NISTIR 7754

Experiments in Parallel Fingerprint Matching - Architectural Implications for Large Scale Fingerprint Matching Evaluation Systems

Antoine Fillinger[1]
Lukas Diduch[2]
Imad Hamchi[2]
Vincent Stanford[2]
[1] Dakota Consulting Corporation
[2] The National Institute of Standards and Technology

NIST
National Institute of Standards and Technology
U.S. Department of Commerce

NISTIR 7754

Experiments in Parallel Fingerprint Matching - Architectural Implications for Large Scale Fingerprint Matching Evaluation Systems

Antoine Fillinger[1]
Lukas Diduch[2]
Imad Hamchi[2]
Vincent Stanford[2]
[1] Dakota Consulting Corporation
[2] The National Institute of Standards and Technology

April 2011

TABLE OF CONTENTS

Abstract

The need for very large fingerprint benchmarks to test the efficacy of emerging commercial technologies for fingerprint minutia detection, matching, compression, and retrieval is well recognized. It is driving more intensive use of networked parallel processing clusters. This report demonstrated the feasibility of a parallel fingerprint matching hosted in the NIST Data Flow System Version II middleware layer, and identified specific hardware and software constraints that cause performance problems when introducing parallelism. The starting point consisted in a standalone matcher that takes two fingerprint images as an input and provides a matching score by using the well-known Bozorth fingerprint matcher. The publicly available NIST Special Database 14 and tools were used for matching and in order to handle fingerprint images and perform the minutia detection and print matching. Scalability to about 100 compute cores was shown using conventional gigabit Ethernet connectivity.

1 Acknowledgements and Disclaimers

The authors gratefully acknowledge the collaborative support from the Image Group including Mike Garris, and his sponsors, without whose support this work would have been impossible. We also wish to acknowledge Shahram Orandi, Craig Watson, Wayne Solomon, Ken Ko, and Mike Indovina whose assistance with the complex software environment necessary to process fingerprint evaluations was also essential. The Image Group's public datasets and reference implementation of algorithms for the Bozorth fingerprint matcher were also used. We note that any mention of open source or commercial software products such as Linux or Microsoft Windows™ and hardware lines like Intel microprocessors are purely for descriptive purposes and do not constitute recommendations by the National Institute of Standards and Technology, or certification or warrant that they are fit for any particular purpose. Finally, the authors gratefully acknowledge the very helpful reviews of the manuscript contributed by Dr. Steve Barry of MITRE Corporation. The final document is better for his reviews.

2 Introduction and motivation for this work

The work described herein is a prototype for parallel processing of minutia detection and matching of fingerprints from NIST Special Database 14 which NIST makes available to research groups in the fingerprint biometrics field. The data set was replicated to create processing loads that would stress the present generation computers and require the use of federated multicore processors in parallel.

The two key motivations for the work were: 1) The expressed desire to scale up the existing fingerprint benchmarks to test the efficacy of emerging commercial technologies for fingerprint minutia detection, matching, compression, and retrieval. This will drive more intensive use of networked parallel processing clusters than in the past. 2) The US-VISIT Extreme Scalability Workshop (*DHS Next Generation Technology Discovery Project - Workshop 2 Extreme Scalability, Arlington, VA July 2009*) which focused on the advanced computing requirements for the next-generation fingerprint processing systems which will produce very large numbers of fingerprints requiring realtime matching and response at points of entry to the US. There were two major approaches presented to providing the required distributed access, and low latency response times that are required for the US-VISIT application. These included Cloud Computing with non-local allocation of computing resources which offers minimum cost, and High Performance Computing with high bandwidth interconnect switching fabric which offers minimum latency.

3 Technological context of this work

These included, at a broad level:

Cloud computing: Exemplified by the Amazon Cloud Computing Business as presented by the Amazon CTO Mr. Werner Vogel who specializes in scalable and reliable enterprise computing. Mr. Vogel described the business and technical model that Amazon is developing in providing so-called cloud computing services. In this model many virtual computers, configured with desired operating systems and middleware such as relational database management systems as well as applications are instantiated at need on data centers worldwide. The distributed nature of the global hardware network raised questions of data security outside US borders. Google and Microsoft are also developing capabilities in this space.

The stated goals of the Amazon Cloud line of business are to provide:

- Quickly scalable computing infrastructure to support highly variable system loads and duty cycle
- Major Components of Amazon Cloud include a basic but adequate set of basic services for cloud computing including:
    - Virtualized Compute services (EC2)
    - Simple Queue Services (SQS)
    - Virtualized Storage: (S3 and Simple DB) – 70% of queries retrieve single rows, so stores by key, not optimized for joins
    - Messaging Services (EBS)
- Libraries of system images for preconfigured machines virtual machines with desired OS, database middleware, and matching application software libraries available at the instantiation of the machines.
- Rapid provisioning of large numbers of preconfigured virtual machines in under three minutes
- Integrated disaster recovery is included in the Amazon infrastructure
- Dynamic net I/O bandwidth for a unit price rate which scales on demand

Advantages imputed to cloud computing by the Amazon CTO include:

- Scalable – Users can add large numbers of virtual servers configured as they wish to meet demand spikes, and then relinquish them
- Cost effective – Users pay for servers by the compute core minute (~ 10 cents/minute/core)
- Reliable – Distributed Amazon infrastructure has redundant backup and distributed provisioning on a worldwide scale. This should provide resiliency under even regional disasters and failure of service/
- Secure – In some configurations, virtual clusters can run encrypted file systems in a virtually provisioned VPN which encrypts the node to node communication
- Can provision five stock server types including Microsoft Windows Server and Enterprise Linux operating systems, and with optional Oracle RDBMS, and other application licenses on an as needed basis in the standard configurations. Billing rates per CPU minutes varies by use of proprietary applications and middleware.
- Vogel was claimed that Peta-byte databases are “routine” in the Amazon Cloud infrastructure so data is scalable

Some key technical issues in cloud computing solutions for applications such as the FBI and US-VISIT systems, were observed and included:

- Deleting data from redundant cloud storage can be difficult, and globally distributed backups may move data to a variety of national settings outside the U.S.
- The data security status, and even location of databases there are no nodes instantiated on a provisioned cluster were not clarified when the issue was raised.
- Currently, the Amazon Cloud cannot guarantee provisioning in selected national settings to guarantee low latency connectivity, or data security within US borders.
- Currently, the Amazon Cloud does not support computational co-processors such as Graphic Processing Units (GPU's), so the computational element is the x86 Intel Compute core.

High Performance Computing Clusters: Exemplified by the Lawrence Livermore National Laboratory Sequoia Clusters as presented by Matthew Leininger, the Sequoia Project director. He described the Lawrence Livermore National Laboratory (LLNL) Hyperion Collaboration Testbed on Sequoia 2 petaflop cluster:

- The testbed currently consists of about 9,000 compute cores
- The networking and switching fabric consist of Infiniband 40 GB cards with 500 GB Flash-RAM buffers, and offer 10-30x faster execution than clusters with gigabit Ethernet connectivity. This speed improvement does not require any modification for message-based multi-node computational codes already developed for earlier hardware architectures.
- The cluster switching fabric offers ~300 GB/sec. of overall node interconnect bandwidth
- The Hyperion staff and users test and help maintain parallel file systems such as Linux PNFS
- Solid State Disks (SSD's) are rapidly emerging as a new storage tier, particularly in applications that require random access to large databases.
- Major corporate supercomputing players such as IBM, and HP and others are key Sequoia collaborators and provide detailed support for parallel system components

Some key technical issues for large-scale HPC as implemented in the Sequoia Project include:

- One processor core failure takes a whole host system node out of active use. This will be increasingly important as processors evolve to 64 cores and above.
- The emerging SSD tier still has limited lifetime rewrites and high cost per GB.
- Virtualization in the manner of cloud computing is used for testing but not production due to unacceptable runtime overhead for compute applications with high connectivity
- Application rewrite burden required to fully exploit distributed HPC if the applications were not developed for distributed processing at their inception.
- Exascale cluster will probably require 30 megawatts or more power – Processor power efficiency matters in terms of computation per watt matters.

Its proponents saw the HPC option as providing more compute power (because of much lower latency for internode communication) in a more secure environment (because of local firewalls) than cloud computing does. The following work is aimed at assessing similar, if smaller-scale, HPC Clusters. Its approach to fingerprint matching is based on the public NIST Biometric Image Software (NBIS) toolkit.

4 Experiments in scalable parallel distributed fingerprint matching at NIST

Having become convinced of the advantages of HPC for parallel fingerprint matching, our goal for this effort was to demonstrate the feasibility of a parallel fingerprint matching system on a local HPC-style cluster, and identify the specific constraints and performance problems that arise when introducing parallelism. The bottlenecks that can appear when scaling such systems are also subjects of the study. The starting point consisted in a standalone matcher that takes two fingerprint images as an input and provides a matching score by using the well-known Bozorth fingerprint matcher. Publicly available tools described below were used in order to handle fingerprint images and perform the minutia detection and print matching.

4.1 The NBIS Toolkit

The authors made heavy use of the NIST Biometric Image Software distribution, which is developed by the National Institute of Standards and Technology (NIST) for the Federal Bureau of Investigation (FBI) and Department of Homeland Security (DHS). Further documentation can be found at (http://www.nist.gov/itl/iad/ig/nbis.cfm). The NBIS utilities fall under seven general categories:

- A neural-network based fingerprint pattern classification system, called PCASYS, automatically categorizes a fingerprint image into the class of arch, left or right loop, scar, tented arch, or whorl. This is an updated system that includes the use of a robust Multi-Layered Perceptron (MLP) neural network. It is the only known no cost system of its kind.
- A minutiae detector called MINDTCT that automatically locates and records ridge ending and bifurcations in a fingerprint image. This system includes minutiae quality assessment based on local image conditions. The FBI's Universal Latent Workstation uses MINDTCT, and it too is the only known no cost system of its kind.
- A NIST Fingerprint Image Quality (NFIQ) algorithm, which analyses a fingerprint image and assigns a quality value of 1 for highest quality, to 5 for lowest quality, to the image. Higher quality images have been shown to produce significantly better performance with matching algorithms. The ability to retrain the NFIQ weights is provided with the software utility programs called FING2PAT, ZNORMDAT, and ZNORMPAT.
- A reference implementation of the ANSI/NIST-ITL 1-2007 (AN2K) "Data Format for the Interchange of Fingerprint, Facial, Scar Mark & Tattoo (SMT) Information" standard is included. This reference implementation contains a suite of utilities designed to read, write, edit, and manipulate files formatted according to this interchange standard. The utilities support updated and new record types introduced by this latest version of the standard (Record Types 9, 13, 14, & 15).
- A large collection of general-purpose image utilities (IMGTOOLS) are also included to support the processing of fingerprint images. Source code is provided for Baseline JPEG, Lossless JPEG, and the FBI's Wavelet Scalar Quantization (WSQ) encoders and decoders. (The Baseline JPEG code uses the Independent JPEG Group's compression/decompression libraries.) Utilities are also provided that support color component interleaving, colorspace conversion, and format conversion of legacy files distributed in NIST fingerprint databases.
- A fingerprint matching algorithm, BOZORTH3, which is a minutiae based fingerprint matching algorithm. It will do both one-to-one and one-to-many matching operations. It accepts minutiae generated by the MINDTCT algorithm.

- A fingerprint segmentation algorithm, NFSEG, which will segment the four-finger plain impression found on the bottom of a fingerprint card into individual fingerprint images or it can be used to remove white space from a rolled fingerprint image [1].

4.2 NIST Special Database 14 Version 2 of April 2002

NIST Special Database 14 [2] is being distributed for use in development and testing of automated fingerprint classification and matching systems on a set of images which approximate a natural horizontal distribution of the National Crime Information Center (NCIC) fingerprint classes and were compressed using an implementation of the Wavelet Scalar Quantization (WSQ) compression specification.

The database consists of three CD-ROM disks, each containing 9,000 image pairs and requiring approximately 670 megabytes of storage compressed and 11.5 gigabytes uncompressed (18.0:1 average compression ratio). Each segmented image is 832 by 768 pixels and classified using the NCIC classes given by the FBI. The database also includes example software. NIST Special Database 14:

- Has 27,000 pairs of segmented 8-bit gray scale fingerprint images compressed with an implementation of the WSQ compression specification
- Has NCIC classifications given by the FBI for cards selected randomly thus approximating the natural horizontal distribution of the NCIC classifications.
- Were scanned at ~19.685 pixels per mm, or equivalently 500 pixels per inch ppi.
- Has Image format documentation and example software written in ANSI C
- The first 13,500 fingerprint images are the same as the NIST Special Database 9, Volumes 1-5 images archived in lossless format,

Though it is relatively small scale, this database has been widely used for automated fingerprint classification research, including algorithm development, and system training and testing. The present study uses images from the NIST Special Database 14 for performance evaluations that validated the feasibility of the distributed version of the 1:1 matching system.

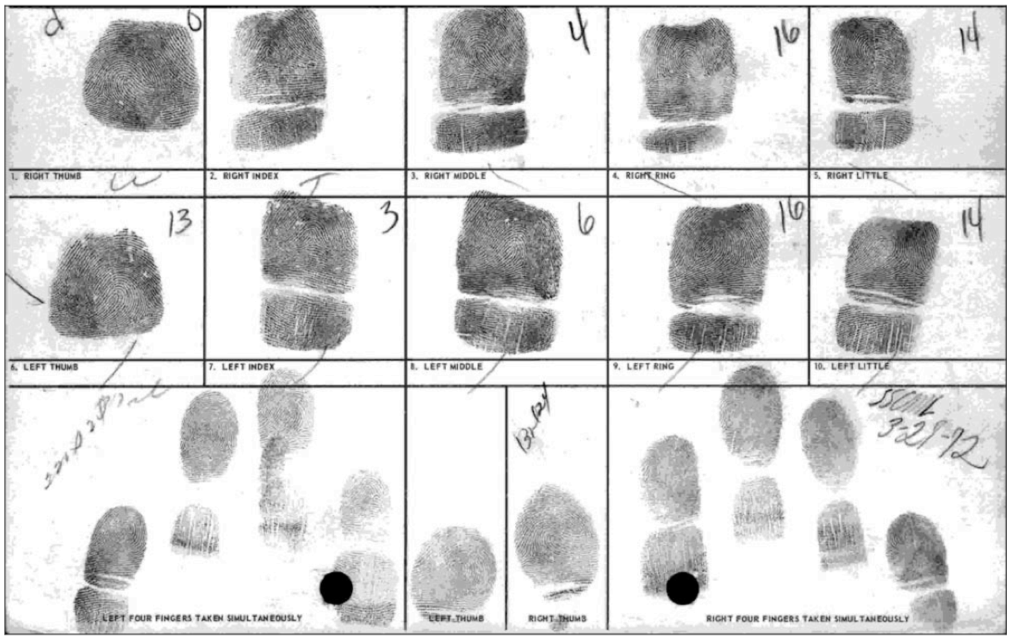

Figure 1 Cropped tenprint card filled with fingerprints.

5 The NIST Data Flow System parallel processing middleware

The NIST Data Flow System II is a middleware data transport layer that moves data between client nodes via flows registered with the local NDFS Daemon and which are used via a publish-subscribe protocol [3]. Clients can produce and/or consume streams of data called flows. An application is thus represented as a graph with flows as edges and clients as nodes, in which client nodes functional codes perform successive operations on data pipelines. A major strength of the system is to abstract the network locations where data is produced and consumed. Clients connect to each other through flows that they subscribe to by name. These flows names are unique pair type/ID to identify them to the middleware, which then connects data producers to consumers. If a node offers a flow that no other flow subscribes to, the local NDFS Daemon will use flow control commands to cause the node to stop producing data when a specified amount has been enqueued into the flow buffer.

5.1 NDFS server

NDFS servers (daemons) discover the peer servers on the local network and maintain lists of active servers, clients, and flows by using XML-based control messages. The high bandwidth data flow blocks are sent in binary format with block headers (including time tags) in the interest of low transmission overhead. NDFS servers have the capability to spawn client nodes on request and to establish data communication among them. They do this using the application description tables that are shared across all NDFS Daemons. In the event of node failures this permits the Daemons to attempt to re-spawn failed clients to reestablish the complete data flow application.

5.2 Data Flows

Flows are buffered data streams connecting a producer client to an arbitrary number of consumer clients and provide distributed data transport. Flows are defined with a type, a name, and a connection ID. It is possible to add properties to flows and to create custom flow types.

5.3 Processing Clients

A client is an executable program that uses the NDFS-II library to consume and/or produce flows. There are also client nodes that do not use flows, but provide control over the NDFS-II network such as the Control Center.

5.4 Flow Duplicators

The NDFS-II is designed to support applications requiring high data rates. It takes advantage of gigabyte networks and avoids redundant copying of data to minimize the bandwidth and thus speed up the communication between multiple remote client nodes. A goal at the beginning of the project was to transport the data as efficiently as possible. So we introduced the concept of duplicators in order to optimize the transport. Duplicators are stand-alone programs handling flows within and between hosts. There is one duplicator per instance of flow per host. Duplicators are spawned by NDFS servers on-demand when client nodes create flows.

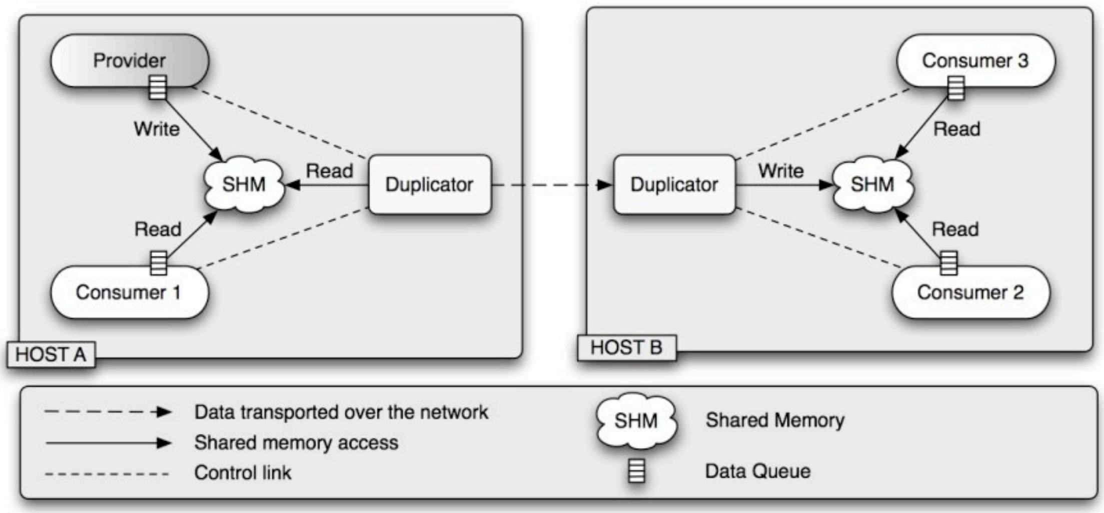

Figure 2 A NDFS-II application composed of four clients. There is one producer feeding with data three consumers. The producer and one consumer run on the same host (A) and the two remaining consumers run on a remote host (B).

On each host, a duplicator handling the flow instance manages the shared memory used to store data blocks and communicates with other remote duplicators if necessary.

The data is transported as follows. *Provider* writes a data block in the shared memory and notifies the duplicator that the block is available for reading in the shared memory. The duplicator then notifies *Consumer 1* running on the same host that a block is available for reading in the shared memory and also sends the data block to the remote duplicator, which copies it into the shared memory once received and notifies its consumers (*Consumer 3* and *Consumer 4*) that a new block is available for reading. Once the duplicator on host A received the confirmation from *Consumer 1* that it has read the data block, and also the confirmation from the duplicator on host B that the block has been properly received, the duplicator on host A notifies *Provider* that it can write a new data block in the shard memory.

Introducing the duplicator concept has several advantages. It allows concurrent reading of the shared memory within a host and avoids multiple transfers of the same data blocks for efficient network usage when several consumers are connected to the same flow produced on a remote host.

Clients also benefit from an internal buffering mechanism provided by flows. When a producer provides a data block, this block is not sent directly but is en-queued in the internal flow queue of the client node. A separate thread in the background running in the client node de-queues the data block and passes it to the duplicator. Symmetrically for consumers, a thread gets data blocks from the shared memory when notified by the duplicator and en-queues it in the internal flow queue. This method of transporting data avoids maintaining a reference count at the producer level and transports the data as fast as possible independently of consumers requests; therefore, when a consumer requests a new data block it could already be in the internal flow queue and available immediately.

The size and behavior of the flow queue is customizable to allow blocking or non-blocking modes depending on the requirement of the supported application.

5.5 The ACE Adaptive Communication Environment

The NDFS-II is a cross-platform system that runs on a variety of GNU/Linux, Unix, Mac OS X and Windows operating systems. This platform neutrality is achieved thanks to the ADAPTIVE Communication Environment that provides a common API for the low level operations required by the NDFS-II.

The ADAPTIVE Communication Environment (ACE) is a freely available, open-source object-oriented (OO) framework that implements many core patterns for concurrent communication software. ACE provides a rich set of reusable C++ wrapper facades and framework components that perform common communication software tasks across a range of OS platforms. The communication software tasks provided by ACE include event demultiplexing and event handler dispatching, signal handling, service initialization, interprocess communication, shared memory management, message routing, dynamic configuration of distributed services, concurrent execution and synchronization.

ACE is targeted for developers of high-performance and real-time communication services and applications. It simplifies the development of OO network applications and services that utilize interprocess communication, event demultiplexing, explicit dynamic linking, and concurrency. In addition, ACE automates system configuration and reconfiguration by dynamically linking services into applications at run-time and executing these services in one or more processes or thread[4].

6 The parallel matching architecture

6.1 Overview of the NIST SDK

The parallel architecture has been inspired by a stand-alone matching system that extracts fingerprint templates from two compressed images and then uses these templates to return a matching score. This stand-alone system follows the NIST API Requirements for Fingerprint SDK Biometric Evaluation and therefore can potentially use SDK from vendors adhering to the same API.

The parallel architecture is basically a functional decomposition of the standalone system. Several key functionalities have been identified:

- Loading a image/fingerprint template from a file
- Saving fingerprint templates to a file
- Getting a fingerprint template from a raw picture
- Returning a matching score of two fingerprint templates
- Dispatching evaluation commands
- Gathering matching results

Besides dispatching the evaluation commands and gathering the results, all other functionalities are found in both the stand-alone and in the distributed version of the matching system.
The philosophy behind the NDFS-II is to move the complexity of an application to the graph and keep the client nodes composing the graph as simple as possible.

6.2 NDFS-II Wrappers, flows, and clients

6.2.1 Clients

Clients are stand-alone programs that use the data transport capability provided by the NDFS-II middleware library to provide input/output communication operations with other clients. They utilize libraries providing domain specific functions including matching, template extraction, uncompressing WSQ pictures. The following functionalities have been wrapped in NDFS clients:

- ImageLoader: Takes a command containing a path to an image from an input flow, loads this image and decompresses it if necessary and sends it to an output flow.
- FMRLoader: Takes a command containing a path to a fingerprint minutiae record file, loads and sends it to an output flow.
- imageToFMR: Takes an uncompressed image from an input flow, gets the fingerprint templates from it (using mindtct), and sends it to an output flow.
- bozorthMatcher: Takes two fingerprint templates from an input flow, computes their matching score (using bozorth3) and sends it to an output flow.
- minutiaeCreatorController: Manages the extraction of fingerprint templates from images files. Each fingerprint template extraction is represented as single command containing the path of a picture from which the fingerprint template is constructed. The client sends these commands using output flows, and gathers the minutiae fingerprint templates from input flows. These fingerprint templates are then stored locally for future use.
- matchController : Manages an evaluation consisting of a list of fingerprint probes. Each probe request is represented as a single command containing the path of the two fingerprint templates to be matched.

6.2.2 Application graphs

By connecting these clients we built application graphs whose distributed clients communicate via flows. Two application graphs were created to support the evaluation of fingerprint systems. They each support a part of the evaluation.

- The first one creates a list of fingerprint templates from a fingerprint image list
- The second one performs the matching of fingerprint templates pairs using the results generated by the first graph.

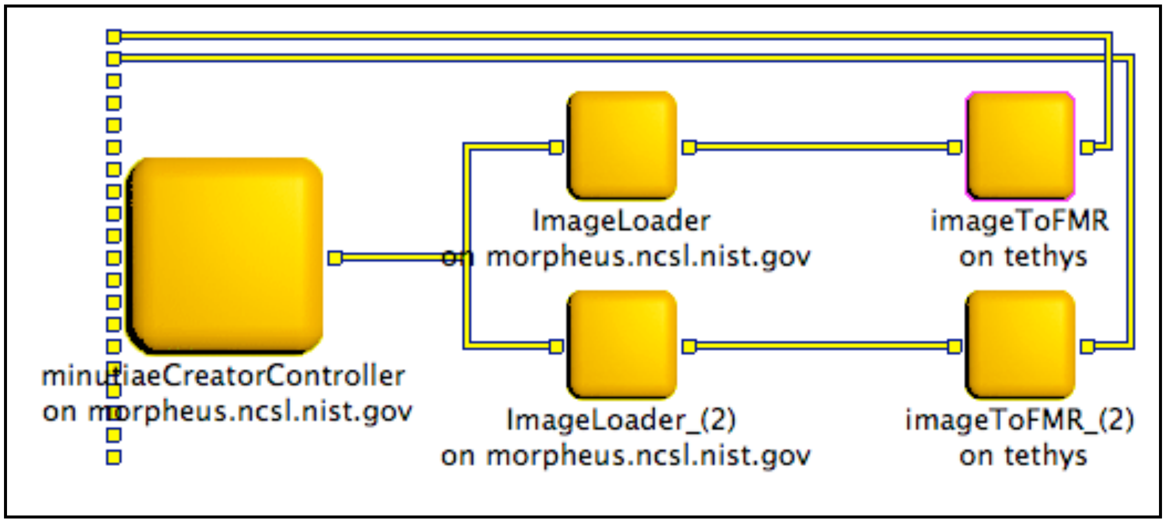
ImageLoader
on morpheus.ncsl.nist.gov
imageToFMR
on tethys
minutiaeCreatorController
on morpheus.ncsl.nist.gov
ImageLoader_(2)
on morpheus.ncsl.nist.gov
imageToFMR_(2)
on tethys

Figure- 3. Gets fingerprint templates from image files. Commands are sent by minutiaeCreatorController to an array of ImageLoader clients. Commands contain a path to an image accessible from the computer that runnning the ImageLoader client. Each ImageLoader instance loads a file containing an image and decompresses it, if necessary, and then sends it to imageToFMR that uses it to generates a fingerprint template. Theses templates are then sent to minutieCreatorController, and saved for later use.

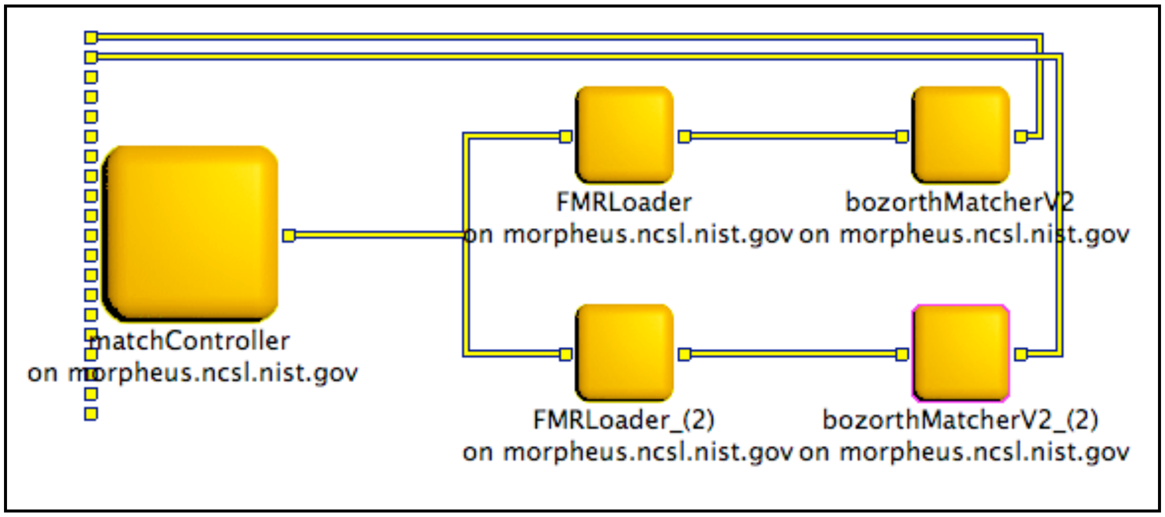
FMRLoader
on morpheus.ncsl.nist.gov
bozorthMatcherV2
on morpheus.ncsl.nist.gov
matchController
on morpheus.ncsl.nist.gov
FMRLoader_(2)
on morpheus.ncsl.nist.gov
bozorthMatcherV2_(2)
on morpheus.ncsl.nist.gov

Figure-4. Matches fingerprint templates . Commands are sent by matchController to an array of FMRLoader clients . Commands contain the path of two fingerprint templates that we want to match against each other. These templates must be accessible from the an FMRLoader clients. Each FMRLoader loads a pair of templates and sends it to a bozorthMatcher client that generates a matching score from the two fingerprint templates. Matching scores are gathered by matchController which saves them in a report.

These graphs are designed to scale to many processing elements to take full advantage of the computational power provided by clustered multicore computers with one matcher (or one fingerprint template extractor) per core.

6.2.3 Flows

Several flow types were developed to support the needs of the biometric operations:

- Flow_Command: This flow connects the controller clients (matchController and minutiaCreatorController) to the ImageLoader and FMR_Loader clients. It transports commands for loading fingerprint minutiae records and fingerprint images. Commands information such as a command recipient and the path of a picture/minutiae record to load. The recipient is necessary because many consumer clients can share the same instance of a Flow_Command.
- Flow_Image: As its name indicates, this flow transports an image (uncompressed) and various information required to handle it (dimensions, depth, etc.) It is used by ImageLoader to send images to imageToFMR for minutiae extraction.
- Flow_FMR: It transports a fingerprint minutiae record. It is used by minutiaeCreatorController to gather minutiae extracted by imageToFMR clients for future matching, and by the FMRLoader clients to send minutiae records to the matcher clients on their pipelines.
- Flow_Result: It transports the numerical result of a match and all the pieces of information related to this result.

From a design viewpoint, these flows are all derived from a common flow that factors the common functionalities needed in the different flow types and also provides interoperability between flow types.

7 Test methods and results for simulated large-scale databases

Several experiments were performed to validate the distributed architecture design. The main idea was to define a general model and scale it gradually to identify the bottlenecks and constraints arising with increasing scale.

The graph topology in Figures 3 and 4 can be described as a master client having an arbitrary number of processing branches or pipelines. Commands leave the master client, are processed by minutiae/matching pipelines and results are returned to the master client thanks to a loopback.

The first performance evaluations used the NIST Special Database 14 that contains the 27,000 pairs of fingerprint images belonging to 2,700 subjects. Each subject has its 10 fingerprints record in two different datasets.

The first evaluations consisted of building a similarity matrix of matching scores for one finger (thumb, index, ring…). First, a finger was chosen. Then the chosen finger of every subject of the first dataset was matched against the same finger of every subject in the second dataset. We were building a matrix with a size of 2700 by 2700 whose main diagonal contains the match score and the rest of it non-match scores.

In order to perform the experiment, we had four 8-core computers dedicated to matching and one 8-core hyper threaded computer for storing the images and templates. The controller clients (minutiaeCreatorController and matchController), as well as the data loader clients (ImageLoader and

FMRLoader) ran on the hyper threaded 8-core computer. The 8-core computers were used to host the matchers or imageToFMR clients depending on the graph we were running.

We started with a graph having only one processing pipeline (one matcher). The matcher was allocated on one of the 8-core computers. We ran the experiment and measured how long it took to build the matrix. Then we added a second processing pipeline to the graph, so one more matcher was allocated to the 8-core computer and we measured the duration it took to build the same matrix. We added matchers to the 8-core computer until the number of matcher was equal to the number of computing cores provided by the host; 8 in the present case, but 16 on our production nodes. The ninth matcher was added to the second 8-core computer until it had one per core, and so on. The duration of time required to process the test was measured. This experiment was repeated with the graph having as few as one branch and up to 32 branches. We allocated one matcher per core, up with 32 matchers running on 32 cores located on 4 computers.
The experiment was performed using four Mac Pro systems running a GNU/Linux distribution, each with 8 computing cores and 32GB of RAM. The four 8-core computers used in the simulation were almost homogeneous; the computing power difference between the most and least powerful computers was only about 10%.

7.1 CPU Saturation

Full utilization of all 32 processor cores was our goal. This would mean that the middleware itself is not introducing latency resulting in idle processor time. The processor overhead due to the middleware is necessarily part of the load for the processors. So the data for images or fingerprint minutiae records were delivered fast enough to the applications that requiring the most computational power.

Having an application composed of multiple processes naturally takes advantage of clustered multi-core architectures. Modifying an existing application to introduce concurrency with all the programming errors and pitfalls that can come with it is fraught with difficulties.

We chose instead to break apart the original program by identifying its core functionalities and put each of them in a standalone program connected via flows. When this step is done, a careful allocation of the application graph leverages the multiple cores available, not only within one single machine (as it is the case with the threaded version), but across the whole cluster network thanks to the capability of NDFS-II to abstract data source/sink locations.

Full utilization of the CPU depends on how the graph is allocated. The main factor that can affect CPU saturation is data starvation. If one client consumes and processes its data faster than the arrival rate, the CPU used by this client will not be fully utilized. The main causes of data starvation are usually I/O performed on either the network or local disk storage. We believe that in an environment of many processes per host, disk IO performance may be greatly reduced by seek times of rotating disks. Fully loading the CPUs means that the network is able to deliver adequate data for the computing resources, and so the processors are used to best advantage. When this goal is reached, the possible improvements are to update algorithms for improved performance viewpoint and getting more powerful hardware.

7.2 Match scores VS non-match scores

The results from the first round of experiments couldn't be interpreted because the matchers take a longer time to return a score of two fingerprint templates having a positive match (from the same subject) than two fingerprint templates having a negative match (from two different subjects). The score of a positive match is referred as a match score while the score of a negative match is referred as a non-match score. The experiments showed that it takes about 3 times longer for bozorth3 to return a score from a positive match compared to a negative one. These differences created a problem in the measurements because we could end up in cases where many fingerprints that match were sent on one branch while other branches processed a lower ratio of pairs that match. This would put an increased load on a single processing branch and therefore create an unbalanced distribution that prevented any results interpretation. At the end of the test, all branches were done but one.

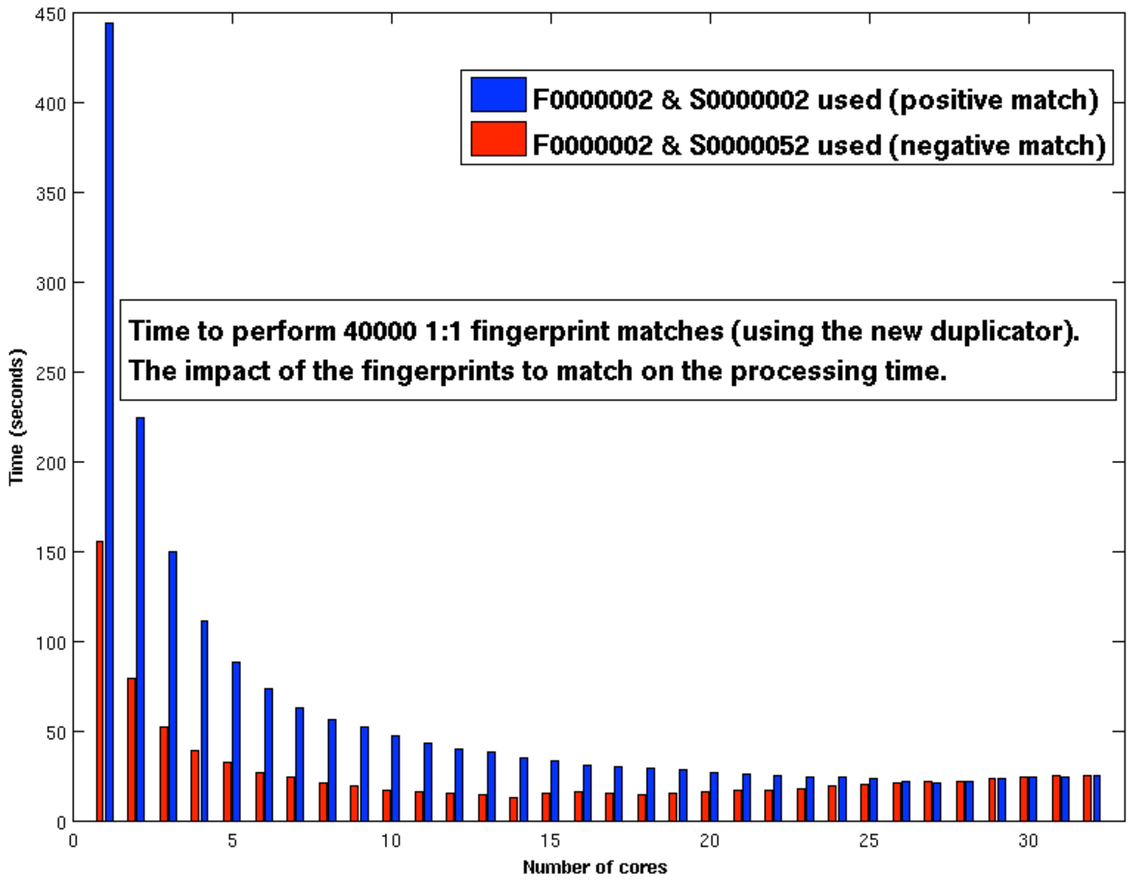

Figure-5. Time to perform the verification is impacted by the pair of fingerprints. The time to verify a pair that gives a match score is about 3 times longer than a pair that gives a non-match score. The new duplicator is used here.

Using the whole dataset made for difficult interpretations of performance test results. The tests needed to be better designed to clarify the factors affecting processor utilization. But it did highlight a very important point: naive job scheduling to distribute an evaluation by simply taking account of the computing power of your machines is not enough to guarantee efficient resource utilization. To do it efficiently, we would either need prior knowledge of how long it could take to match each pair (which is not feasible) or have feedback information from the pipeline itself. We implemented feedback from the data flows using the flow control and setting the number of fingerprint blocks allowed in the queue to one. This allows us to enqueue a single print set for processing and have the flow queue hold the print dispatcher off until that print has been consumed by the matcher subscribing to the particular node.

So the method to do performance testing was refined to use only one fingerprint pair at a time during each experiment. Tests were divided in two subsets: the first one used a pair where fingerprint matches, while the second one used a pair with fingerprints that didn't match. Because it takes more time for a matcher to return a match score than a non-match score, these two subsets of tests allowed us to identify and manage several bottlenecks.

7.3 NDFS Duplicator architecture

The original NDFS duplicator has a single threaded architecture managing a single block of shared memory. It was mainly used for applications that connecting perhaps ten or twenty providers to a like number of consumers. Multimedia data were the data type transported [5] so the data blocks sent were usually large, as in applications supporting agent-based distributed graph search [6]. The single thread duplicator architecture was adequate for this case. When it was used to connect a single provider to many consumers using smaller data blocks, the weakness of this approach became apparent.

This single threaded architecture became an issue when the NDFS-II started to be used to support graphs such as the one described in Figure 6 with large numbers of graphs. In this application, one provider sends commands to many consumers. All consumers sharing the flow receive each command containing a recipient ID. As a result, all but one discards the command because only one consumer has the recipient ID that matches. It means that, with N being the number of consumers, only 1 command out of N is useful for the consumer. So the command usefulness ratio decreases with the number of consumers increasing and, at the same time, the duplicator load increased proportionally with the number of consumers connected to it.

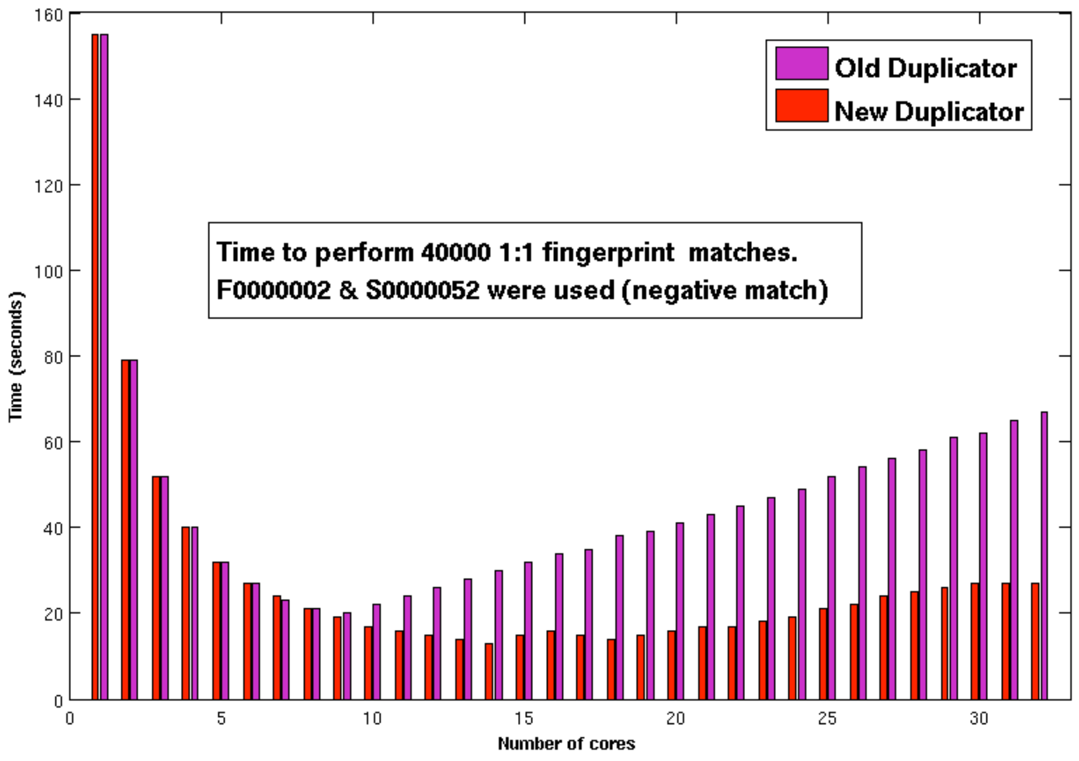

Figure-6. The performance of the new version of the duplicator is better than the previous version when the application is scaled up. A fingerprint pair having a negative match is used here. The number of pairs required to keep matchers constantly busy is greater when using matching pairs than when using non-matching pairs because it is faster for them to return a non-match score than a match score. As a result, using a non-matching pair during this experiment puts a heavier load on the duplicator than the one using a pair with a positive match.

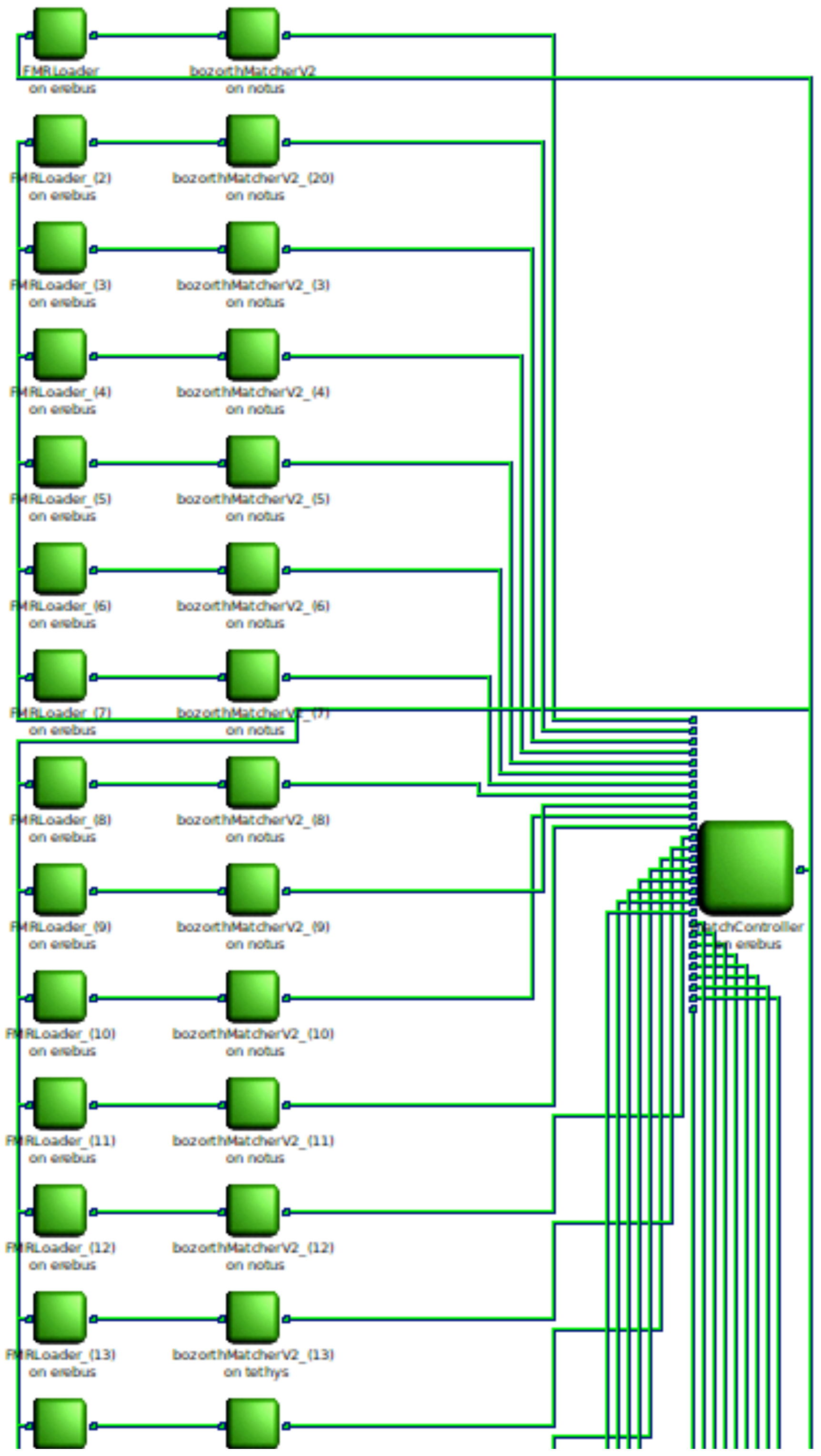

Figure-7. Partial view of the graph used for the distributed matching application on four 8-core computers. The topology of the graph puts a heavy load on the duplicator that transports data between the matchController and the FMRLoader clients.The new duplicator performs better than the old one but we hit a performance ceiling. Once the ceiling is reached, increasing the scaling factor of the application becomes counterproductive: there is no more performance gain. Performance starts to drop due to an increase in overhead induced by the scaling effect (the application graph is getting more complex).

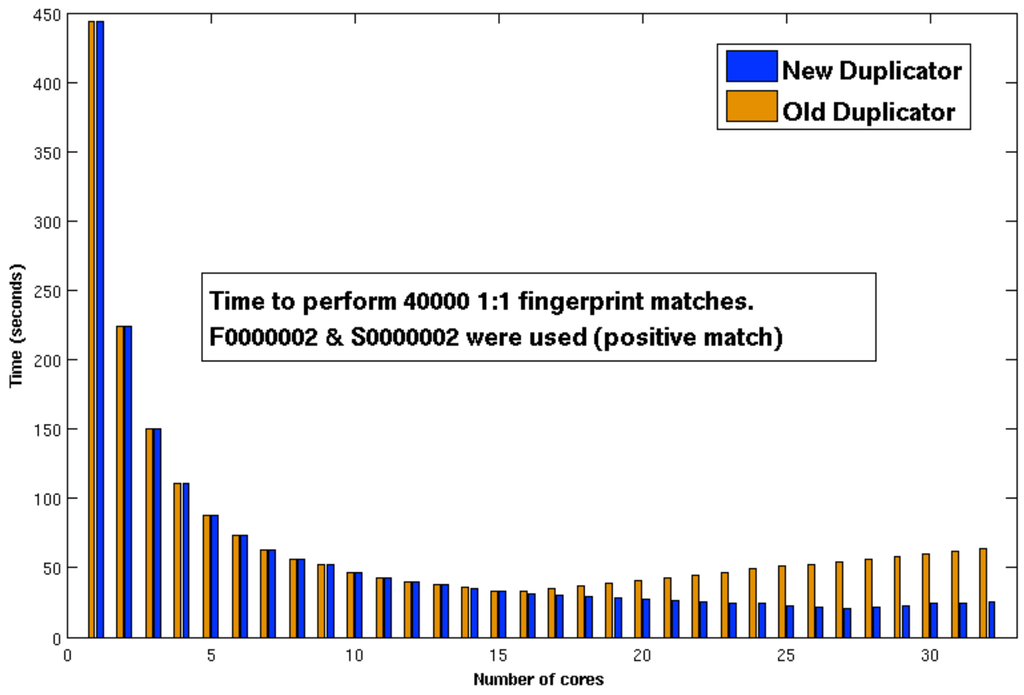

Figure-8. Fingerprint pair having a positive mach is used in this experiment. The load on the duplicator is lighter than when using a fingerprint pair with a negative match. This explains why the application scales better than when using a pair with a negative match.

After acknowledging that the topology of the graph was working against the duplicator, and after seeing that we didn't scale as expected as shown in (darker) purple in Figure 8 and in (lighter) orange in Figure 8, we saw the need to have a better duplicator. Figure 7 shows that the performance peak using the old duplicator is reached with 9 cores when using a pair of minutiae giving a non-match score, and at 16 cores when using a pair of minutiae giving a match score.

The duplicator was the first bottleneck we identified when we increased the scaling factor of the distributed application. So it was clear that the duplicator needed to be redesigned to be able to handle more load arising from this new graph topologies. Being able to communicate concurrently with each of its consumers at the same time instead of one after another seemed to be required. Also using a shared memory pool bigger than one block would allow the producer to write while consumers were reading different data blocks.

The new design of the duplicator that increased concurrency on the shared memory and among clients was successful; as the duplicator was able to move commands faster between the matchController client and the FMRLoader clients. But it still didn't scale linearly up to 32 cores. This is highlighted in red in Figure 9 and blue in Figure 10.

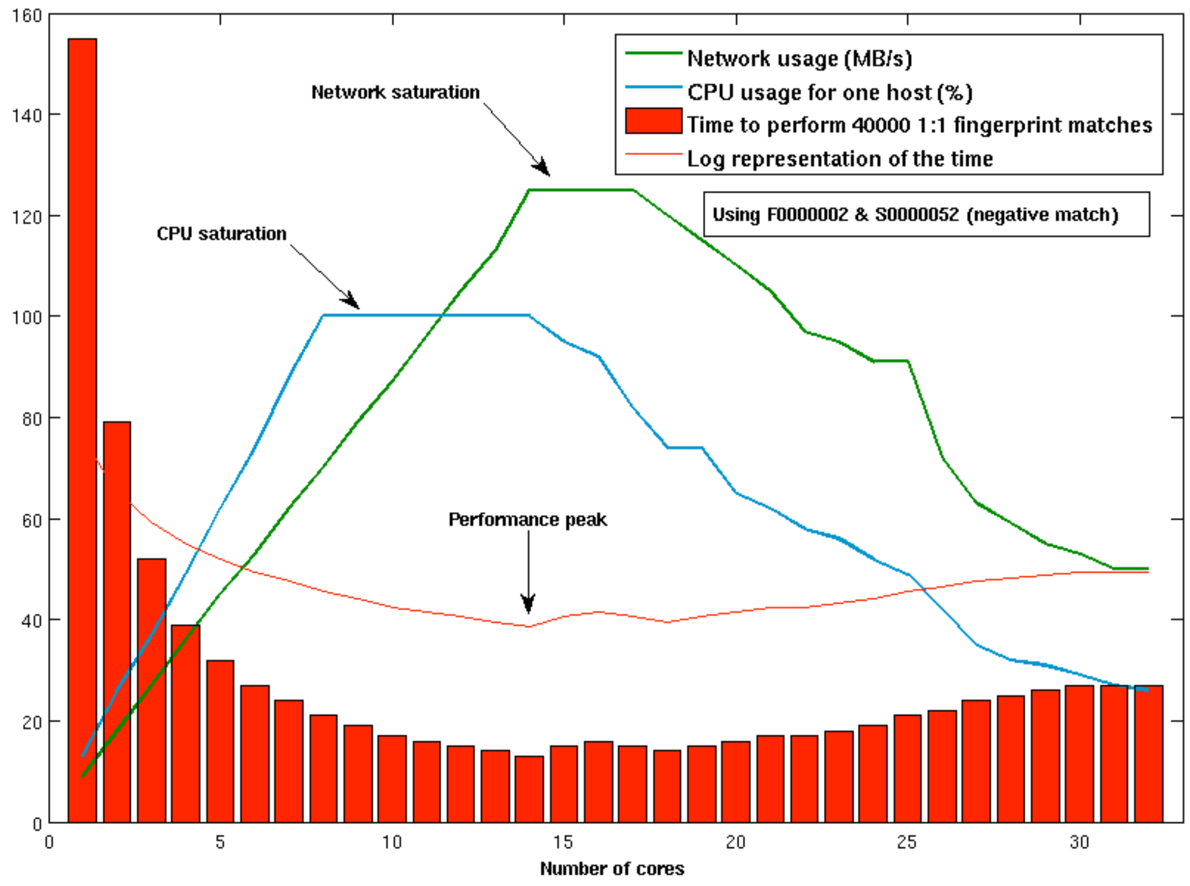

Figure-9. The performance of the new duplicator. We reached the performance peak at network saturation. Once the network capacity is fully used, scaling the graph only reduces the capacity dedicated to the transport the information. The capacity used for management (overhead due to more branch on the graph, i.e more connection using the network) is increased, which explains the beginning of the performance drop.

When we keep scaling up, we noticed that the network use decrease from its saturated point. The decrease in network use means that even though the network was the bottleneck at some point, there is another bottleneck beside the network. We identified this new bottleneck as the duplicator that has reached its peak of performance with its new design. This problem could be solved or at least ameliorated by aggregating commands into blocks that do not require as many flow blocks to be processed. Future work should be directed to this as a goal.

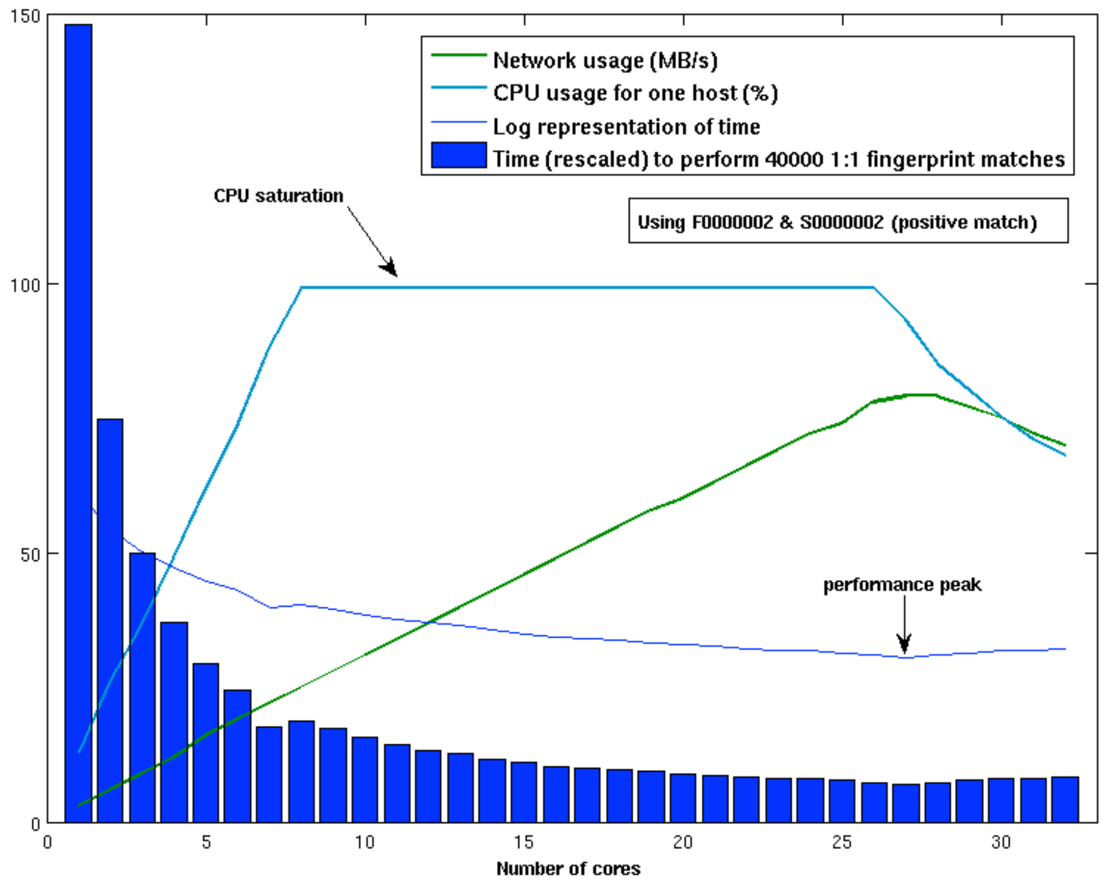

Figure-10. The performance of the new duplicator in blue in Figure 8 are shown along with the CPU usage of first 8-core of used in the experiment and the network usage of the host storing the templates to match. It can be seen that we never reach network saturation before a drop in performance.

These results showed that the bottleneck when scaling was not the CPUs, nor on the network. We note that the bottleneck is actually on the network when using a pair of fingerprint giving a non-match score when the experiment is run using between 14 and 17 cores. Above 17 cores, the network is not overloaded, so the network was discarded as being the main performance bottleneck. This conclusion is confirmed by looking at Figure 10, we never reach the saturation point of the network, but the more the application is scaled, the more performance drops.

So the new design of the duplicator was not sufficient performant. Analysis of its behavior at runtime showed that the new design was good, and performance was clearly improved. While the duplicator was very fast, it was taking more than half of the resources of the hyper threaded 8-core computers while it was only taking 1/16 before due to its single threaded architecture. Investigation showed the problem was in the data granularity the duplicator was transporting between the controller clients and the ImageLoader/FMRLoader clients.

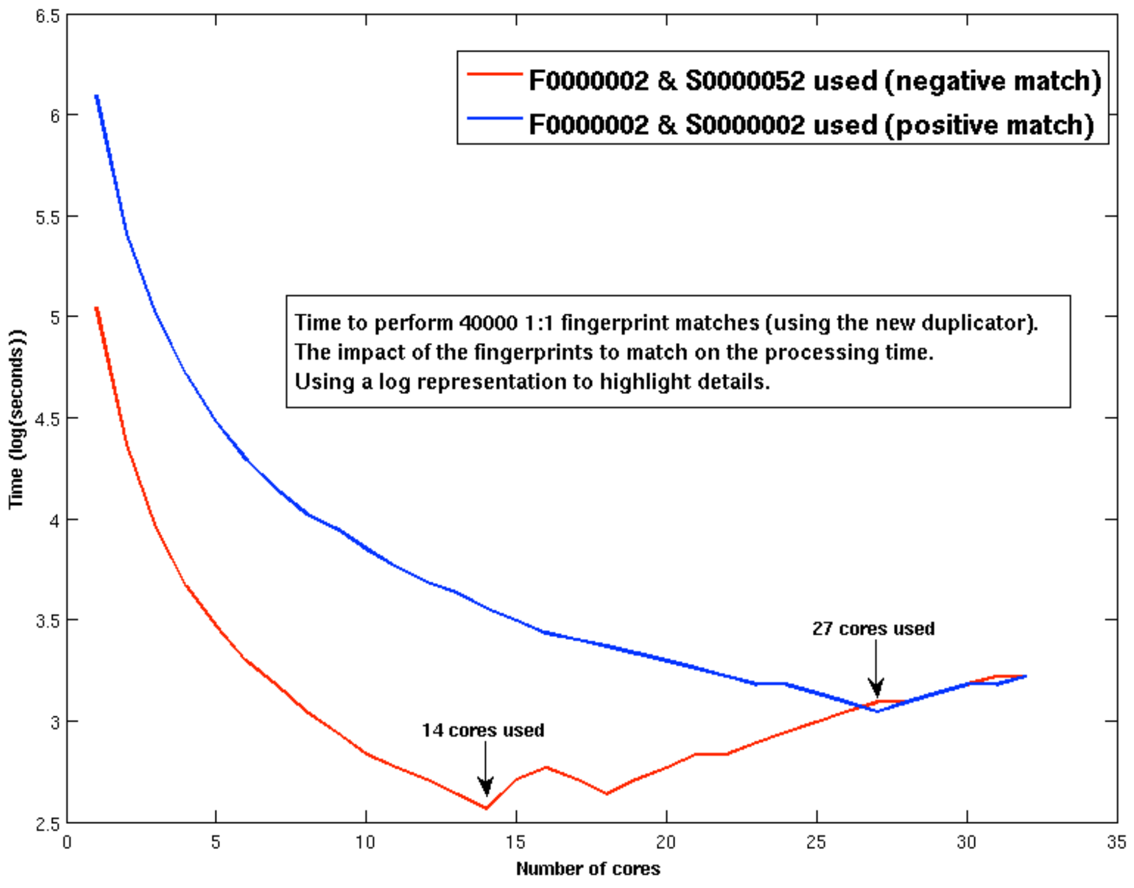

Figure-11. The time to verify a pair that gives a match score is about 3 times longer than a pair that gives a non-match score. This has an impact on the data transport capacity needed to keep matchers busy. The performance peak is reached using 27 cores when having a pair giving a match score. With a non-match score, the performance peak was reached using only 14 cores.

7.4 Command aggregation

The command usefulness ratio being negatively correlated with increasing scale was the bottleneck. Even though the duplicator has been redesigned, its new architecture was not efficient enough to support the needs of the biometric distributed applications. The duplicator (that has been designed solely toward performance) could not transfer useful commands to the FMRLoaders fast enough to keep the matchers busy by sending them fingerprint templates fast enough. Sending individual commands were just too small to be sent individually. Moreover, the duplicator was using more than half of the computational power of our hyper threaded 8-core computer when working at such thin granularity.

We used the method of command aggregation to reduce duplicator communication overhead by enqueuing fewer and larger command blocks. The implementation of the concept of command groups enabled us to aggregate them to solve this problem.
The duplicator handles the data block management in the shared memory within a machine (and sends data over the network when necessary) for a specific flow instance. Having the commands grouped by a factor N reduced its duplicator load by a similar factor.

So the performance bottleneck that laid between the master client and its consumers was resolved by modifying two layers in the system:

- At the middleware layer by redesigning the duplicator for higher performance.
- At the application layer by command aggregation to reduce the time spent in the enqueuing of commands and transporting them to the relevant matchers.

So we implemented the commands aggregation, and we ran our experiments again.

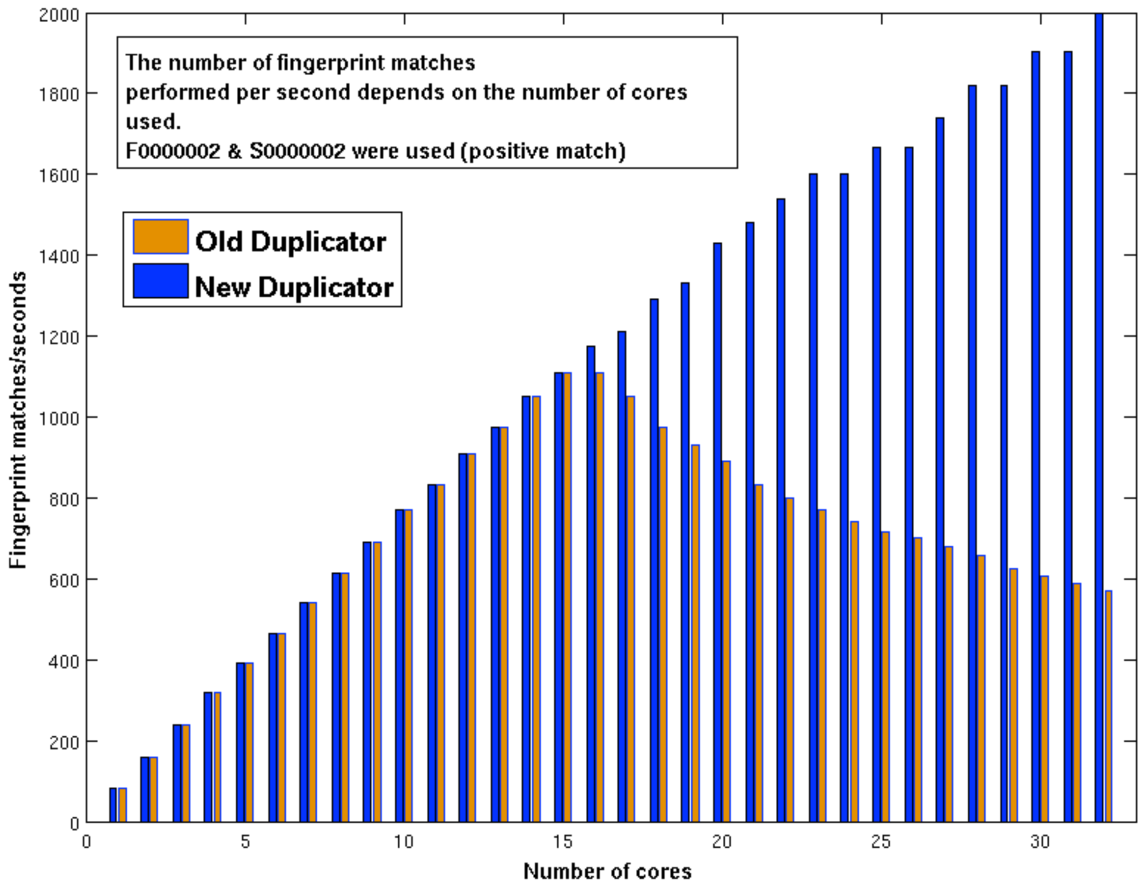

Figure-12. Performance difference between the original duplicator without command aggregation, and the redesigned duplicator with command aggregation. No bottleneck preventing the experiment to scale linearly up to 32 cores is noticed when using a fingerprint pair having a positive match.

As Figure 12 shows, redesigning the duplicator provided the expected results: performance scaled up in a linear way. It needs to be noted that the experiment, shown in figure 12, used a pair with two fingerprints giving a match score (positive match). Therefore we knew the new duplicator needed to support a data transport speed three times higher from match vs. non-match performance to handle the worst scenario. When performing the experiment with a pair composed of fingerprints giving a non-match score, the obtained performance didn't meet our expectations as shown on Figure 13.

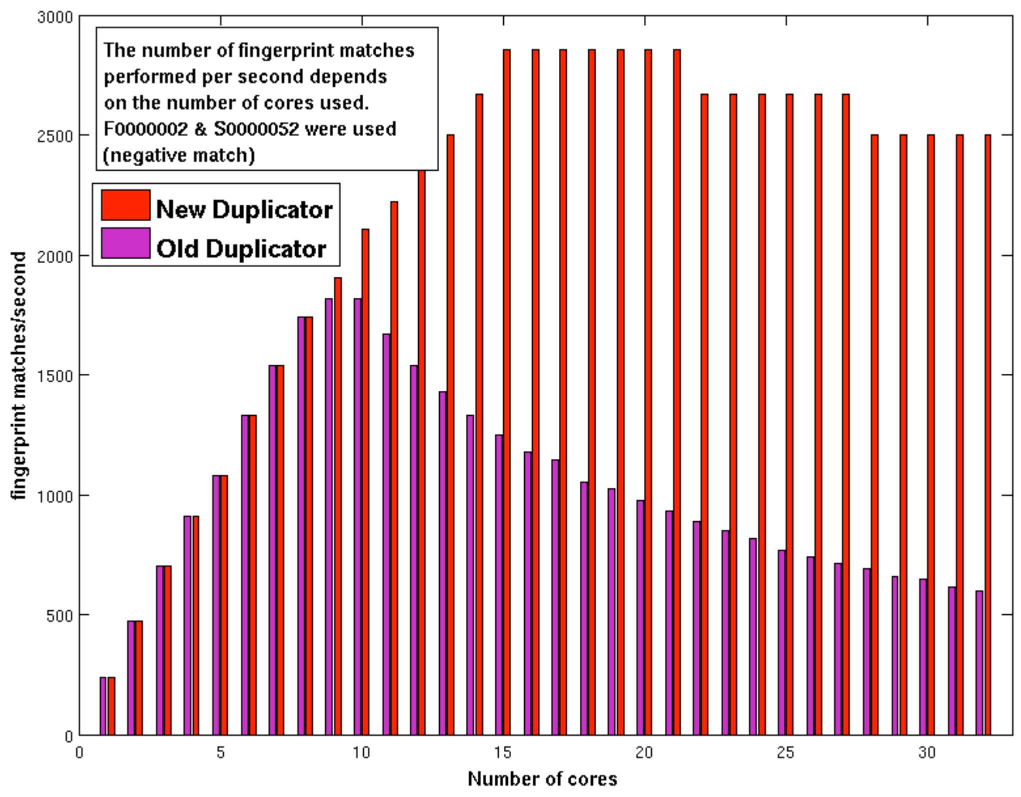

Figure-13. The difference of performance between the original design of the duplicator without commands aggregation and the redesigned one with command aggregation. A bottleneck preventing the experiment to scale linearly up to 32 cores is noticed when using a fingerprint pair having a negative match.

We can see in Figure 13 that the old duplicator (single threaded design) has its performance peak when using 9 cores (the graph is constituted of 9 branches) when using a minutiae pair giving a non-match score.

The old duplicator fully occupied a single core of the computer and no extra computational resources could be allocated to it due to its single threaded architecture. When adding extra branches to the graph, which would translate into adding extra consumers to the duplicator, the new consumers compete within the duplicator with all other consumers for a pool of finite resource of one core that has already reached its maximum. So the average number of matches per second should reach a plateau but as the graph shows, it gradually decreased. This decrease is due to the added overhead of having an extra client for the duplicator. This overhead is taken from the finite pool of computational resource allocated to the duplicator.

The new duplicator reached its improved performance peak when using 15 or 16 cores (with a pair of fingerprints giving a non-match score). The performance decreased gradually in discrete a stair step pattern. This is due to a rounding effect: we measured the time it took to perform 40000 fingerprint matches and measured the performance at the second. The more we sped up our experiments, the more we lost precision in our measures. But the pattern was still clear. Looking at the duplicator at runtime didn't explain why the performance peaked with 16 cores, but we found that the new design of the duplicator has actually been a success, but one that was masked by shifting the bottleneck to the network.

7.5 Network saturation

We decided to look at more parameters and included the network and CPU usage to our graph. First we had a look at what happened when using a minutiae pair giving a match score and we found the network wasn't saturated.

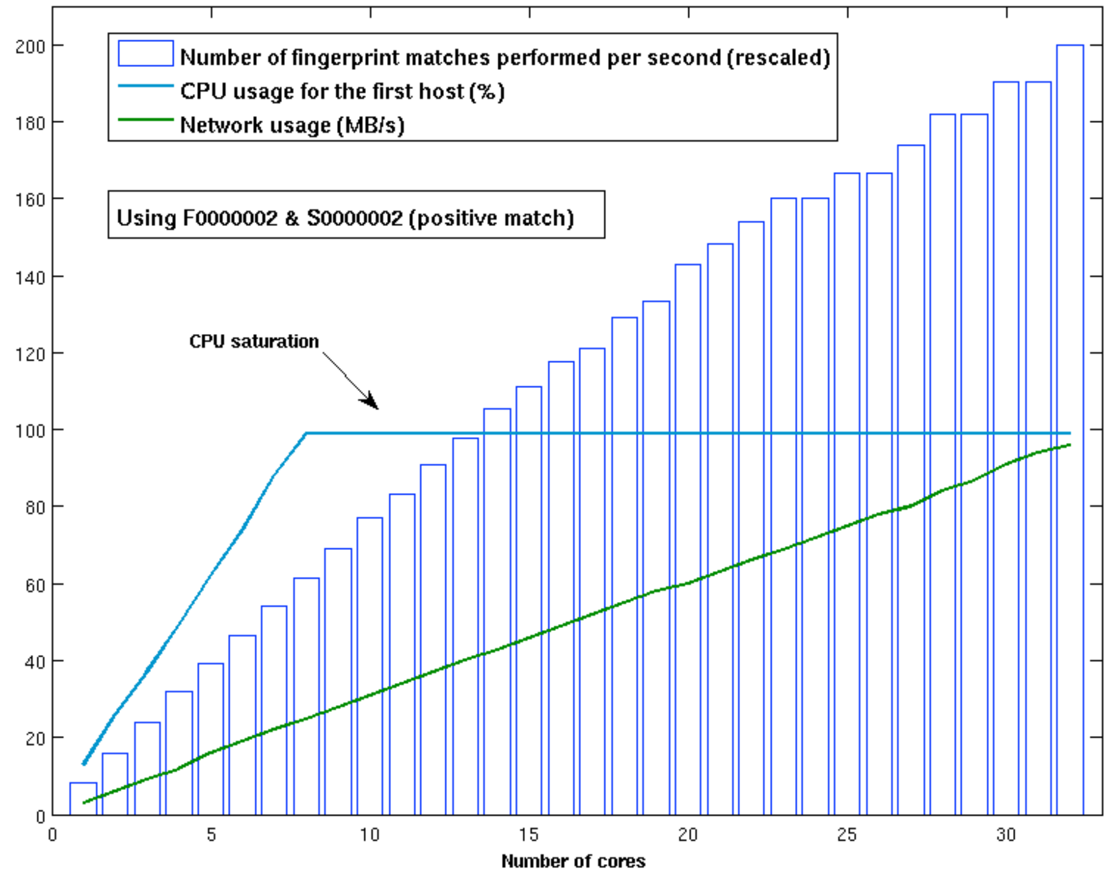

Figure-14. The performance of the new duplicator with commands aggregation in blue in Figure 12 are shown along with the CPU usage of first 8-core node used in the experiment and the network usage of the host storing the templates to match. The linearly increasing network usage data show that we never reach network saturation when using a pair having a positive match. No bottleneck preventing the experiment to scale linearly is noticed here.

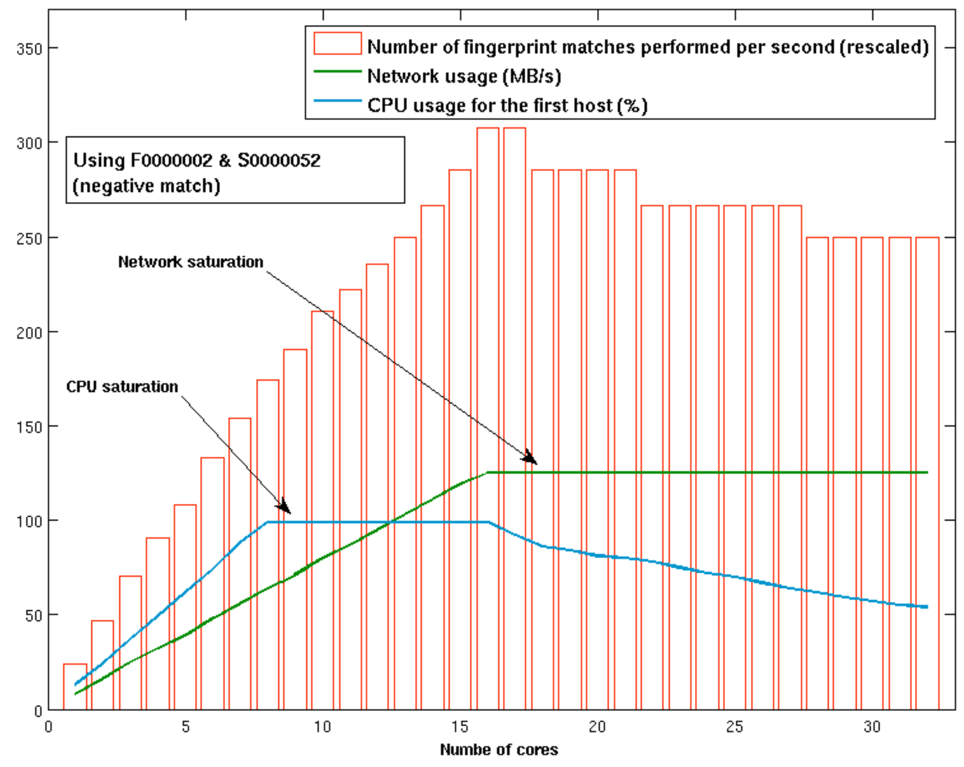

Figure-15. The performance of the new duplicator with commands aggregation in red in Figure 13 is shown along with the CPU usage of first 8-core used in the experiment and the network usage of the host storing the templates to match. The bottleneck preventing the experiment to scale linearly up to 32 cores identified in Figure 13 is explained: commands aggregation combined with a new design of the duplicator shifted the bottleneck to the network.

The experiment that used fingerprints giving a non-match score was more interesting. Figure 15 shows the number of fingerprint matches performed per second by the new duplicator as it is represented in Figure 13, along with the network usage of the host storing the fingerprint templates and the CPU usage (in percent) of the first computer used in the experiment, i.e. the first 8 cores allocated to the graph. Notice that when using 8 cores, which are all the cores available of the first computer entering the experiment, the overall CPU usage of the computer reached 100%. That means that each matcher (each allocated to one core of the computer) is never waits for new data to arrive. Any CPU performance decrease at this point would highlight that some matchers are idle, waiting for a fingerprint template pair to be delivered for matching, such a decrease would point to a network issue.

We noticed that the performance of the first 8 cores saturates at 100% when using 8 branches and started to decrease when using more than 17 cores. We can also see when we allocated 16 or 17 branches to our graph (i.e. we use 16 or 17 computing cores), we reached network saturation by using its capacity at about 125MB/s, which is the theoretical limit of a gigabit Ethernet network.

The data rate basically represents the number of fingerprint templates sent to the array of matchers. So once we saturated the network, the data rate required to keep all matchers of the distributed application busy is at or above the network capacity.

7.6 Data formats and data compression

Reaching network saturation pushed us to look at the fingerprint templates themselves. The API that enabled us to extract templates from images also provided methods to get a serialized representation of the templates. This was very convenient, because the serialized version is not tied to any computer architecture and thus can be transported and reloaded by any kind of computer.

The serialized fingerprint template was returned in a buffer having a fixed size of 16KB. This buffer was then saved as a file. A closer examination of these buffers showed an average space usage reaching at the maximum less than 10% of the buffer. So we saturated the network by sending buffers that were 90% empty. To reduce the minutiae size the representation of the fingerprint templates was moved from INCITS-372 (that was stored in the 16KB buffer) to the internal NIST representation format that was in ASCII. This modification saved network bandwidth, the average template size was reduced from 16KB to an average three to four kilobytes. This reduces network usage, i.e. increase potential scalability. But the focus of this study was the architecture of a distributed model to support extreme scalability in biometric evaluations. The code handling the ASCII version was available, while having an appropriate size for the binary fingerprint was not implemented.

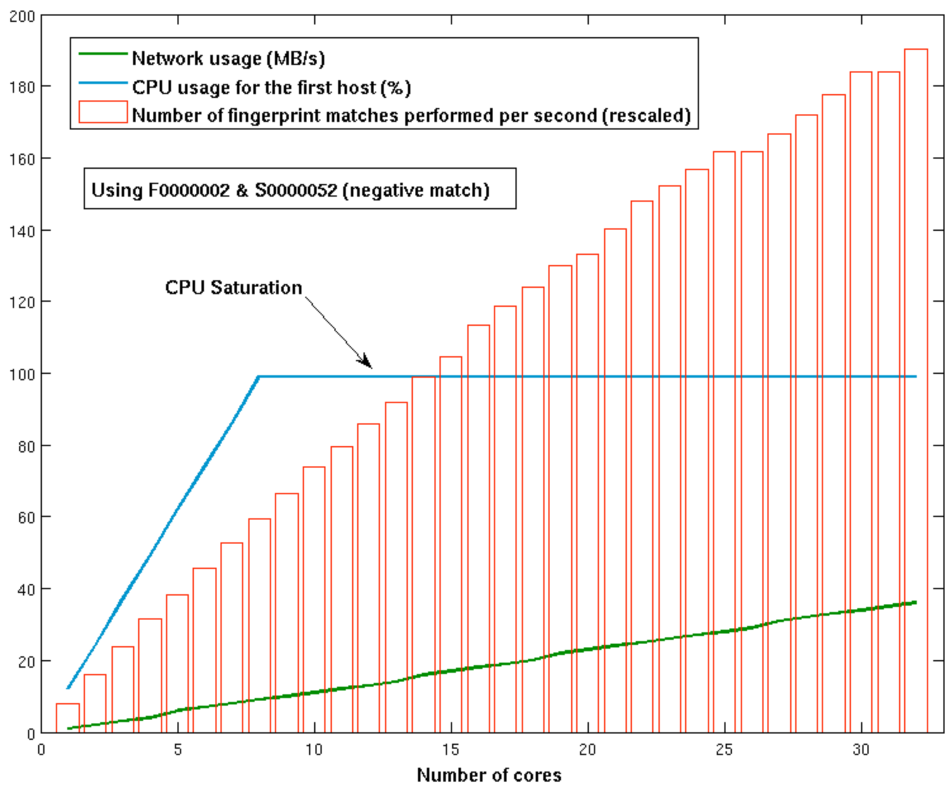

Figure-16. The performance of the new duplicator is shown along with the CPU usage of first 8-core of used in the experiment and the network usage of the host storing the templates to match. We used a different representation of the templates to reduce network usage. The network is not saturated anymore and the experiment scales linearly up to 32 cores when using a fingerprint pair having a positive match.

As Figure 16 shows, a better representation of the fingerprint templates allowed using the network more efficiently and thus bringing templates fast enough to matchers in order to avoid idle time. At that point, we demonstrated that we scaled linearly performing fingerprint verification up to 32 cores.

7.7 Scalability limits

Performance has been measured as the number of fingerprint matches performed per second. The different computers we used to run the matchers provided similar computational capabilities. So we were able to measure the matching rate performed per second per core. Multiplying this number by 8, we get the matching rate performed per computer.

As long as we do not allocate more than one matcher per computational core and we do not saturate the network (i.e. there is no data starvation for the matchers), performances of the application scaled linearly. By using the NIST internal representation for minutiae, we used the network at a rate of 36MB/s out of the theoretical throughput of 125MB/s. Based on these data and that theoretical network rate. We project that this could be scaled up to ~110 cores, assuming the computational capabilities of each new core is similar to those used during the experiment, before saturating the network. We believe the linear performance scaling would plateau or even drop beyond that scale. In our experiment, that would have consisted in adding 10 additional 8-core computers. Fortunately, the minutiae transported to the matcher were in ASCII format, so it was clearly not the most optimized representation. This leaves room for further improvement. Using a compressed binary version of the minutiae could substantially reduce the network usage thus scaling to more computational cores before saturating the network.

7.8 Data load balancing

The application graph shown in Figure 6 has an inherent design flaw. It is not very well suited to run on clusters composed of computers having heterogeneous processors. A pipeline having its matcher allocated to a slower computer, can potentially block a fast one that has its matcher running on a faster computer. This is because all processing pipelines of the graph share the same output flow of the master client to get their commands.

Consumers of this flow receive commands and keep the one that are intended to them, buffering them in a local finite queue for later processing. If the queue runs full, that would prevent the master client from sending new commands because no commands can be lost. It will, as a ripple effect, affect the other pipelines that will not receive commands fast enough and eventually run with a queue almost empty.

In that case, the slowest pipeline over time limits the processing rate on the whole graph. Moreover commands are distributed uniformly among the different pipelines. It would be wiser to balance the load and give more commands to the fast pipelines than to the slow ones. These are clearly not big issues if all computers used in the evaluation are homogeneous. Unfortunately many clusters built and maintained incrementally, adding computers over time, thus making them heterogeneous. When we ran a very long experiment, we were able to witness this behavior: the slowest computers are fully used while the faster one alternate busy time with idle time.

In order to solve this we split the unique flow sending commands of the master client into several flows. The ideal case would consist in having as many flows as pipelines. This approach was not chosen. Instead we chose to have as many flows as computational computers (the ones that run the matcher clients). We did that because:

1. We consider the computational cost per match fixed (averaged on many matches)
2. We assume that each computational core within a host offer the same power.

The original graph was adapted to support the data balancing capabilities but we did not have balancing methods to take advantage of it.

7.8.1 The active load balancing method

A simple PID controller (proportional–integral–derivative controller) was implemented in order to balance the load. Simplistically put, the controller was adjusting the number of commands sent depending on the number of results received to try to homogenize the load among our network. The algorithm was tested under Matlab and provided very good results.

Adding the controller to the system however didn't provide the expected results. We would count for every N commands sent how many results we received from each pipeline during that time. In practice, we either measured that most commands have been processed and therefore most of the results were available, or on the contrary we measured that very few any results had arrived and the OS scheduler allocated time slices too big to have a proper reading for the control loop to stablize. We were left with no proper data to feed our PIV controller. So it was decided to drop this active balancing method and we instead focused our efforts on a passive one.

7.8.2 The passive load balancing method

The passive method achieves balancing through new the architecture of graph combined with the action of "pulling" commands instead of "pushing" them.

In order to implement this behavior, the data buffering within pipelines was reduced to a very low level (the buffering level of flows, expressed as a number of data blocks within a client, is under user control). Data loss cannot be tolerated in such applications. Client internal flow's queue filled up when clients are not able to process data fast enough. When the consumer's queue is full, data will start to accumulate in the queue of the output flow of the producer. When the producer queue filled it will block the producer from enqueuing further data. If this producer is also a consumer, it won't be able to consume the new data because it is blocked trying to send. In that case the client with the slowest processing rate on a pipeline defines the overall processing rate of the whole pipeline. Each client on the pipeline will eventually converge to its processing rate by a ripple effect. In our application, that behavior quickly ripples to the controller client that ends up being blocked from sending commands.

In the active version, the controller client distributed commands, one after another, to all the available command flows, effectively pushing them. That was done within one context (i.e. within one thread). With the passive method, each command flow has its own context, i.e. the action on one flow cannot block the action of the others. So the only limitation that remains to send commands for each flow is to have room to send it in the pipeline. So that change allowed us to correlate the sending rate of commands with the processing speed rate of the pipeline only. The last item missing here was to get rid of the fixed slice of load allocated a-priori on each processing pipeline. This was done by putting commands in a global pool accessible by all command flows, thus allowing a flow serving a faster pipeline to pull more commands from the pool than a flow serving a slow one. As opposed to the active/predictive method, the passive method was a very successful way to efficiently distribute the load on the graph. This pull driven approach will perform equally well for a variety of probe transactions since actual processing time is used to gate the pulling of further commands.

We found that that:

- Each processing node finished processing data at approximately the same time.
- Each processing node used its core at nearly 100% during the whole simulation.
- The ratios of the number of commands processed by the different hosts correlates almost perfectly the ratios of the CPU computational capabilities of the different hosts.

Another big advantage of the passive method over the active one is its capability of starting an effective balancing process immediately and to aim at the perfect balancing. The predictive simulation of the active method, using a feedback mechanism to adjust, was an iterative process that tends to overshoot and undershoot, with decreasing intensity until equilibrium was reached. This learning process takes some time. If the condition of one computational host changes (e.g. some of its resources are taken by other task), we will again by dichotomy (overshoot, followed by undershoot) find a new equilibrium. The system is not in an optimal state when stabilizing; therefore losing time compared with the passive method that has nearly instantaneous adjustment and reaction time.

8 Conceptual design for parallel matching on large scale clusters

Distributed processing architectures are limited by processor, memory, network, and disk storage bandwidth. When scaling the application, one of these bandwidth issues will be eventually saturate either I/O or computational resources. As a result, any scalable architecture should be:

- Lightweight
- Network efficient
- Takes the storage tiers into account in organizing data access
- Optimally scheduled
- Distributed

The NDFS-II is well fitted as a middleware to support scalable architectures applicable to fingerprint matching. It is a peer-to-peer architecture and so has no single point of failure. It has been developed in C++ to avoid compromising performance levels. Flows (supported by the duplicator) are optimized for network transport: if several clients within one computer consume the data from a common provider, the data are only transported once over the network and only duplicated when they reached the destination host.

8.1 Limitations due to network technology

Our experiments showed that we saturated the network quickly before optimizing the buffer size for each actual fingerprint templates. When we introduced the ASCII representation of the fingerprint templates, we were able to fully utilize 32 cores by using a data rate of about 36MB/s out of theoretically maximum 125MB/s available for use in one gigabit Ethernet. This suggests that using this representation for templates, we can only scale efficiently up to ~110 cores (or about fourteen 8-core computers). After that, the network would not be able to transport data fast enough to keep the matchers busy using one gigabit Ethernet technology. Medium or large clusters comprised of a thousand to many thousands of compute cores will need to be optimized in several ways. This can include higher bandwidth interconnect, such as the 40 gigabit Infiniband standard for networking might allow a few thousand compute cores to be used effectively with the optimization of the fingerprints.

8.2 Clusters with heterogeneous processors

As our experiments highlights, one cannot to assume that we have a homogenous cluster of processor cores. Even a homogenous cluster is not insulated from performance unbalance. Performance of a single host could be reduced by partial hardware failure, overconsumption of RAM, bugs in software, etc. In such cases, a single host would have less computational power than any other one and as we established in section 7, reduce the performance of the whole cluster.
Any software solution providing distribution for this kind of applications should ensure that:

- A slow computational node cannot be allowed to slow down faster ones by its processing rate.
- The load is distributed according to the pull mechanism run from remote hosts, with no pre-assumption of processor node homogeneity.

8.3 Bandwidth considerations

Our experiments highlight that by using an uncompressed ASCII representation of minutiae it would take about only ~110 cores to saturate the existing one gigabit network. Processors are still following Moore's law in doubling the number of transistors every eighteen months so the number of cores per processors is still rising. So the main future bottleneck will be the network transport itself. A few modern computers, with up to sixteen processors, are required to build a cluster with many cores, so attention should be focused on the network bandwidth. Certainly, compressed formats should be used to transport data from storage to processing units. Computational power, which is in abundance, can be allocated for compression and decompression of data in order to minimize the load on the networks with bandwidth in short supply. Local caching to Solid State Disks (SSD's) on processing hosts should not be overlooked to reduce the load put the network. Still, emerging technologies such as 40 gigabit Infiniband networking should be investigated for incorporation in massively parallel processing clusters.

8.4 Using flows to reduce network bandwidth

Data are transported using flows in NDFS-II applications. There is no direct connection between data provider(s) and consumer(s). Duplicators are acting as middlemen to transport data. This brings several advantages such as the management of connection/disconnection of clients. But the main one here is to avoid data duplication. If two consumers running on the same host computer subscribe to the same flow produced on a remote machine, the data will be transported only once over the network and duplicated once they have reached their destination host. This can save tremendous network bandwidth in an application where many consumers subscribe to the same flow.

8.5 Distributing the data

Modern applications tend to increase their data use. Having only one data host can be a bad architectural choice because it tends to create a bottleneck. Data duplication could potentially solve this issue but unfortunately it is not always possible depending on the resource available and the amount of data to store. One can use a distributed file system to solve this issue but in that case there is still a central service handling the file system therefore creating a potential bottleneck and a single point of failure. An alternative solution would be to spread all the data on different computers and use their native file system to store the data. This is the approach used in the applications presented in this report. Rather than using services such as nfs/samba to access data, clients that read the data on the disk and send them in flows would be spawned where data are located. No need for an extra protocol or layers in that case. It also has the advantage of avoiding a central file server. The option of SSD's could be very important as the number of processors increases. If each matcher process a different area of a rotating disk there can be excessive seek times when data on different tracks is needed, which would not be the case on a random access SSD.

8.6 Local caches

Some applications are composed of client nodes that need to access data several times when they are too large to store in RAM. In that case, a retransmission over the network might be suboptimal from a performance viewpoint. A cache on a local hard drive is then more efficient. This local cache drive can be a bottleneck in its own right, particularly if slow mechanical data seeking is required. SSD drives offer faster read performance than mechanical drives, particularly when many data files or databases must be accessed concurrently from the drive.

8.7 Collecting results

In most standalone applications, command files tend to be processed in order of receipt and results gather in the same way through an iterative process therefore removing the needs to have methods reorganizing results. A distributed application introduces weaknesses that are not present in standalone versions including:

- Out of order result arrivals that require a whole flow to be reassembled.
- Remote hosts can fail and therefore their results are lost if redundant state is not maintained across the data flow graph.

The distributed version must be able to reorganize results so results of different evaluations can be compared against each other. This can be and is done in our distributed version by associating each command with an incremental unique ID. This ID is propagated with the result in order for the controller client to re-order results it receives.

The distributed version must also be able to:

- Perform an analysis of the state of the current evaluation to establish which results haven't been received when a problem is detected.
- Regenerate an evaluation containing only the missing results to complete the one that has been formerly executed.

This capability was not implemented in the version we used due to its nature: an experiment.

8.8 Implications for the next generation SDK API

The distributed version provided excellent results to support Proprietary Fingerprint Template (PFT) Evaluation [8]. The data flow middleware is an effective tool for PFT because in that evaluation data can freely be moved among computers. Fingerprint images can be moved, the templates that are derived from these images can be serialized for storage or transports and results are not proprietary but rather follow specifications.

Evaluation of Latent Fingerprint Technologies (ELFT) [9] is different. It is a study of latent fingerprint identification (one to many search) rather than verification (one to one match). In PFT, each command contains two references to two templates. These templates differ in each command.

ELFT first builds a gallery of templates to probe. This gallery is built locally within a computer. Once the gallery is built, it would be convenient to transfer it to other computers so they can probe this gallery. Unfortunately it is not possible to do that with the SDK supporting ELFT because the vendor gallery is just an object in memory under vendor scope. Adding serialization methods for the gallery in the SDK would permit to distribute the gallery and therefore gain tremendous amount of time to perform the evaluations.

Vendors may not wish to provide a serialized version of their gallery that can potentially give some hints on the proprietary underlying technology used. Thus, additional serialization methods in the SDK API to provide encrypted galleries with serialization and encryption methods under vendor control may become important. Updating the SDK in that manner could open the door to high performance distribution for ELFT evaluation performed by NIST.

9 Conclusions

Experiments in parallel fingerprint matching have been successful in extending the number of processors we can fully utilize from ~10 up to ~100 before the conventional 1gigabit Ethernet is saturated. Future work could include the study of an advanced high bandwidth networking technology used in High Performance Computing (HPC) clusters at various national laboratories. We have now constructed a test bed for this purpose with both 1 gigabit Ethernet, and 40 gigabit Infiniband network cards and switching fabric.

The NDFS middleware technology can be applied as it is for PFT, but ELFT would require modification in the SDK. We also highlighted the potential bottleneck that arises when scaling a distributed fingerprint evaluation. Particular attention should be paid to the network data transport issues; the gigabit link clearly appears to be the main bottleneck in modern cluster. Thus, the network will be utilized most efficiently by implementing some sort of source compression, compressed transmission, and decompression at the destination machines.

References

[1] NIST Biometric Image Software. (http://www.nist.gov/itl/iad/ig/nbis.cfm)

[2] NIST Special Database 14 Version 2 of April 2002. (http://www.nist.gov/ts/msd/srd/nistsd14.cfm)

[3] NDFS-II documentation. (http://www.nist.gov/smartspace)

[4] Overview of ACE. (http://www.cs.wustl.edu/~schmidt/ACE-overview.html)

[5] *Middleware and Metrology for the Pervasive Future.* Antoine Fillinger, Imad Hamchi, Stéphane Degré, Lukas Diduch, Travis Rose, Jonathan Fiscus and Vincent Stanford. IEEE Pervasive Computing Mobile and Ubiquitous Systems. Vol. 8, num. 3, page 74-83, *July-September **2009***. (http://nist.gov/smartspace/downloads/IEEEPervasiveNDFS_II.pdf)

[6] *A Data Flow Implementation of Agent-Based Distributed Graph Search.* Imad Hamchi, Antoine Fillinger, Mathieu Hoarau, Lukas Diduch and Vincent Stanford. Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS), Cambridge, Massachusetts, USA, *November 2009.*

[7] *Two Finger Matching With Vendor SDK Matchers*. NIST IR 7249. C. Watson, C. Wilson, M. Indovina and B. Cochran. NIST Interagency Report 7249. *July 2005*. (http://fingerprint.nist.gov/SDK/ir_7249.pdf)

[8] Proprietary Fingerprint Template Evaluation II – PFT II. *2010*. (http://www.nist.gov/itl/iad/ig/upload/PFTII_Evaluation_Plan.pdf)

[9] ELFT Phase II - An Evaluation of Automated Latent Fingerprint Identification Technologies. NIST IR 7577. *April 2009.* (http://fingerprint.nist.gov/latent/NISTIR_7577_ELFT_PhaseII.pdf)

10 Appendix

Figure 18 highlights relations between the figures presented in this report. It is a visual add-on that helps to understand the efforts reported in these report.

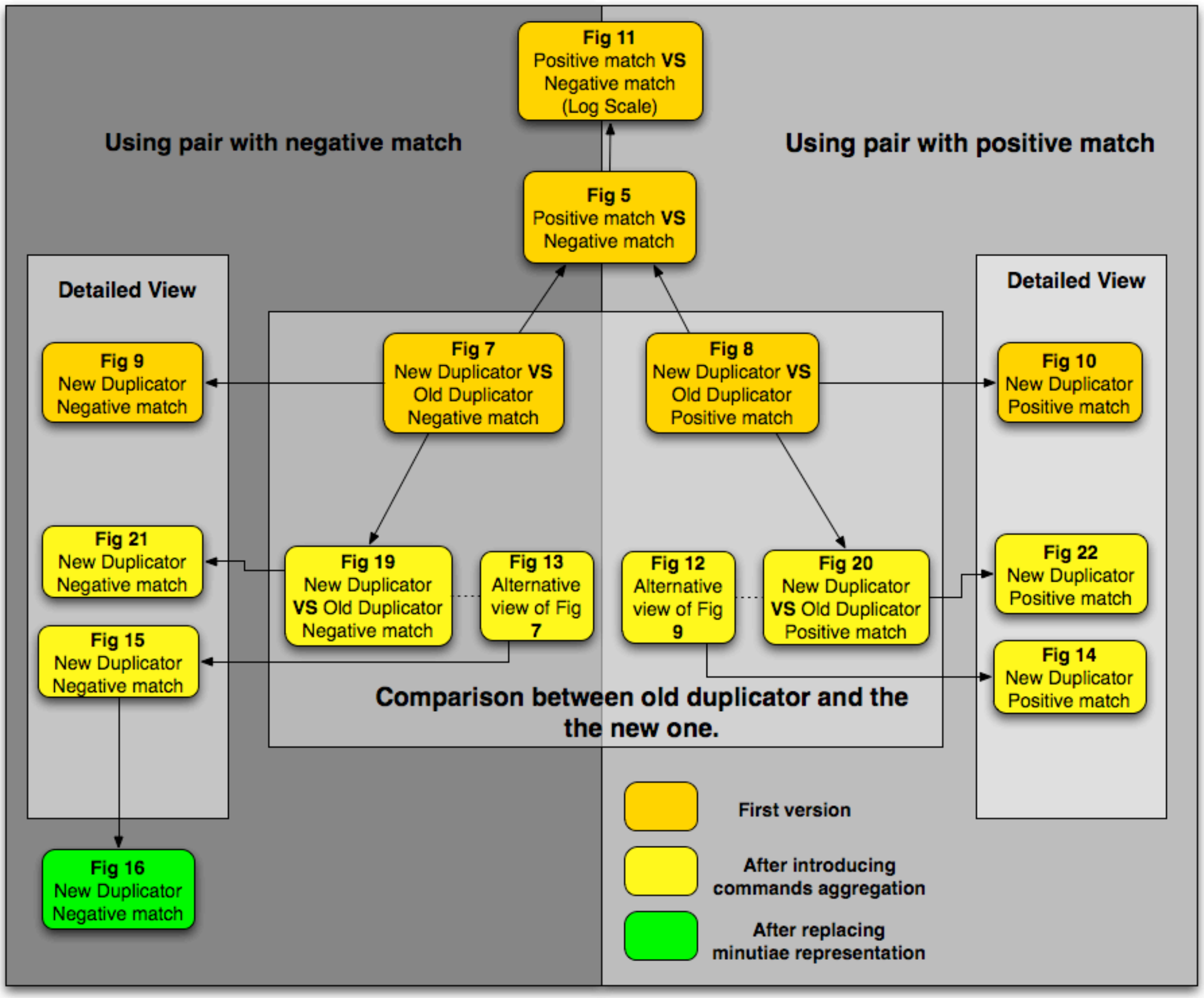
Fig 11
Positive match VS Negative match (Log Scale)
Using pair with negative match
Using pair with positive match
Fig 5
Positive match VS Negative match
Detailed View
Detailed View
Fig 9
New Duplicator Negative match
Fig 7
New Duplicator VS Old Duplicator Negative match
Fig 8
New Duplicator VS Old Duplicator Positive match
Fig 10
New Duplicator Positive match
Fig 21
New Duplicator Negative match
Fig 19
New Duplicator VS Old Duplicator Negative match
Fig 13
Alternative view of Fig 7
Fig 12
Alternative view of Fig 9
Fig 20
New Duplicator VS Old Duplicator Positive match
Fig 22
New Duplicator Positive match
Fig 15
New Duplicator Negative match
Fig 14
New Duplicator Positive match
Comparison between old duplicator and the the new one.
Fig 16
New Duplicator Negative match
First version
After introducing commands aggregation
After replacing minutiae representation

Figure 17 Visualizing the relation between the figures presented in this report can ease the comprehension.

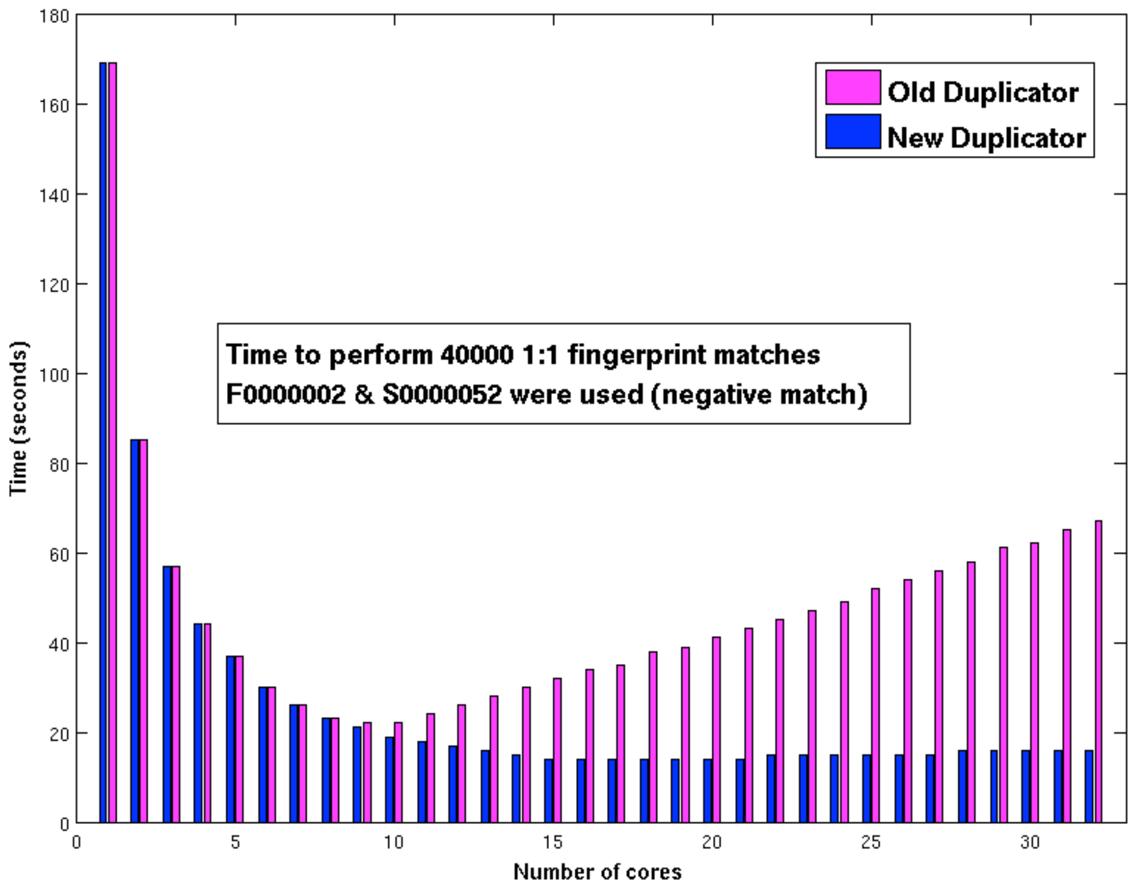

Figure 18 The difference of performance between the original design of the duplicator without commands aggregation and the redesigned one with command aggregation. A bottleneck preventing the experiment to scale linearly up to 32 cores is noticed when using a fingerprint pair having a negative match. This is an alternative view of Figure 13 that displays performance in match per second instead of the total time to run the experiment.

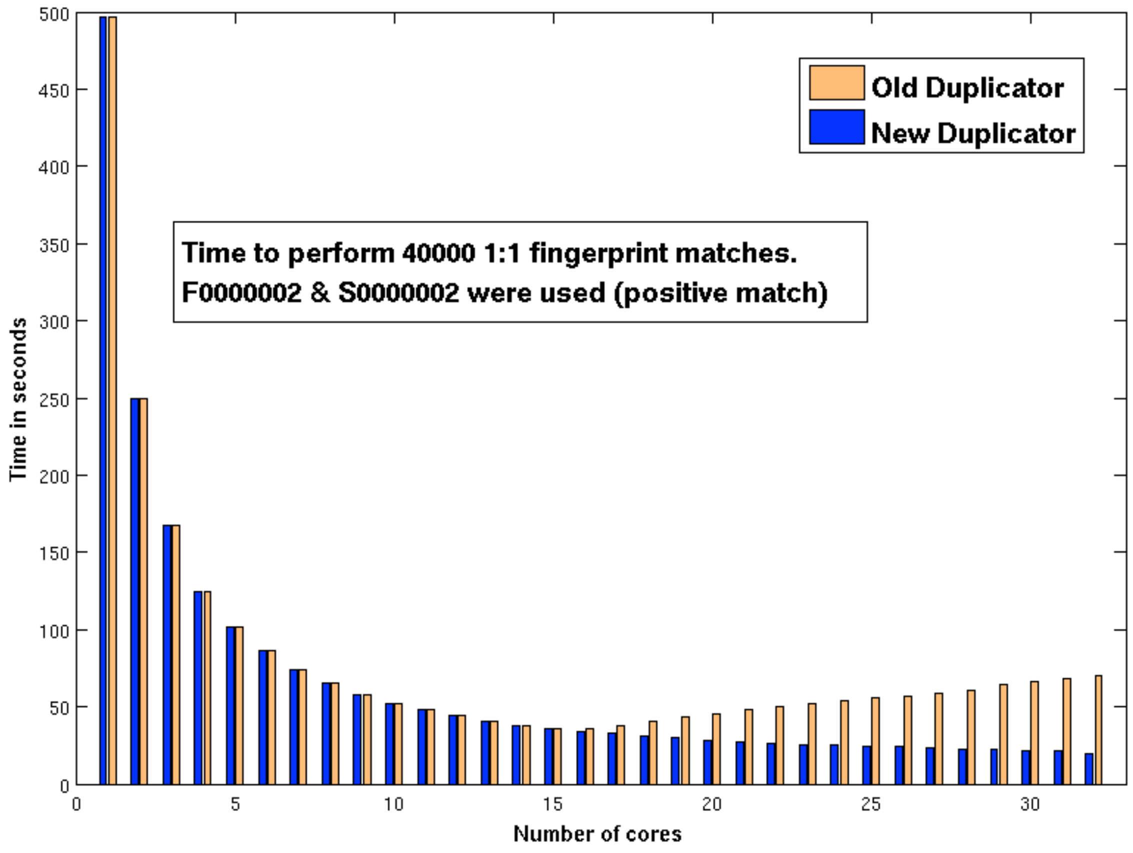

Figure 19 The difference of performance between the original design of the duplicator without commands aggregation and the redesigned one with command aggregation. No bottleneck preventing the experiment to scale linearly up to 32 cores is noticed when using a fingerprint pair having a positive match. This is an alternative view of Figure 12 that displays performance in match per second instead of the total time to run the experiment.

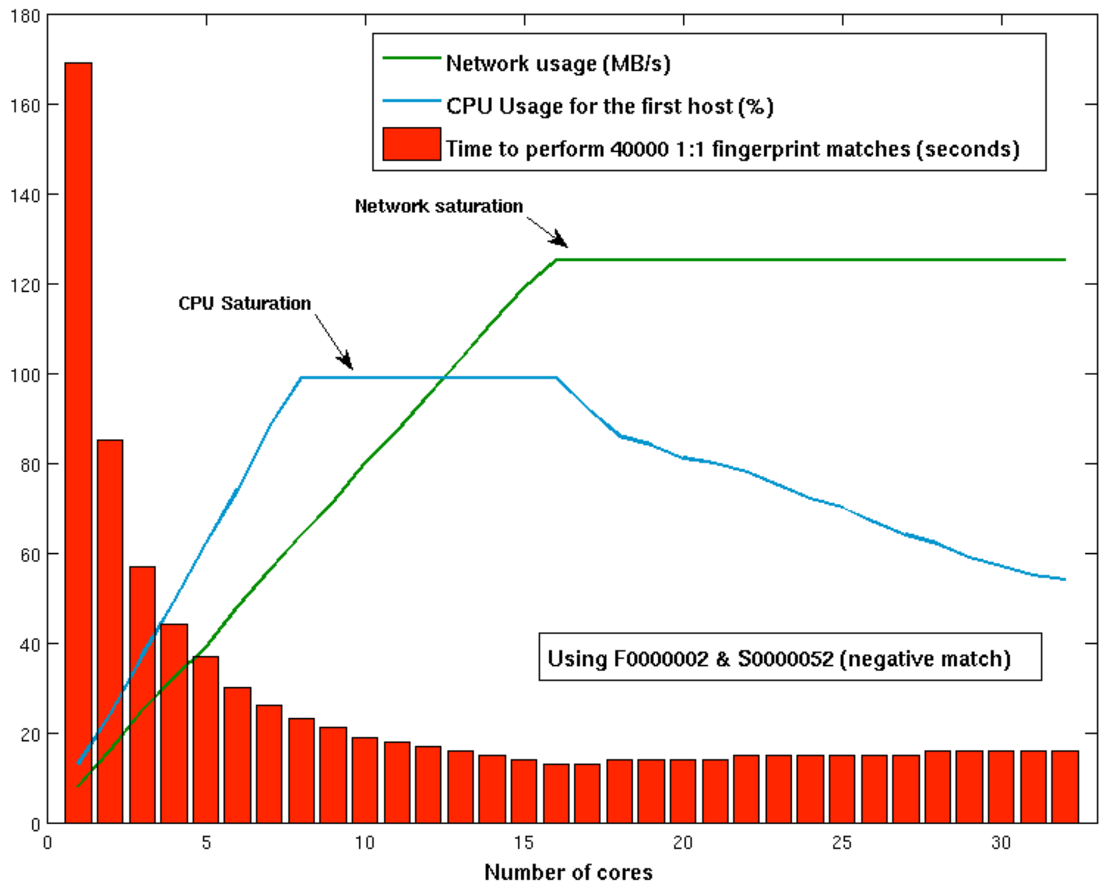

Figure 20 The performance (with commands aggregation) of the new duplicator in red in Figure 13 are shown along with the CPU usage of first 8-core used in the experiment and the network usage of the host storing the templates to match. The bottleneck preventing the experiment to scale linearly up to 32 cores identified in Figure 13 is explained: commands aggregation combined with a new design of the duplicator shifted the bottleneck on the network. This is an alternative view of Figure 15 that displays performance in match per second instead of the total time to run the experiment.

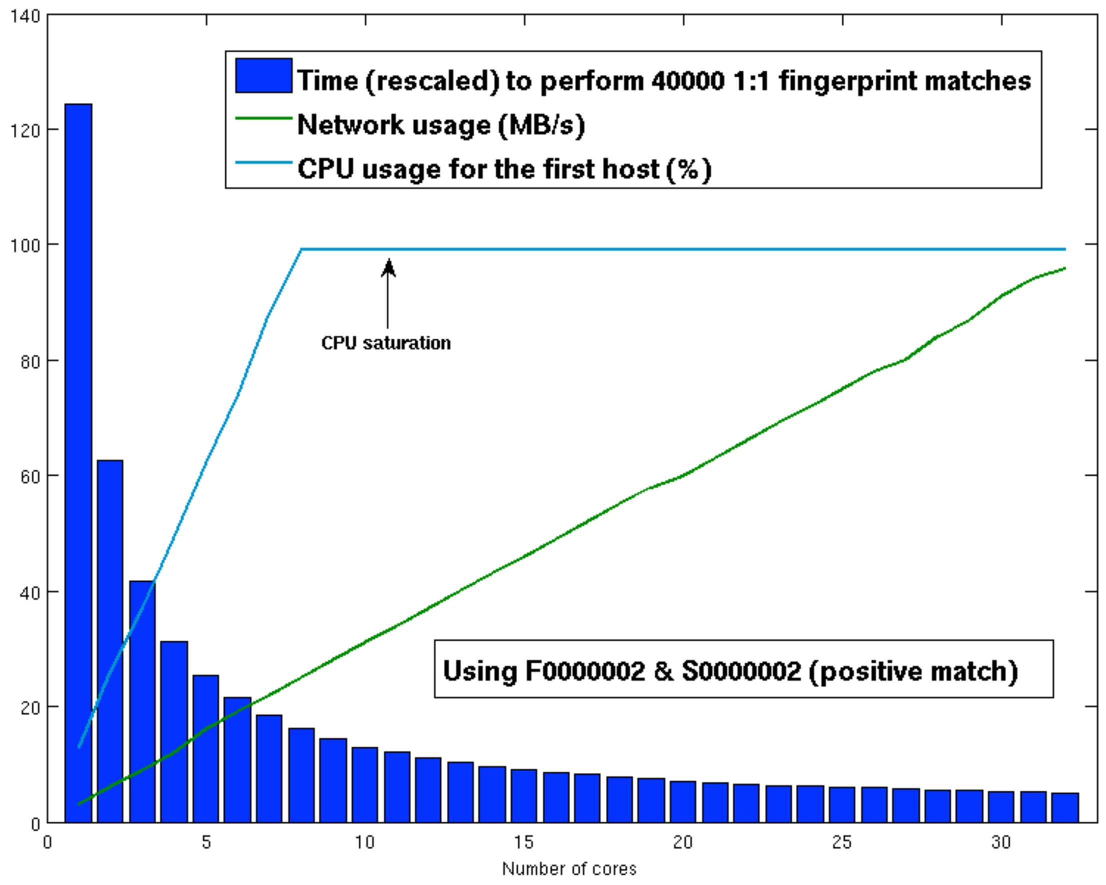

Figure 21 The performance (with commands aggregation) of the new duplicator in blue in Figure 12 are shown along with the CPU usage of first 8-core used in the experiment and the network usage of the host storing the templates to match. It can be noticed that we never reach network saturation when using a pair having a positive match. No bottleneck preventing the experiment to scale linearly is noticed here. This is an alternative view of Figure 14 that displays performance in match per second instead of the total time to run the experiment.