# PVTM Architectural White Paper

## Introduction

PVTM is an acronym for Pontus Vision Thread Manager. PVTM accelerates software by optimizing the execution layout of threads on a server, and often doing so on less hardware.

PVTM improves the performance of a variety of applications, from trading systems to Relational Databases and data analytics.

In addition to this document, readers can get additional information on PVTM by viewing the following two videos:

http://www.pontusvision.com/thread-manager-threadmanager/threadmanager-powerpoint-video/

http://www.pontusvision.com/thread-manager-threadmanager/pvtm-thread-manager-gui-quick-tour/

The White Paper is sub-divided into the following sections:

**Introduction**

**Abstract**

**What is the logic behind PVTM?**

**How much delay in a single server?**

**Threads vs Processes**

**In-house vs Third Party applications**

**How does it work?**

**Benchmarks**

**Conclusion**

# Abstract

We have adapted resource scheduling techniques used to optimize seating arrangements in trains, airplanes, and even offices, and adapted them to computer systems. Rather than arranging the most efficient arrangement for office workers in teams to best communicate with each other, we are arranging the most efficient layout of threads. We move the threads that communicate with each other so they can sit on the closest hardware cores to reduce communication overheads.

An automated way of speeding up software applications has long been a holy grail of computing and information technology, but there has usually been a caveat or two. An example of this is the promised of 10X, 100X or 1000X performance boost by porting your application to a GPGPU, or Intel Xeon Phi. The caveats here are:

a) You have to port the application to a new hardware architecture
b) You have to vectorise the application to take advantage of the inherent parallelism of the new computational unit
c) You must have access to the source code of the application
d) Software engineers who can vectorise code efficiently are rare beasts indeed.

Given the points, a) – d) above, the number of such applications which have been successfully ported to, either GPGPU or Xeon Phi and are in a production environment is tiny compared to the effort exerted in the attempt.

PVTM speeds up software applications by better placing software threads onto hardware cores to maximise performance. A secondary advantage, (which is arguably as important), is that it maximises hardware utilization too. PVTM does all this without any of the caveats above.

# What is the logic behind PVTM?

Modern operating systems (OSs) on modern NUMA (non-uniform memory access) servers are not very good at managing threads for performance-sensitive apps. OSs typically balance the load across various cores rather than focus on application performance. When dealing with performance-sensitive applications, balancing the load across various cores causes application latency to increase significantly.

As an example, in Figure 1 below, there are two Four-CPU servers running the same workload. This hypothetical example shows a pipeline with seven steps. The OS on the left server distributes the red threads across all CPUs in a round robin fashion; by doing this, the distances for data movement are much greater. Therefore, the time to move data between the threads is several times greater than in the server on the right. For many applications, constraining the threads to fewer CPUs can significantly increase performance.

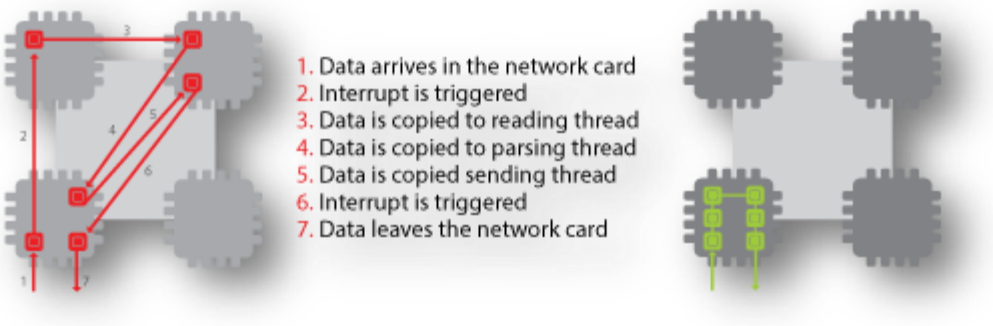



**Figure 1 - Scheduler behaviour with and without PVTM**

PVTM improves the probability that software threads will use less time to communicate with each other. PVTM places threads that have high levels of communication as close to each other as possible whilst at the same time avoiding context switches. To achieve this, PVTM captures the following information about the system:

- Firstly, PVTM analyses the target hardware layout figuring out which cores are closest to each other.
- Secondly, it analyses the communication patterns between threads including which threads use sockets, shared memory and locks to talk to each other.
- Thirdly, PVTM captures the CPU utilization and current location of each thread, as well as optionally the position of I/O devices, such as disks and network cards.

PVTM then sends all this information to a simulator that applies a performance score to the system. The score penalizes threads that have strong communication links to each other, but that are located in cores that have long relative distances to each other. The score also penalizes context switches by avoiding moving active threads to the same core as other active threads as much as possible. The simulator is then capable of running millions of what-if analysis to determine a thread execution layout that improves the score as much as possible.

# How much delay in a single server?

Most people think that the delays inside of a single server are negligible; however, the cumulative effect of these delays is very large. The speed at which threads communicate with each other can be over one hundred (100) times slower depending on which cores they are running in a single server. The relative distance between the CPU cores and their memory access affects how quickly data can move between threads.

The following DISTANCE RATIO TABLE shows rough relative speeds of sending data between threads measured on a four CPU server. This table reflects the access speed of memory in different cores, as well as the amount of time that it takes to re-fetch data from main memory if not cached. Access speed is relatively fast when the CPU accesses data stored in level 1, level 2 and level 3 caches. The access speed is slower when the CPU accesses local memory, and even slower when the CPU access remote memory from a neighbouring CPU.

DISTANCE RATIO TABLE

| Level 1 Cache | Level 2 Cache | Level 3 Cache | Local Memory | Remote Memory | No-Cross Bridge Memory |
|---|---|---|---|---|---|
| 1 | 2 | 6 | 10 | 83 | 113 |

It is important to notice that these speeds can vary dramatically depending on the thread behaviour. As such, applications will seldom always run at the fastest speed or at the lowest speed. As the Intel Haswell stats below show, the time to access memory can vary from 4 cycles to access L1 to over 45 cycles plus around 57ns to access main memory. If a thread never moved, and always accessed memory that was within the TLB and in L1 cache, it would use around 4-5 cycles to access the data. In contrast, a thread constantly being moved across cores (in the same CPU) would be missing L1/L2 cached data, having to go to L3 or main memory and having its TLB trashed would use 36 + 9 + 57ns to access the data. In reality, threads will never behave as perfectly as the 4-5 cycles 100% of the time; however, when threads are not pinned, they tend to be constantly in the misbehaved state. By pinning threads in place, PVTM is trying to increase the chances of the good 4-5 cycle behaviour.

Here are some stats for Intel's Haswell processor:

- L1 Data Cache Latency = 4 cycles for simple access via pointer
- L1 Data Cache Latency = 5 cycles for access with complex address calculation (size_t n, *p; n = p[n]).
- L2 Cache Latency = 12 cycles
- L3 Cache Latency = 36 cycles
- RAM Latency = 36 cycles + ~ 57 ns
- L1 Data TLB - miss penalty of 1-8 Cycles , with 4x1Gb/ 32x2Mb / 64x4Kb entries
- L2 Data TLB - miss penalty of 9 -22 cycles with 1024 2Mb / 1024 x 4Kb entries

# Threads vs Processes

Note that up to this point, we have not mentioned processes, but rather threads. It is easy to confuse the two terms, process and thread. As the name implies, PVTM works at a thread level, and largely does not really care much about the process boundaries. **PVTM can equally optimize single-threaded processes, or multi-threaded processes, or any combinations thereof**.

In most modern operating systems, the main distinction between a few single-threaded processes and a multi-threaded process is their rights to access memory. Threads within a single process can access the same memory area directly, whereas threads that reside in different processes have to make special calls to use shared memory. Many single-threaded applications use shared memory to communicate; similarly, many multi-threaded applications use other inter-process communication (IPC) mechanisms such as sockets or named pipes between their threads. The reasons and merits of using single-threaded architectures vs multi-threaded architectures and their IPC mechanisms vary dramatically, and it is beyond the scope of this document to discuss them.

PVTM's model makes no distinction between two threads that belong to the same process and decide to use futex locks to lock a shared memory area, and two threads that belong to different processes, and use futex locks to lock a shared memory area. As long as there is a strong communication pattern between the threads, we do not really care to which process they belong.

# In-house vs Third Party apps

PVTM works with in-house, third party apps, brand new apps and legacy apps equally. PVTM is also agnostic to which computer language is used. PVTM captures low-level system calls to determine which threads communicate with each other. It behaves like a high-performance profiler that captures basic system calls with little overhead to the system. Because these calls are at quite a low-level, PVTM can easily figure out inter-thread patterns between threads in languages as diverse as C, C++, Java, R, Perl, C#, Python, and any other higher level language, as long as the applications are dynamically-linked against the system calls.

# How does it work?

As seen in Figure 2 below, PONTUS VISION Thread Manager (PVTM) has 3 main components:

1) PVTM Agent – (pvtm-agent) – A lightweight single threaded agent that collects information about the hardware, and discovers the data communication patterns between threads. PVTM Agent is written in C, and usually uses < 1% CPU to capture its data. To aid PVTM Agent discover the thread pinning strategies, the following components may also be used:
   a) libpvtm-agent-preload.so – On Linux, a library that can help the PVTM Agent discover communication patterns between threads. This can be injected in existing applications without recompiling them by using the LD_PRELOAD environment variable.
   b) pvtm-agent.jar – an optional java agent file that exposes the names of Java threads to the operating system. To use this, you need to change your JVM command line to add the `-javaagent:<path to the pvtm-agent.jar file>`.
   c) pvtm-agent-preload-windows.dll – On Windows, a library that PVTM Agent discovers communication patterns between threads. PVTM Agent automatically connects to the

applications that need to be monitored using this DLL along with remote debugging techniques

2) PVTM Simulator / Thread Manager – (run-threadmgr.sh) – a Java 7 standalone simulation engine that receives TCP/IP connections from PVTM Agent, and can take the hardware information, as well as the data communication patterns between the threads to produce an optimal layout of software threads on the hardware cores.
3) PVTM GUI Server (run-server.sh) – a self-contained server that hosts a browser-based graphical user interface. The GUI enables users to visualize the layout of the threads, and produce scripts for static thread pinning configurations. Users can use this in environments where PVTM Agent is unable to run.

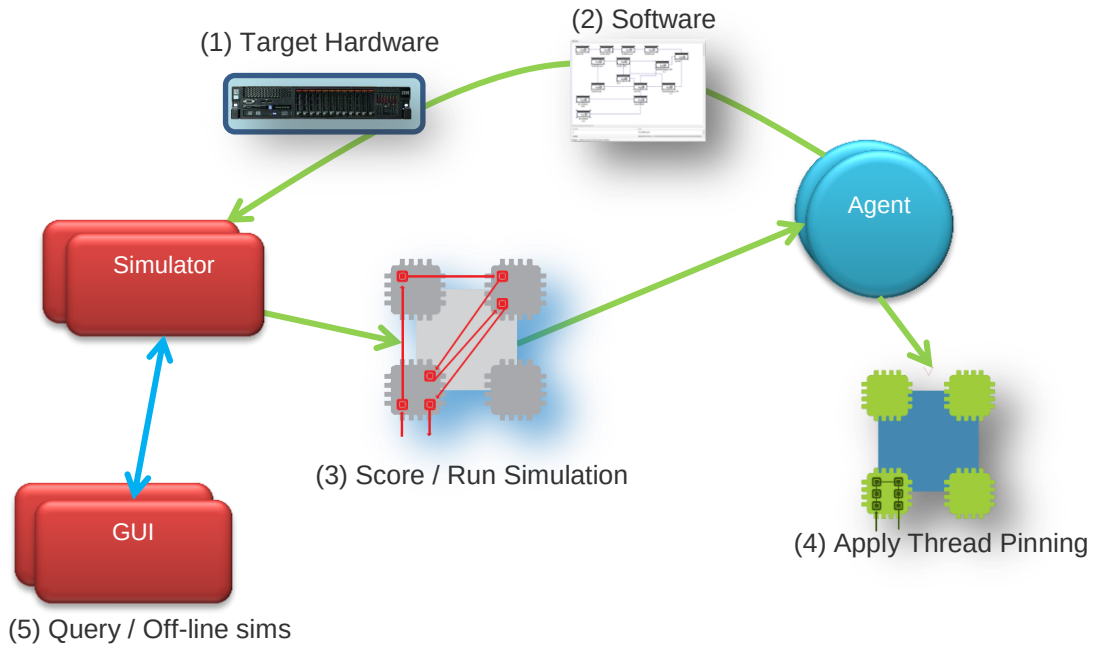



**Figure 2 - PVTM Architecture**

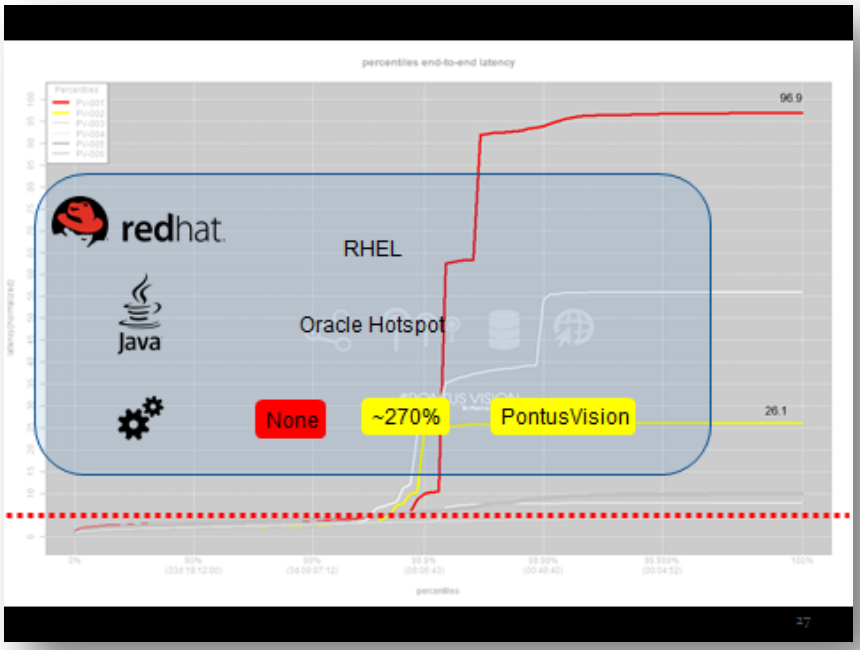
# Benchmarks

### FX Trading platform

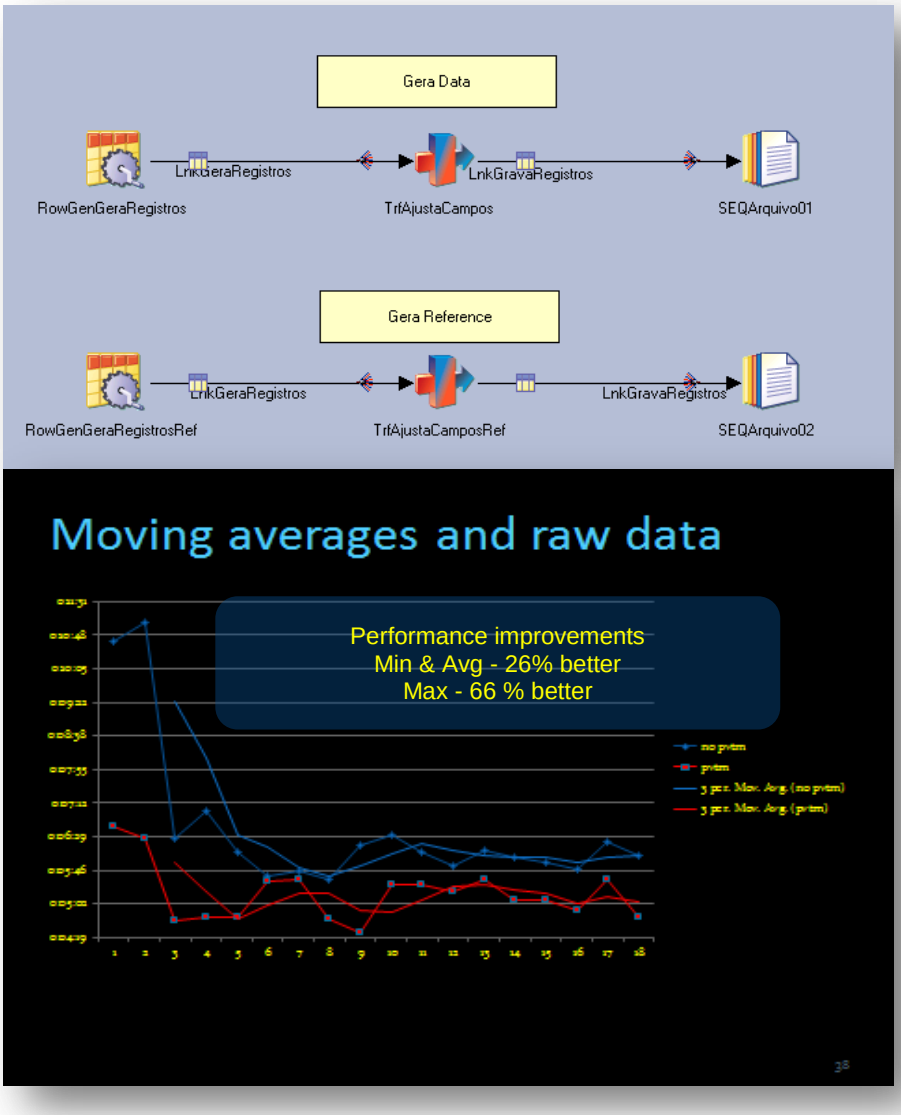
A major investment bank was co-locating an FX trading platform with their liquidity venues.

The system comprised a mix of C, C++ and Java components, with some developed in-house, and others by third-party vendors. PVTM collapsed apps running on 10 servers down to one, delivering a **50% CapEx reduction**.

PVTM also **improved the platform's latency by 270%**.



### IBM DataStage, ETL & MDM tool




IBM's DataStage is a closed-source extract, transfer, load (ETL) and master data management (MDM) analytics tool.

The use case shows a retail bank that has thousands of databases with duplicated customer names that need to be de-duplicated daily.

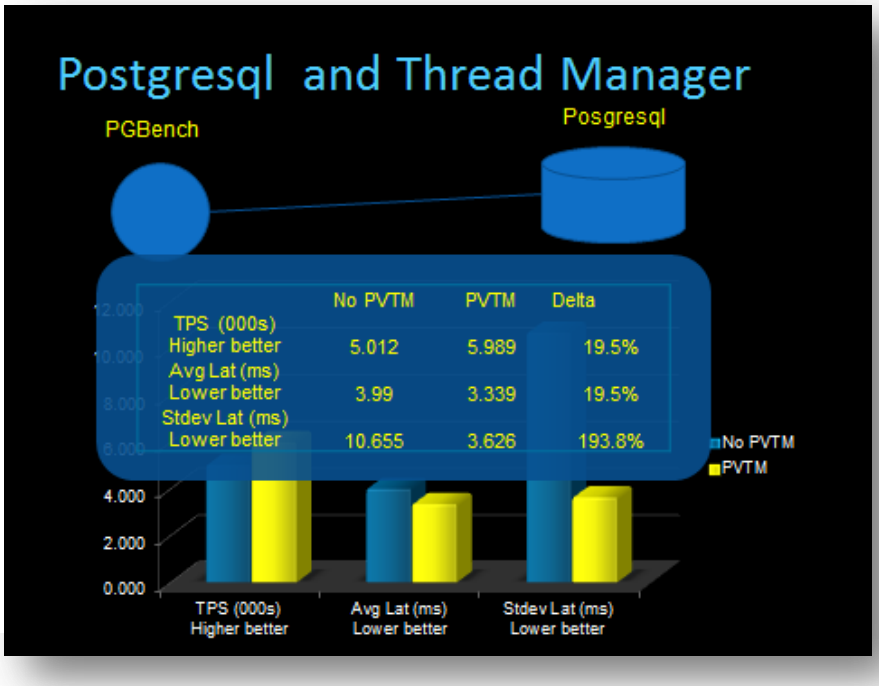
This normally takes **~10 hours** to complete; PVTM reduced this **to ~7 hours.**

The ETL workflow to the left was used to reproduce the workload in production. The graph below the workflow shows the run-times for several batches with (red) and without (blue) PVTM. PVTM made the batches run faster by **26% on average**, and **66% on peak** periods of activity (e.g. start of day).

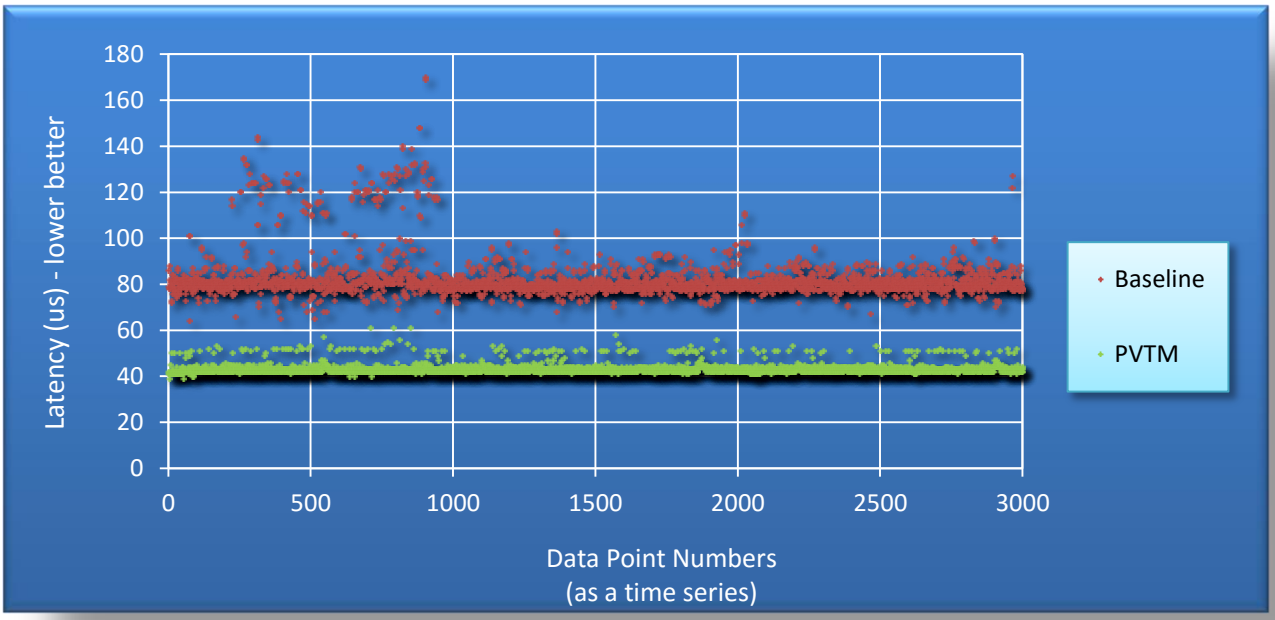
### PostgreSQL PGbench

PGbench is an open source, benchmark which simulates a retail bank with 100,000 bank accounts and 10 tellers.

PVTM **increased TPS by 19.5%** and reduced latency by the same margin; the staggering improvement was a **193.8% reduction in standard deviation**, showing the capability of PVTM to deliver highly deterministic results.



### Market Data connectivity suite




This case study shows performance improvements in a widely used market data pricing platform in the financial services industry. The graph above shows how PVTM enabled the system to perform with a **91% decrease in average latency** and **349% decrease in standard deviation** compared to a sample RHEL 7.1 system using the 'network-latency' profile for 'tuned', and the kernel's NUMA-optimized scheduler as a baseline (but without any thread pinning applied).

These figures were measured using the vendors' own tools shipped with the product using a 10K msg/sec rate, which is traditionally very difficult to optimize for latency because of the large gaps in between messages.

# Conclusion

As seen by the benchmarks above, Pontus Vision Thread Manager (PVTM) speeds up a diverse range of software applications; however we are still in a journey of discovery to find out which applications respond the best to thread pinning. During our various benchmarks, customers noticed that a lot of the statistics that PVTM captures also provide very good diagnostics of bottlenecks, which also help improve other areas of performance. Before embarking in any serious benchmark, it is important to see whether the system under test is suitable for thread pinning. During a proof of concept, we can quite quickly determine whether the hardware and the application are suitable for thread pinning. As a rule of thumb, if applications have fewer active threads than the number of cores on the server, they should be able to benefit enormously from PVTM.