



## White paper

## Executive Summary

Multi-core machines, developed to prolong Moore's Law without the negative side effects caused by increased heat generation, are becoming the norm. All the major chip manufacturers are shipping multi-core processors, which can now be seen on computers ranging from low-end laptops to high-end servers.

The bad news is that multi-core machines running on legacy code do not automatically deliver any performance improvements. Exploiting the additional computing power requires code to be split into separate threads which can run in parallel. Software developers face complex issues that are unique to parallel programming such as deadlocks, race conditions, synchronization and load balancing. And today, few companies have the required skills and experience to tackle these challenges in house.

Jibu is a cross-platform suite of libraries developed by Axon7 which offers powerful and sophisticated constructs for concurrent and parallel programming. The simplicity of Jibu enables already skilled programmers with no previous experience in writing concurrent or parallel programs to learn the basics of Jibu in a matter of hours.

Jibu provides a uniform high-level approach to parallel programming in Microsoft .NET, Java, C++ and Delphi which abstracts low-level threading and synchronization primitives. Almost any concurrent program can be modeled using the predefined parallel methods, tasks, channels, mailboxes and choices. Testing by Axon7 has demonstrated as high as 3.7 speed up improvement on an Intel Quad core machine.

Companies across the globe have been aware of the disruption caused by multi-core machines and are now confronted with a dilemma: invest in adapting legacy code to exploit multi-core machines, with all the ensuing benefits of differentiation, but risk doing so using the 'wrong' tools and platform; or wait until standards emerge but risk obsolescence vs. the competition.

Deploying Jibu will ensure near seamless adaptation of legacy code, save time and costs, and enable product differentiation by ensuring speedy release of multi-core compatible solutions which exploit the enormous performance potential of multi-core machines.

## Background

Multi-core machines are fast becoming the norm.

Moore's Law, which states that the number of devices integrated on a chip of fixed area doubles every 12-18 months, has held for a remarkably long time but the laws of physics are starting to get in the way. By 2001, clock speeds had reached 2GHz. Extrapolating Moore's Law to 2007 should have yielded speeds of 16 to 31 GHz. However, today the fastest mainstream microprocessors only run at around 3 GHz because increasingly faster clock speeds are generating unmanageable amounts of heat.

As a workaround, manufacturers of integrated circuits have developed chips on which two or more processor-cores are integrated on the same die of silicon. Adding more cores instead of increasing the clock frequency not only improves performance, but reduces power consumption and heat dissipation.

By now all the major chip manufacturers are shipping multi-core processors. The current generation of processors includes Opteron/Phenom from AMD, Core 2/Xeon from Intel, Niagara from Sun and the Cell processor from IBM, all of which feature between two and nine cores on a single chip. Soon, single core machines will be a rarity – these already count for as few as 10% of Intel's shipments. The trend toward multi-core processors is seen across all computing segments from low-end laptops to high-end servers.

For many years software developers working in fields where massive computing power is essential (meteorology, financial forecasting, digital signal processing, geological simulation, optimization, scientific computing) have worked with computers using multiple processors or cores. The average programmer however is neither experienced nor trained in writing programs which are optimized multi-core computers.

## The Challenge

Most applications today consist of only one thread of execution which presents a major problem since single-threaded programs don't exploit all the extra power added by multi-core processors. Benefiting from multi-core machines requires code to be split into separate threads able to execute in parallel - an exercise that is far from trivial.

Parallel programming suffers from the same pitfalls as normal sequential programming but also add new ones. Complex issues such as deadlocks, race conditions, synchronization and load balancing are all problems that are unique to parallel programming. These are issues that very few programmers have any experience in dealing with.

The greater complexity of parallel programming increases development time and leads to more error-prone programs which are harder to debug and maintain. The fact is that parallel and concurrent programming experts are few and far between.

Software developers must write efficient and scalable applications which are capable of exploiting the processing power of upcoming generations of multi-core machines as well (Intel has already demonstrated an 80-core processor!)

Solving the multi-core challenge independently is an option only for very large companies which have the resources and in-house expertise (the classical example being Google). Other companies are, in many cases, in a waiting pattern expecting their suppliers to solve the problem.

The hard fact is that multi-core machines will not improve performance by themselves. Legacy code cannot be expected to run faster and will have to be re-written, or at the very least adapted.

Those companies which take on the multi-core challenge now stand to gain significant competitive advantages.

## Jibu

Jibu is a library for concurrent and parallel programming offering a few very powerful and sophisticated constructs. The simplicity of Jibu enables already skilled programmers with no previous experience in writing concurrent or parallel programs to learn the basics of Jibu (parallel programming) in a matter of hours.

Gone are the days of low-level threading and synchronization using traditional threading construct like locks, monitors, semaphores, mutexes, condition variables, synchronized methods and gone are the days where every new programming language required relearning existing skills, as illustrated below.

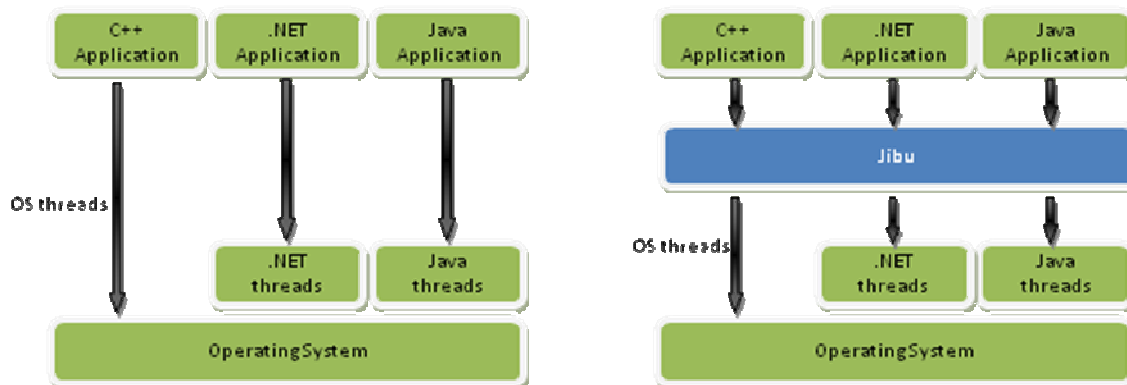


Figure 1: Standard concurrent programming vs. concurrent programming with Jibu.

Jibu provides a uniform high-level approach to parallel programming in Microsoft .NET, Java, C++ and Delphi which abstracts low-level threading and synchronization primitives.

Benefits include:

- Predefined parallel methods, such as a parallel for loop and parallel reduction
- Task-based concurrency instead of thread-based concurrency
- CSP-based communication and synchronization, such as channels, mailboxes and choices
- Efficient and scalable task scheduling

Almost any concurrent program can be modeled using the predefined parallel methods, tasks, channels, mailboxes and choices.

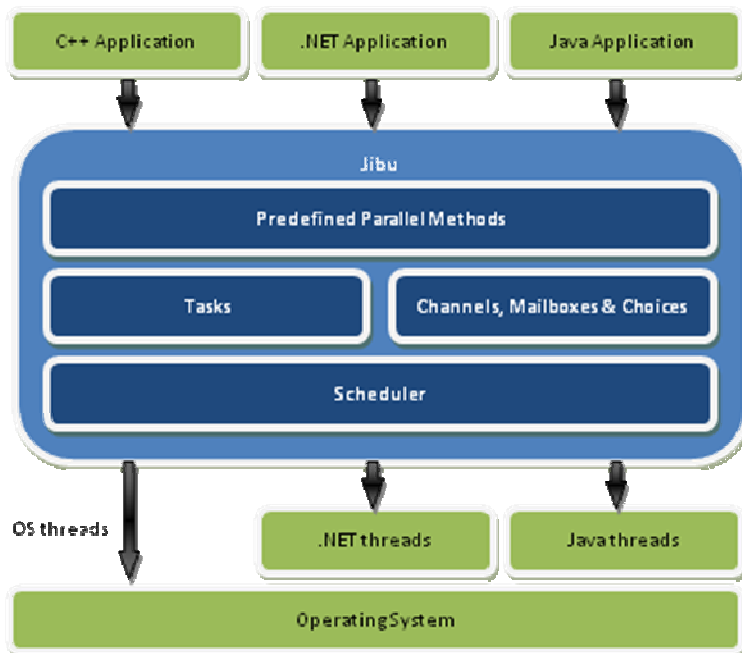


Figure 2: Jibu overview.

The predefined parallel methods make it extremely simple to execute for loops, foreach loops and reductions (all frequently used in most sequential programs) in parallel, thus they provide a fast and easy way of introducing concurrency into existing sequential programs.

Tasks replace threads as the central concept in Jibu. Tasks are easy and fast to create and offer simple mechanisms for cancellation and safe inter-task communication via mailboxes. Tasks are lightweight objects compared to threads and millions of tasks can easily be created and run. Jibu automatically maps tasks onto threads ensuring optimal utilization of the available CPU resources. This enables the developer to focus on high-level logical tasks without having to efficiently map the tasks to low-level threads.

Channels and mailboxes abstract low-level communication and synchronization mechanisms and provide a simple way for multiple tasks to exchange data, without using traditional constructs like locks, mutexes, semaphores, critical regions or conditions variables.

Choices enable high-level event coordination supporting both prioritized and fair selection of events.

Combine these constructs with the advanced Jibu Runtime Infrastructure (JRI) and we are left with a platform for concurrent and parallel programming that is as easy to use as it is powerful.

The following code snippets demonstrate how to introduce concurrency into an existing sequential program using the parallel for loop. A sequential method calculating the kernel matrix for a given dataset of vectors is illustrated below.

```
// Calculate kernel matrix.
public double[,] SeqKernelMatrix(List<double[]> dataSet)
{
    int size = dataSet.Count;
    double[,] kernelMatrix = new double[size, size];

    for(int i = 0; i < dataSet.Count; i++)
        for(int j = 0; j < i; j++)
            kernelMatrix[i,j] = PolyKernel(dataSet[i], dataSet[j], 2);

    return kernelMatrix;
}
```

Using Jibu's parallel For method makes it remarkably simple to make a parallel version of the kernel matrix method, as shown below.

```
// Calculate kernel matrix
public double[,] ParKernelMatrix(List<double[]> dataSet)
{
    int size = dataSet.Count;
    double[,] kernelMatrix = new double[size, size];

    Parallel.For(0, dataSet.Count, delegate(int i)
    {
        for(int j = 0; j < i; j++)
            kernelMatrix[i, j] = PolyKernel(dataSet[i], dataSet[j], 2);
    });

    return kernelMatrix;
}
```

The sequential and parallel versions are similar but the outer for loop in the sequential version is replaced by a call to the For method in the parallel version. Running this simple parallel implementation with a dataset of 5000 vectors, each containing 100 floating point numbers, on an Intel Quad core machine resulted in a 3.7 speed up.

This example will scale seamlessly to machines containing many more than four cores because the For method automatically creates 5000 tasks which are efficiently executed by Jibu's built-in scheduler.

## Tasks vs. threads

With normal thread programming the smallest unit of work able to run concurrently is a thread. If the task to be solved naturally divides into many independent units of concurrent work (as the example in the previous section) an operating system thread needs to be created for each unit.

In many real-world problems and algorithms this would mean that anywhere between a few and a few million threads would be needed. This is often impossible due to the threads memory requirements (the minimum memory requirement for a thread in .NET is 128 KB). Even if it was possible from a memory standpoint the time required for the operating system to context switch between thousands of threads would completely ruin performance.

The rule of thumb in thread programming is to keep the number of threads as close to the number of cores as possible. In order to enforce that rule of thumb many problems need to be restructured so that each thread handles a much bigger piece of work. This restructuring is not trivial and in the process another problem is introduced: how to ensure that each thread does an equal amount of work.

Picture this: A program is running on a quad core computer. The problem can potentially be divided into 1000 independent units of work but to enforce the one-thread-per-core rule we divide the program into only four concurrently running parts. Unless we can be absolutely sure that all four parts do exactly the same amount of work we have just introduced another performance problem. If the workload is distributed unevenly across the threads we will end up in a situation where one or more cores will sit idle waiting for the remaining cores to finish the work.

What is needed is a combination of the two models just described. We want to be able to use a fine-grained approach to concurrency where thousands of independent units of work are able to run concurrently. At the same time the rule that the number of working threads should reflect the number of cores in the system should be enforced.

To achieve that we need to separate the units of work from the operating system threads and avoid a one-to-one relationship between units of work and threads.

In Jibu the independent units of work are tasks and the component that maps tasks to threads is the task scheduler.

## The Scheduler

The task scheduler is basically a work-stealing algorithm that continuously monitors the number of tasks lined up for execution, the number of threads currently executing tasks, the number of blocked tasks and the number of available cores in the system.

All the information is analyzed and the scheduler steps in when necessary to ensure a balanced workload, an optimum number of threads and a responsive program. The Jibu scheduler works in conjunction with the Jibu thread pool to ensure that threads are started when needed and stopped when they are no longer required.

Adding thousands of tasks must necessarily add some overhead but, as illustrated in figure 3, the overhead of adding tasks is very limited in Jibu.

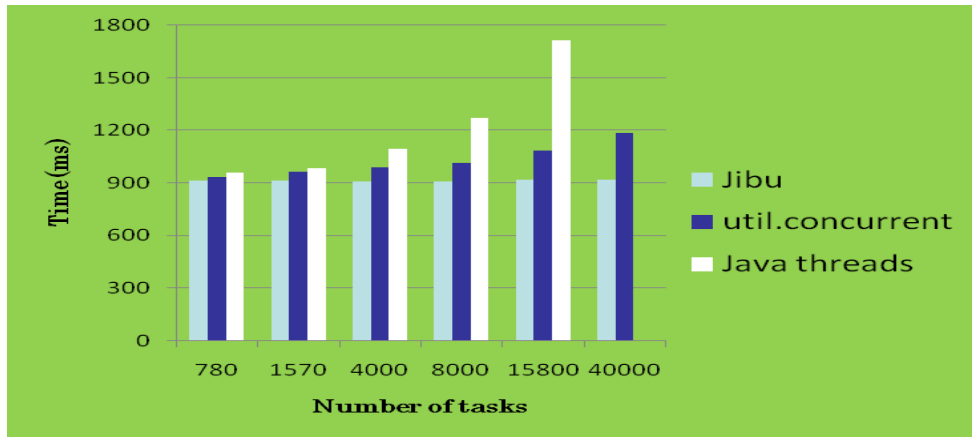


Figure 3: Running times for three simple parallel Divide & Conquer implementations of Quicksort, running on an Intel Quad core machine.

Figure 3 shows the running times of three simple parallel Divide & Conquer implementations of Quicksort, using standard Java threads, util.concurrent (part of the Java class library) and Jibu.

It is important to remember that the Jibu task scheduler is not preemptive like the thread scheduler in Windows. That means that a task that has been scheduled for execution by the task scheduler will never be interrupted even though other tasks are pending execution. As there is no fixed time-slice for a task to run before being preempted the scheduling performed by the Jibu task scheduler is inherently unfair in nature. If fairness matters or if tasks are created that are never meant to terminate it is possible to run tasks directly as threads, bypassing the task scheduler completely – in most real world problems however the task scheduler works fine.

The benefits of the scheduler are many but first and foremost it enables developers to write simple, efficient and scalable concurrent programs that dynamically adapts to the available CPU resources.

## Conclusion

Companies across the globe have been aware of the disruption caused by multi-core machines and are now confronted with a dilemma: invest in adapting legacy code to exploit multi-core machines, with all the ensuing benefits of differentiation, but risk doing so using the ‘wrong’ tools and platform; or wait until standards emerge but risk obsolescence vs. the competition.

Deploying Jibu will ensure near seamless adaptation of legacy code, save time and costs, and enable rapid differentiation by ensuring rapid release of multi-core compatible solutions which exploit the enormous performance of multi-core machines.

Further information and contact details, visit [www.axon7.com](http://www.axon7.com)

©Axon7 – March 2008