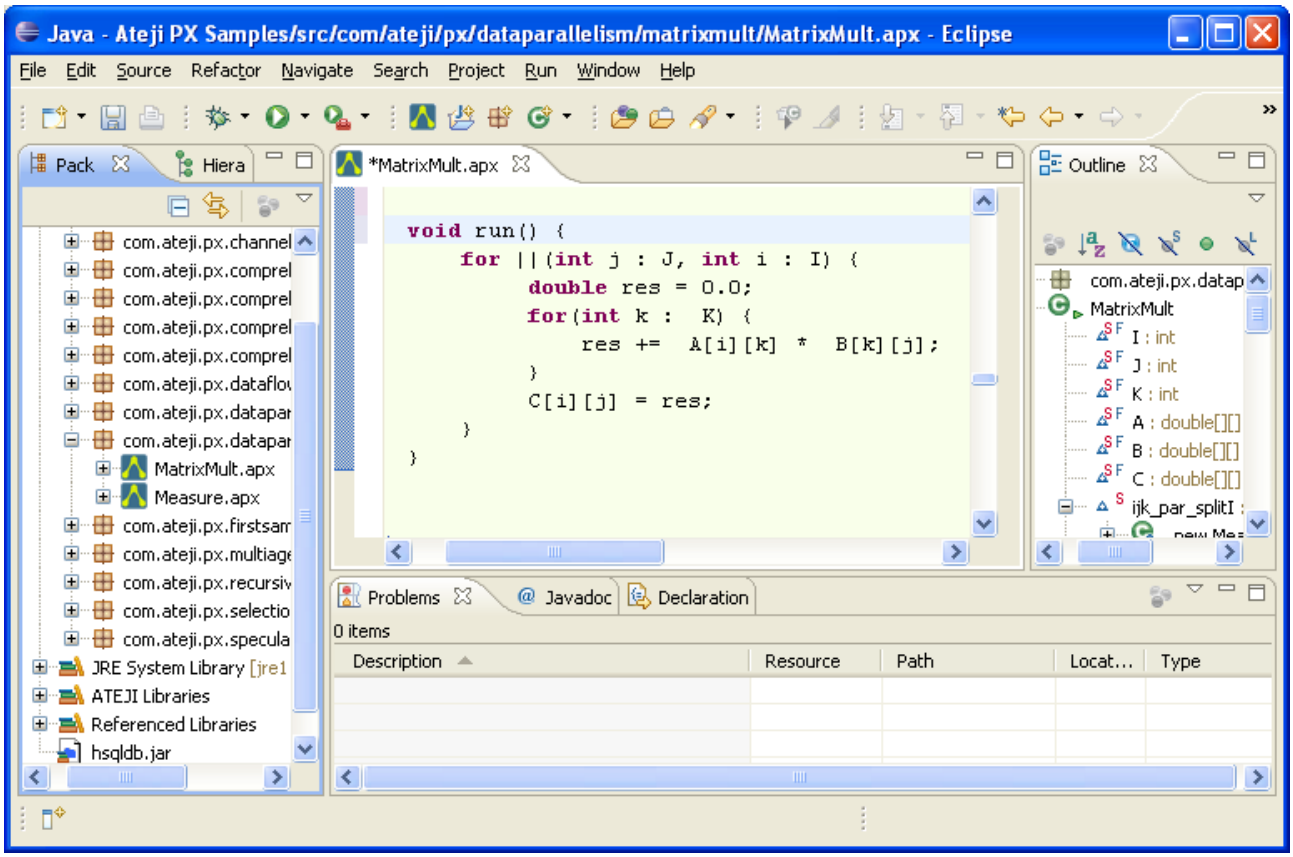
The increase in processor clock rates that drove the increase in processing power since the dawn of computers has come to an end, reaching physical limits related to heat dissipation. Green computing is even driving clock rates down, aiming at reducing power consumption.

In compensation for reduced clock rates, future hardware will become increasingly parallel, quickly reaching hundreds and thousands of cores. Ateji PX provides a smooth transition path from today's sequential programming languages to future parallel languages, likely to be very different and to require a very different way of thinking and designing programs. Code written in Ateji PX is both compatible with today's languages and ready for tomorrow's hardware.

Thinking parallel is not only a matter of improving performance, it is also a way to better understand what the code is actually doing. We have observed on many occasions super-linear speed-ups when parallelizing legacy code, because the introduction of parallelism led to a better understanding and refactoring of the code. Many developers also mention that should they have thought about their code in parallel from start, the code would have been much better written.

Tool support

A programming language without tool support is just a mathematical abstraction. Ateji PX comes with a set of tools integrated as an Eclipse™ plug-in: install the plug-in using the standard procedure, and you can right away transform Java projects into Ateji PX projects or write parallel code from scratch.



Further reading

There's much more to Ateji PX that can be said in a single whitepaper.

- For a hands-on experience, the Ateji PX plugin is available from www.ateji.com
- The plugin comes with an extensive documentation and samples library

Other whitepapers available from www.ateji.com/multicore/whitepapers.html cover various topics related to Ateji PX.

