An ATEJI Whitepaper

Matrix Multiplication with Ateji PX for Java

Simplicity and Performance for Multicore-Enabling your Data-Intensive Operations

by Patrick Viry

```
for(...)   →   for||(...)
```

Summary

Matrix multiplication is a standard benchmark for evaluating the performance of intensive data-parallel operations on recent multi-core processors. We show how to use Ateji® PX for Java™ to achieve state-of-the-art parallel performance, by adding one single operator in your existing code.

intel Software Partner

www.ateji.com

Matrix Multiplication with Ateji PX for Java

Ateji is a specialist of computer language technology that was founded with the aim of making recent scientific developments available to the community of software developers. Ateji's products promote simplicity, efficiency and performance in a world dominated by a multitude of incompatible languages.

Ateji demonstrates, with the design and launch of Ateji PX, how applied language technology provides a simple and effective solution to the multicore crisis, by making parallel programming of multicore processors easily accessible to all application developpers, and by providing a smooth transition path for parallelizing legacy applications.

The purpose of this whitepaper is to evaluate Ateji PX in the context of data-parallel applications, that typically apply the same or similar operations on all or most elements of large arrays or matrices. Matrix multiplication is the standard benchmark for data parallelism and will be the basis of this evaluation.

Ateji PX also handles task parallelism, data-flow and streams, and can target shared-memory (multicore) as well as distributed-memory (GPU, clusters, grids, cloud) architectures. Its main features are:

- built over mainstream programming languages
- compatible with current practices and tools
- suitable for parallelizing legacy applications
- based on a sound theoretical foundation.
- parallelism is made simple and intuitive
- leaves more time for developers to concentrate on performance improvements
- provable correctness helps to avoid many of the errors that plague parallel programs.

Ateji's solution provides a fast track enabling all mainstream application developers to write parallel programs with a minimal learning curve and no changes in tools and pratices. It greatly improves software quality and programmer productivity.

About Matrix Multiplication

Matrix multiplication[1] is a standard performance benchmark representative of massive data parallel operations. This simple example demonstrates the simplicity of parallelizing standard Java code using Ateji Parallel Extension, and the performance impact of various formulations.

Given two matrices A and B, their product C is defined as $C[i,j] = \Sigma_k A[i,k] * B[k,j]$. Namely, computing the value of an element of the result C requires summing each element of the same row in A with each corresponding element of the same column in B. Most algorithms have a complexity of $O(n^3)$, which means that multiplication is very computationally intensive for large matrices.

1 http://en.wikipedia.org/wiki/Matrix_multiplication

Naive Matrix Multiplication Algorithm

The naive multiplication algorithm in Java has three simple nested for loops ranging over indices i,j,k in this order:

```
for(int i=0; i<I; i++) {
      for(int j=0; j<J; j++) {
            for(int k=0; k<K; k++) {
                  C[i][j] += A[i][k] * B[k][j];
            }
      }
}
```

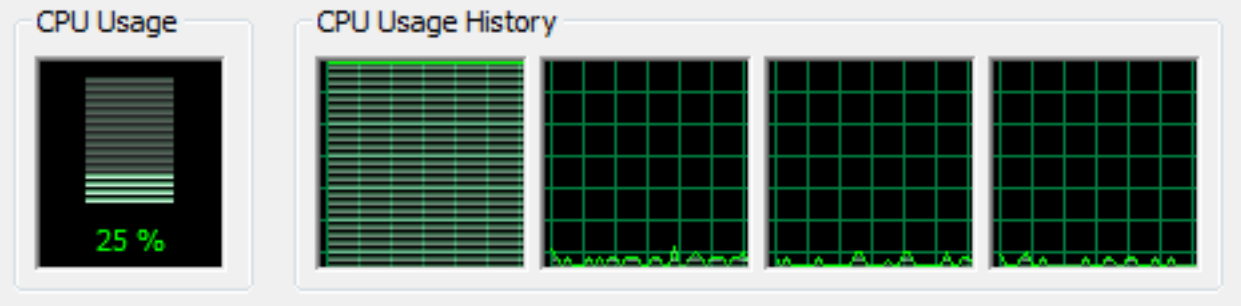
Figure 1: Sequential code - Naive Matrix Multiplication in Java

Before even starting to parallelize our code, Ateji PX makes it possible to remove some sequential dependencies. The code below (Figure 2) is strictly equivalent to the previous Java code (Figure 1) in terms of semantics and efficiency, but removes the sequential dependencies inherent to the C-like expression of for loops:

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

Figure 2: Sequential code - Naive Matrix Multiplication with Ateji PX

This code is inherently sequential. Running it on a 4-core PC, we see in the task manager that only one core is used.



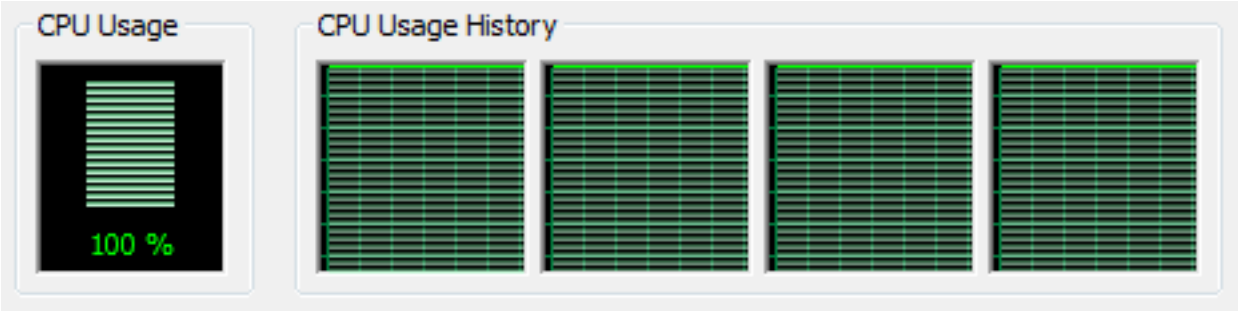
Parallelizing the Naive Matrix Multiplication Algorithm

With Ateji PX, a for loop is parallelized by simply inserting a parallel operator ("||") immediately after the for keyword. Its effect is to split the corresponding loop in parallel over all available cores. The code in Figure 3 below is the same code as Figure 2 above, with the outermost for loop parallelized :

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

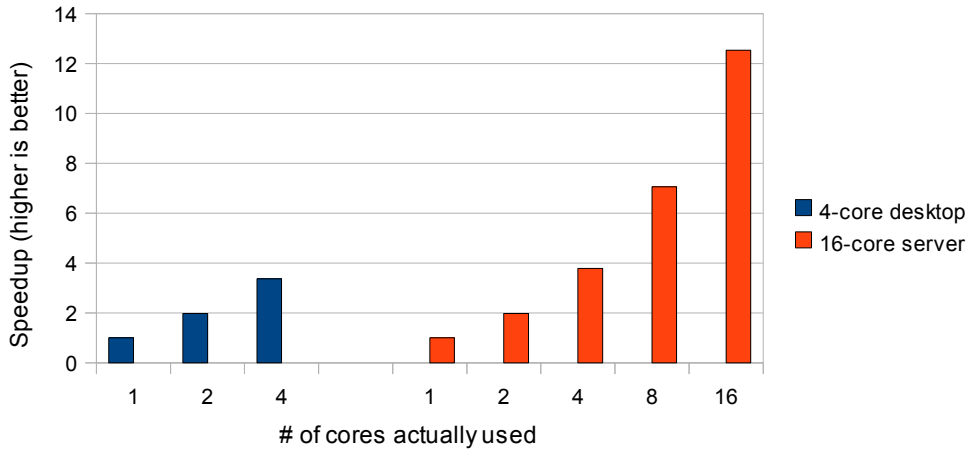
Figure 3: Parallel code - Naive Matrix Multiplication with Ateji PX

Running this parallel code on the same 4-core PC, we see in the task manager that all cores are used.



Speedup

With the single addition of a parallel "||" operator, we measured a speedup of 3.4x on a 4-core desktop PC and 12.5x on a 16-core server:



Comparison with Java threads

Obtaining these performance improvements with Ateji PX could hardly be simpler: starting from a standard Java source code, we added a parallel operator to the outermost for loop, after simplifying the expression of the iterator.

Here is the same naive parallel matrix multiplication algorithm written using the Java threads library:

```
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);
      }
}
```

Figure 4: Parallel code - Naive Matrix Multiplication with Java threads

This code is significantly more verbose compared to the one in Figure 3. This wouldn't matter much if code was written once and for all, but this is rarely the case. Here, the two major concepts present in this code (matrix multiplication and parallelism) are hidden behind a lot of syntactic noise, which makes maintaince and quality control a pain.

Measured performance of the threaded code (Figure 4) is similar to that of the naive implementation using Ateji PX (Figure 3). But we'll see in the next section that **getting the best performance requires a lot of experimentation**, such as trying different orderings of the for loops, or divising different schemes for splitting the work. Because these experiments require a heavy rewriting of the threaded code (try exchanging the I and J loops!), a developper is unlikely to try them out, and thus to find the best combination for performance.

In other words, given the same time and effort for parallel code development, **code based on Ateji PX tends to be much more performant at runtime than code based on the threads library,** because developpers have more time available for performance tuning and can easily try different parallelization strategies.

Raw Performance – Improving over the Naive Algorithm

While the previous code samples demonstrate a pretty good speedup on multicore processors, raw performance is deceptive. The reason is simple: we are parallelizing a slow algorithm.

The sequential version of the naive matrix multiplication algorithm is far from optimal, for two major reasons:

- The ordering of loops has a tremendous impact on cache locality – you can try it by yourself by permutating the three for loops over I, J and K.
- The matrix element A[i][k] is read j times from memory (similarly for B[k][j]) – working on a local copy of each row or column also improves cache locality

The fast sequential algorithm taking these points into account is more verbose than the naive version, but is about 5x to 10x faster. This is more or less the speedup achivable on an 8-core processor, and no to be neglected.

In the code below, the order of loops has been changed from I,J,K to J,I,K, and for each j the whole column B[k][j] is cached in a local array:

```
for(int j : J) {
      double[] bcolj = new double[K];
      for(int k : K) {
            bcolj[k] = B[k][j];
      }
      for(int i : I) {
            double[] arowi = A[i];
            double sum = 0.0;
            for(int k : K) {
                  sum += arowi[k] * bcolj[k];
            }
            C[i][j] = sum;
      }
}
```

Figure 5: Sequential code - Improved Matrix Multiplication with Ateji PX

Like before, the algorithm is parallelized by adding a "`||`" operator right after the outermost for keyword:

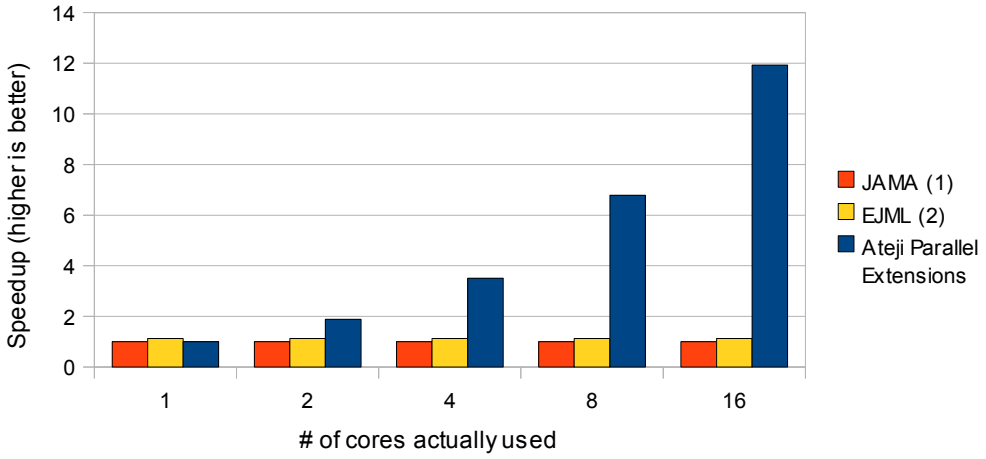
```
for||(int j : J)  ...
```

Figure 6: Parallel code - Improved Matrix Multiplication with Ateji PX

This improved algorithm has the same scalability properties than the naive implementation, but now the raw performance is on par with standard linear algebra librairies. The chart below compares this algorithm on a 16-core server with two representative Java-based libraries providing matrix multiplication, JAMA[1] and EJML[2].

1 JAMA (http://math.nist.gov/javanumerics/jama/) is a basic linear algebra package for Java. It provides user-level classes for constructing and manipulating real, dense matrices.

2 EJML "an Efficient Java Matrix Library" (http://code.google.com/p/efficient-java-matrix-library/) is a linear algebra library for manipulating dense matrices.



Our hand-written code based on Ateji PX is on par with both libraries when comparing raw sequential performance, and also scales well when using multiple cores. The libraries have not been designed for parallelism and cannot benefit from multiple cores.

This example also stresses the problem of using legacy code with multicore processors. Without a **simple** way to parallelize existing code – such as inserting a "`|||`" operator with Ateji PX – , most existing libraries and components won't benefit from multiple cores.

What about inner loops ?

You've probably noticed that we've parallelized only the outer loop. What about the inner loops?

A "`|||`" operator tells the Ateji PX compiler that it can split the computation into a number of parallel *branches* that can be run in parallel. Branches are light-weight objects scheduled by the runtime.

For good performance speedups on multi-core processors, it is important to keep all cores busy. In other words, to have more branches than available cores. This is typically the case when the cores count is in the 10's or the 100's.

Having a little more branches then cores helps the scheduler keeping cores busy, especially when different branches may have very different computation times.

But having a much larger number of branches than of cores does not improve performance further. On the contrary, creating and scheduling a branch takes a small amount of time, that becomes noticeable if the branch count becomes huge. This is why, in some cases, parallelizing inner loops may actually degrade performance.

For better performance and simplicity, the Ateji PX compiler tries by default to create an optimal number of branches, independently of the size of the loop. There are ways to precisely tune the number of branches, but it is usually enough to remember that inner loops do not require a parallel operator.

More about the "||" operator.

The "||" operator used in a for loop is only a special case of a much more general pattern. It actually introduces parallel branches within a parallel block. This code runs a() and b() in parallel:

```
[
	|| a();
	|| b();
]
```

Figure 7: An example of parallel branches

Branches can also be quantified. The code below runs five copies of a(i) in parallel, one for each value of i between 0 and 4, plus one copy of b():

```
[
	||(int i : 5) a(i);
	|| b();
]
```

Figure 8: An example of quantified branches

The "||" operator used in a for loop is actually a shorthand for a parallel block consisting of one single quantified branch.

Introducing branches in a program expresses how the program can be split in parallel. This doesn't mean that branches actually *have* to run in parallel, but that we're introducing a parallel semantic for the program. The runtime will then do its best to schedule branches in order to make the best out of the given hardware architecture.

More about Ateji PX

Matrix multiplication is a standard example for data parallelism, where the same operation is applied to all elements of a large array or matrix. Ateji PX also handles with a similar simple and intuitive syntax many different aspects of parallel programming:

- parallel reductions
- task and recursive parallelism
- distributed parallelism
- message passing
- non-deterministic choice (select)

Future releases will include correctness proofs for avoiding the introduction of parallelism-related bugs, and support for distributed hardware architectures such as clusters and system-on-a-chip (SoC).

Documentation and evaluation licenses are available from www.ateji.com

Benchmarking specifications

All measurements presented in this whitepaper were made under the following conditions:

- Ateji PX Technology Preview 2 (build 1.0.29)
- All matrix sizes are 1000 x 1000 elements
- The default JVM arguments are used
- Performance is measured by wall-clock time, as given by `System.getCurrentTimeMillis()`
- We consider the average of three measures after dropping the first one (in order to allow for JVM warming-up)

We considered two hardware architectures:

4-core desktop:
1 Intel® Core 2 Quad® at 3.00GHz
Java(TM) SE Runtime Environment (build 1.6.0_18-b07)
Java HotSpot(TM) 64-Bit Server VM (build 16.0-b13, mixed mode)

16-core server:
4 Intel® Xeon® quad-core E7310 Tigerton Processors at 1.6 GHz
Java(TM) SE Runtime Environment (build 1.6.0_11-b03)
Java HotSpot(TM) 64-Bit Server VM (build 11.0-b16, mixed mode)

In the figures, the " number of cores actually used " is the value of the parameter given to the call `BranchScheduler.init(nbCores)`, which specifies to the scheduler the maximum number of threads it is allowed to create.

