ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

MEMORANDUM PRESENTED TO
L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
A MASTER'S DEGREE IN ELECTRICAL ENGINEERING
M. Eng.

BY
Éric THIBODEAU

PROFILING AND OPTIMIZING K-MEANS ALGORITHMS IN A BEOWULF CLUSTER ENVIRONMENT

MONTREAL, DECEMBER 21 2009

PRESENTATION OF THE JURY

THIS MEMORANDUM HAS BEEN EVALUATED

BY THE FOLLOWING BOARD OF EXAMINERS

M. Tony Wong, Memorandum Supervisor
Département de génie de la production automatisée à l'École de technologie supérieure

M. Roberet Sabourin, Memorandum Co-supervisor
Département de génie de la production automatisée à l'École de technologie supérieure

M. Guy Gauthier, President of the Board of Examiners
Département de génie de la production automatisée à l'École de technologie supérieure

M. Eric Granger
Département de génie de la production automatisée à l'École de technologie supérieure

THIS MEMORANDUM WAS PRESENTED AND DEFENDED BEFORE A BOARD OF

EXAMINERS AND PUBLIC

ON NOVEMBER 24 2009

AT ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

PROFILAGE ET OPTIMISATION DE L'ALGORITHME DU K-MEANS DANS UN ENVIRONMENT DE GRAPE DE CALCUL DE TYPE BEOWULF

Éric Thibodeau

RÉSUMÉ

L'algorithme d'agglomération statistique K-means sert à classer des bases de données non libellées en K groupes. Faisant partie de la fonction d'évaluation d'un Algorithme Écolutionnaire (AE), l'optimisation de ce dernier est devenu un point d'intérêt. Malgré les multiples approches proposées pour son optimisation et sa parallélisation, très peu de recherche s'est attardée aux questions entourant la performance et l'efficacité parallèle des implantations. Dans la plupart des cas, les descriptions entourant l'environnement d'exécution demeurent opaques et la présentation précise de profiles d'exécution est souvent absente.

Nous pallions à ces lacunes en présentant une description détaillée de deux environnements, le grappes de calcul Beowulf et les machines parallèles de type Symmertric Multi-Processors (SMP). Une combinaison de modèles théoriques et empirique sert ensuite d'étalon dans la mesure de performance du K-means dans ces environnements. Étant la nécessité d'une expertise pluridisciplinaire, une utilisation détaillée de la suite d'outils Tuning and Analysis Utilities (TAU) est présentée pour simplifier la tâche du profilage de code parallèle. Couplée aux compteurs haute précisions fournies par l'interface Performance Application Programming Interface (PAPI), nous présentons une approche «grey box »ayant permis de muter une implémentation parallèle maître-esclave du K-means vers une version hautement efficace utilisant le paradigme d'îlots de calculs. Les optimisations sont guidées grâce à l'utilisation des modèles théoriques et empiriques que nous avons obtenus.

Notre travail révèle que l'optimisation de programmes parallèles relève de bien plus qu'un équilibre entre calcul et communications. Nous révélons les impacts négatifs de l'utilisation de bibliothèques de fonctions mathématiques ainsi que de certaines versions des bibliothèques de communications. Un profile d'exécution de haute précisions a permis d'établir que la représentation et le pré-traitement des données peuvent s'avérer être plus coûteux que le calcul et les communications combinés.

PROFILING AND OPTIMIZING K-MEANS ALGORITHMS IN A BEOWULF CLUSTER ENVIRONMENT

Éric Thibodeau

ABSTRACT

The K-means algorithm is a well known statistical agglomeration algorithm used to sort a database of unlabeled items into K groups. As part of the fitness function of an Evolutionary Algorithm (EA), the optimization of the K-means algorithm has become a point of great interest. Although many approaches have been proposed for its parallelization and optimization, very few address the question of scalability and efficiency. In most cases, the description of the execution environment remains opaque and precise profiles of the program are mostly absent. Performance and efficiency issues are quickly relegated to communication issues.

We address these deficiencies by presenting a detailed description of two parallel environments, the Beowulf style clusters and the Symmetric Multi-Processors (SMP) parallel machines. A mixture of theoretical and empirical models were used to characterize these environments and set baseline expectations pertaining to the K-means algorithm. Due to the necessity of a multidisciplinary expertise, a detailed use of Tuning and Analysis Utilities (TAU) is provided to ease the parallel performance profiling task. Coupled with the high precision counter interface provided by Performance Application Programming Interface (PAPI), we present a *grey box* method by which a parallel master-slave implementation of the K-means is evolved into a highly efficient island version of itself. Communications and computational optimization were guided by prior theoretical and empirical models of the parallel execution environment.

Our work has revealed that there is much more to parallel processing than the simple balance between computation and communications. We have brought forth the negative impact of using mathematical libraries for specific problems and identified performance issues specific to some versions of the same series of Message Passing Inerface (MPI) libraries. High precision profiling has shown that data representation and processing can be a more significant source of scalability bottleneck than computation and communications put together.

TABLE OF CONTENT

Page

INTRODUCTION 1

CHAPTER 1 HARDWARE CHARACTERIZATION 4
1.1 Basic Computer Architecture 5
1.1.1 The Control Unit and Arithmetic Logic Unit 5
1.2 Caching in on The Main Memory 6
1.2.1 Accessing The Main Memory 7
1.2.2 Cache Size and Contention 11
1.2.3 Processor Performance 16
1.3 Communications 17
1.3.1 Bandwidth 18
1.3.2 Latency 19
1.3.3 The HyperTransport Interconnect 23
1.3.4 Benchmarking Network Communications 24
1.3.5 Theoretical and Empirical Model 27
1.4 Input/Output and Storage 28
1.4.1 Local Versus Remote Storage 29
1.5 Discussions 30

CHAPTER 2 THE PROFILING TOOLS 32
2.1 Black, Grey and White Box 33
2.2 Sequential Profiling: Use Of `gprof` 34
2.3 The Itch Of Measuring Time 36
2.3.1 PAPI: Time To Scratch Below The Surface 36
2.4 Tuning and Analysis Utilities (TAU) 39
2.4.1 Configuring TAU 40
2.5 Profiling the Source Code 41
2.5.1 Automatic Code Insertions 42
2.5.2 Semi-Automatic Code Insertions 42
2.6 Executing the Profiled Code 43
2.6.1 Selecting The Profile Depth 44
2.6.2 Selecting The Desired PAPI Events 44
2.6.3 Controlling The Data Flow 45
2.6.4 Storing The Data 45
2.7 Paraprof and PerfExplorer: The Profiling Graphical User Interfaces 46
2.7.1 The Paraprof Profile Viewer 47
2.7.2 The PerfExplorer Performance Analyzer 51
2.7.3 Application Speedup 52

2.7.4 Application Parallel Efficiency 55
2.7.5 Runtime Breakdown 56
2.7.6 Views 57
2.8 Discussions 60

CHAPTER 3 CASE STUDY: PARALLEL K-MEANS ALGORITHM ANALYSIS 61
3.1 The Sequential k-means Algorithm 62
3.1.1 Empirical Evaluation of the Algorithm 62
3.2 The Parallel K-Means Algorithm 66
3.2.1 First, Divide: The Segmentation Strategies 66
3.2.1.1 Strided Segmentation 66
3.2.1.2 Blocked Segmentation 67
3.2.1.3 Hardware Considerations: Load Balancing 67
3.2.1.4 Hardware Considerations: Physical Limitations 68
3.2.2 Then Tell Everyone: Communications 69
3.3 And Conquer: Master-Slave Model 69
3.3.1 Master-Slave Communications 70
3.3.2 Master-Slave Empirical Modelization 73
3.4 Or, Invade: Synchronous Island Model 79
3.4.1 Optimizing the code 82
3.4.2 Island Communications 83
3.4.2.1 Overlapping Communications and Computation 83
3.4.2.2 Less Talk, More Work 85
3.5 Optimization of Input/Output (I/O) Routines 87
3.6 Computational Optimizations: Coding for High Performance Computing (HPC) .. 88
3.6.1 Compiler Directives 89
3.6.2 Mathematical Libraries Versus Code 89
3.6.3 Using Single Instruction Multiple Data 90
3.6.4 Loop Optimizations 91
3.6.5 Basic Linear Algebra Subroutines (BLAS) Libraries 96
3.6.6 Comparing All Approaches 99
3.7 Looking at the Global Picture 101
3.8 Discussions 105

CONCLUSION 106

APPENDIX I THE GNU C Compiler (GCC) 111

APPENDIX II COLLECTION OF COMMANDS 116

APPENDIX III MACHINE DESCRIPTIONS 117

APPENDIX IV SOURCE CODE 125

BIBLIOGRAPHY 141

LIST OF TABLES

Page

Table 2.1 Black, Grey and White Box definitions. 33

Table 2.2 Black, Grey and White Box capabilities for the presented tools. 60

Table 3.1 Per bottleneck optimization recommendations. Prior profiling to identify the applicability of these approaches is primordial..... 109

LIST OF FIGURES

Page

Figure 1.1 Multi-processor memory access strategies for both Advanced Micro Devices (AMD) and Intel processors. AMD possesses Non Uniform Memory Access (NUMA) characteristics while Intel's implementation is essentially Uniform Memory Access (UMA). 9

Figure 1.2 These schematizations of the AMD Opteron Dual Core processors (800 series) and the Intel Core 2 Duo processors illustrates how the two core variants access Dynamic RAM (DRAM). In both cases, the Memory Management Unit (MMU) possesses dual channel connectivity to DRAM for link bandwidth aggregation. 10

Figure 1.3 Cache memory behavior on an Intel $Q6600$ (4 cores). Execution time characteristics are illustrated in (a). Cache usage zones are identified in (b). 14

Figure 1.4 Cache memory behavior on AMD Opteron 800 series based processors using 14 cores of a SUN SunFire $x4600$. Execution time characteristics are illustrated in (a). Execution zones are identified in (b). 15

Figure 1.5 Execution time comparison between Intel's $Q6600$ and AMD's Opteron 885 processors. The concurrent process count is in parenthesis. The raw computing power of the $Q6600$ outperforms the Opteron 885 for four processes. The case of 14 concurrent processes is presented to demonstrate the proportionally small impact of their simultaneous execution. 17

Figure 1.6 LAM-MPI outperforms OpenMPI for any TCP/IP communications. The non-linearity are noted around the Maximum Transmission Unit (MTU) barriers of 1500bytes. 20

Figure 1.7 The round trip communication times using MPI libraries surrounding the start up times in (a) and the MTU in (b). Since these are round trip figures, all values have to be halved when considering asymmetric communication patterns. 21

Figure 1.8 The communications latency is affected by the Central Processing Unit (CPU) frequency and network topology. Higher frequency

clearly renders lower latency and the addition of a hop between two hosts (denoted as Cross-Switch) adds significant delays. 22

Figure 1.9 Comparing OpenMPI versions 1.1 and 1.2 on HyperTransport by varying the message size passed to the `mpptest` micro-benchmark. The 1.1 implementations had performance issues characterized by a sudden jump in communication times around packet sizes of 1000bytes. 24

Figure 1.10 The Tyan VX50 interconnection strategy for 8 processors using HyperTransport (HT). This twisted ladder topology provides for an average 1.5 hop between processors and their farthest memory pages. ... 25

Figure 1.11 MPI call types and their impact on the communication times. Synchronous (sync) communications outperform both asynchronous (async) and persistant ones as the processor is dedicated to performing the communication task in that specific case. 26

Figure 1.12 Comparing the general theoretical communications model with empirical values for a $100BaseT$ Ethernet network. Results from `netpipe` are slightly higher than `mpptest`, indicating there might be additional overhead to his test suite. The theoretical value bases its t_s on results from `mpptest`, thus biasing it to be closer to that tool's results. An arrow is inserted at 188bytes as a point of reference for a vector of 47 floats, a unit which comes in handy in our case study. 28

Figure 1.13 An example of proportional breakdown of each task's contribution to the execution time for the Parallel Vector Quantization (PVQ) implemented using a textual database (described in Chapter 3) and traversing its entirety at initiation. The loading of the data is performed by the `load_samples()` function and represents a significant portion of the total execution time. 29

Figure 2.1 Output listing from `gprof -brief -p vq`. The columns describe the following metric for each function (each line): *% time* is the proportion of total execution time, *cumulative seconds* is the **inclusive** execution time, *self seconds* is the **exclusive** time, calls is the total count. *Self* and *total s/call* are for the **inclusive** and **exclusive** time per call. Finally, the last column holds the function name. .. 35

Figure 2.2 The annotated source code as per the use of `gprof -A vq`. Only the two most called functions form the source code are presented. 36

Figure 2.3 The program call graph. This call graph draws the execution path of this simple program. Each box represents a function and the arrows indicate the call sequence. The percentages indicate the *inclusive*, or cumulative, time as one walks down the graph. Exclusive times are indicated in parenthesis. 37

Figure 2.4 A sample use of `gprof2dot` to generate a `dot` file to be interpreted by Graphviz. The information is generated by `gprof`, then piped into `gprof2dot.py`, which itself pipes into the `dot` interpreter to generate the `call-graph.pdf` file. 37

Figure 2.5 The PAPI implementation scheme. Adapted from [10] to include the software components, in parenthesis, relevant to each layer used in our implementation. 38

Figure 2.6 Automated general configuration of TAU using the `installtau` script... 40

Figure 2.7 Manual configuration of specific features (lines 4 and 5) using TAU's `./configure` script. 41

Figure 2.8 Example of TAU profiling options that were compiled at installation time. Following the `Makefile.tau-` filename prefix are the options selected at compilation time. 41

Figure 2.9 Selective profiling using Eclipse and TAU's selective instrumentation interface. The `df()` function is selected and specific type of profile pattern is applied to it. The modules then automatically generates a `tau.selective` file to be passed to the wrapper script. 43

Figure 2.10 A sample script that sets up the environment for multiple runs of profiling.. 44

Figure 2.11 Both Graphical User Interfaces (GUIs) possess a *main* window from which the data set(s) to be analyzed is selected. The selection is performed in the left pane where trials are presented in the form of a tree structure. The latter depends on how the data was imported using Performance Data Management Framework (PerfDMF). We see in (a) that `paraprof` has an additional branch, which is used for the current folder's data and that (b) possesses an additional leaf named *view*. 46

Figure 2.12 A normalized profile view of all processes including the global mean and the standard deviation (Std. Dev.) of each functions. In this case the metric is the time proportion as per `GET_TIME_OF_DAY`. Each color represents a specific function

and its length is proportional to the total execution time on that specific node. 47

Figure 2.13 Individual functions and group of functions can be selected to focus the displayed statistics. Here, the `TAU_USER` group is selected in the *Group Legend* pane (bottom left), which highlights the relevant functions in the main window (right). Note that we have de-selected the stacked bar presentation for the main window to present an alternative to the normalized stacked bars from Figure 2.12 48

Figure 2.14 An alternate representation of the data in 3 Dimensional (3D). This view provides a more intuitive view of the data through a landscape representation or a series of bars (as shown). The bar height and color intensity can relate to any of the collected PAPI metric or one of the derived metrics created by the user. 49

Figure 2.15 `paraprof` has the ability to display the call graph if the program was profiled with the `-PROFILECALLPATH` option turned on. By default the box width is proportional to the *inclusive* times and the box color is selected according to the *exclusive* runtime of a given function. Both programs are the same but it is clear that the call path from the master node in (a) is different from one of the slave nodes in (b). 50

Figure 2.16 An example of an analysis sequence in `paraprof`. From top left, circling counter-clockwise, is the sequence from `paraprof` manager window, through the bar charts, the call graph and then to the source code. 51

Figure 2.17 The top line shows the *ideal* speedup, based on the experimental data right below it, which starts with $t_{base} = t_1$ (1 processor) up to the timing for $p = 16$ processors. The bottom line seems to have poor speedup as it is far from the *ideal* line (also drawn). For this curve, the baseline time t_{base} is based on the execution with $p = 5$ processes. This induces a distortion in the speedup representation as the two series have a different reference for t_{base}. 53

Figure 2.18 A closer look of the experiment having a baseline time t_{base} with 5 processors demonstrates that it actually exerts *ideal* speedup according to `perfexplorer`'s guideline. 54

Figure 2.19 The speedup of each event is drawn independently to isolate the functions that do not scale well. Functions that fall off the ideal *speedup* reference line are the most probable barriers to scalability. 55

Figure 2.20 Relative efficiency is not affected by the baseline's processor count p. The most efficient implementation (top line), averaging at 1, was originally presented as having comparatively poor speedup in Figure 2.17 56

Figure 2.21 Relative efficiency *by event* can help identify functions with poor scalability. The ideal is to remain close to 1 as processor count grows. 57

Figure 2.22 Comparing three representation of the same profile run using relative efficiency in (b), relative speedup in (a) and a runtime breakdown graph in (c). The intuitive display from the runtime breakdown eases the identification of functions becoming problematic as processors are added. Simply put, a widening cone such as the second predominant layer from the top, is indicative of a growing bottleneck. A tightening cone, on the other hand, means that the function looses proportional importance in the overall execution time. Parallel or constant area are signs of linear (ideal) speedup of a function. 58

Figure 2.23 The use of `perfexplorer` views help consolidating experimental data for a better analytical perspective. All 15 experiments are presented in (a) whereas an averaged view is presented in (b). 59

Figure 3.1 Profiling and execution of the sequential k-means algorithm using TAU. The program is then started by specifying the reference database and the number of samples to load from the database. Here we load 1% of the entire database. The `[snip]` tags indicate output truncation. 63

Figure 3.2 Each graphic is a window from `paraprof`, used to present a specific view of the sequential k-means profile. The call graph in (a) clearly shows that the execution time is mostly attributable to `centroid_def()`. The stacked bargraph in (b), and its deconstructed version in (c), also indicates this proportional importance. The call counts from (d) help identify potential *partitioning* areas as well as its *grain*. (e) is useful for identifying highly *cohesive* functions (many short calls), thus potential communication bottlenecks. 65

Figure 3.3 Database segmentation strategies: TOP- *Strided* segmentation (fine grained) is used by the master-slave algorithm where each element of the database is assigned to one ω worker node in a round-robin fashion. BOTTOM- *Block* segmentation approach (coarse grained), assigns equal *consecutive* chunks of the database to each worker as per $|DB|/\omega$ with the remainder assigned to the last worker..... 68

Figure 3.4 A typical master-slave topology. All communications originate and terminate on the master. The nodes do not communicate between each other..... 70

Figure 3.5 The workers send their partial results to the master..... 72

Figure 3.6 The master updates the workers with the new values..... 72

Figure 3.7 Master-Slave Message Sequence Chart (MSC) for the inter-iteration communications. All communications are point to point and must be performed by all nodes..... 74

Figure 3.8 The 3D view of the master-slave communications `MPI_Recv()` and computation cycles `df()` for all nodes. The master node (node 0) spends most of its time waiting for the results from the ωorker nodes. Columns are colored according to time per call for the function. 76

Figure 3.9 Average time spent by all nodes in each function. Each calls are sorted by order of contribution importance. Calls under 0.008 seconds aren't shown for clarity. Braces indicate the source file and line numbers, bracket information specify which call parameters were used and function call paths are indicated using '=>'..... 77

Figure 3.10 Correlation analysis for $\omega = [2, 24]$. Each function's time contribution is drawn as the worker count grows. The correlation coefficient r, indicates the correlation between the addition of nodes and the execution time of the function..... 78

Figure 3.11 Runtime breakdown for $\omega = [2, 24]$. Each function's proportional importance for the total execution time is depicted by its surface coverage as nodes are added to the computation. A perfectly scalable function would be represented by a constant surface area whereas a growing surface is indicative of poor scaling. 79

Figure 3.12 A typical island topology. Communications originate and terminate between each node. This model implies a fully connected network where all nodes can see eachother (typical

Ethernet configuration). The number of actual communications varies depending on the MPI implementation of the global communicators. 80

Figure 3.13 The three collective calls used to communicate and perform an element by element summation of all three intermediate variables..... 83

Figure 3.14 Island MSC for the inter-iteration communications. Although drawn as sequential, collective communications can overlap within the same call to `MPI_Allreduce` but must complete within the same call (equivalent to a communication barrier). These barriers are depicted by the horizontal dotted lines. They must also be performed by all nodes..... 84

Figure 3.15 Average communication times for both approaches. Master-slave communications are presented in (a) while the only communication for the island model is in (b)..... 85

Figure 3.16 A single collective call performs the exchange and summation of all intermediate values. The variable `c_sum` is supersized to include C, m and $dist$, hence the communication size of $K * T + K + 1$. Each variable simply points to its specific region within `c_sum`.... 86

Figure 3.17 Simplified Island MSC for the inter-iteration communications. A single collective call from each node communicates all intermediate values and performs their sum at the same time. 86

Figure 3.18 Comparing hand coded squared function ($a \times a$) to the use of `pow()` on Intel $Q6600$. The metric used in all cases is the exclusive mean per-call values of the fucntion. In all figures ((a) to (e)), the top bar (in blue) uses the explicit definition while the red bar below uses the library call to `pow(a,2)`. All the presented metrics point to the expanded version as being more efficient by consuming less total time (a), cycles (b), issuing less instructions (c) (total) and even less floating point (d) and vector instructions (e). 90

Figure 3.19 On the left, the original loop. On the right, the fourfold unrolled version of this same loop. 92

Figure 3.20 Pre-assembly output from GCC for an Athlon XP processor for `df()`. On the left, the code is compiled with explicit use of Single Instruction Multiple Data (SIMD) directives such as `-mfpmath=sse -msse -m3dnow`. On the right, the addition of `-ffast-math` has triggered unrolling of loop as well as additional use of the SIMD capabilities, generating more efficiently *vectorized* code..... 94

Figure 3.21 Execution time comparison between using `-O3` (top bars in blue), adding `-ffast-math` (middle bars in red), and also addinf `-funroll-all-loops` (bottom bars in green). The (a) is for the execution time on Athlon XP processors where we can see that `df()` does not seem to benefit from `-funroll-all-loops` but does perform better with about 6% in time gain with only `-ffast-math`. (b) is on Intel $Q6600$ where very little differences are noted between the three approaches. 95

Figure 3.22 The `df()` function using BLAS. On the left, the original loop. On the right, the BLAS version of this same loop. The operations on the right are aligned with the ones they (mostly) replace on the left. 96

Figure 3.23 The Level 1 BLAS libraraies (top blue bars and line) perform poorly in all cases compared to the code optimized with `-ffast-math`. This is reflected in all aspects of the computation whether it being time (a), CPU cycles (b), instructions (c) or even floating point operations ((d) and (e)). Further investigation by varying the vector size has proven this to always be the case as demonstrated in (f) 98

Figure 3.24 Comparing all approaches Athlon XP (a) and Intel $Q6600$ (c). In both cases, BLAS (purple) and `pow()` (light blue) are the worst performing. A direct correlation is made between performance and Level 2 (L2) cache misses (b) for the Athlon XP . In the case of the Intel $Q6600$, the same clear cut correlation, requires that we go down to the Level 1 (L1) cache (d). 100

Figure 3.25 Total execution times on both clusters. The Headless cluster (a), based on Athlon XP hardware, lends a distinct advantage to the use of `-ffast-math`. On the H^2 cluster (b), based on Intel $Q6600$ hardware, most options overlap leading to no clear "winner", barring the use of `GOTO BLAS` and `pow`. 102

Figure 3.26 The runrime breakdown for the best optimized options on both clusters. In (a) most of the execution time on the H^2 cluster is spent in MPI libraries. We see this is not the case in (b) for the *headless* cluster where most of the time is spent in computation. 104

Figure 3.27 A deceptively simple diagram depicting the iterative optimization process of a program. The multiple entry points recall that a change in any one of the elements from Figure 1 are susceptible to provoking a new optimization pass. The ultimate

convergence being that there is no more possible improvements
given a stabilized environment, and one can then *get on with life*. 108

LIST OF SYMBOLS

$BW_{I/O}^{node}$	Node Input/Output Bandwidth
$BW_{I/O}^{server}$	Server I/O Bandwidth
BW_{useful}	Useful Band Width
c	Number of cores in a given processor
C	Table of k centroids
c_z	One of the k centroids from C
DB	Data Base of vector elements
$\|DB\|$	Data Base cardinality
$DB_{\omega j}$	Element j from ω's Local Data Base
d	Vector dimension
$dist$	Distance
E_{comp}	Execution's comparative efficiency
E_p	Parallell Efficiency
E_p^{rel}	Relative Parallell Execution Efficiency
$HDD_{I/O}^{node}$	Hard Disk Drive Input/Output Bandwidth
$HDR_{Ethernet}$	Ethernet Header Size
$HDR_{TCP/IP}$	TCP/IP Header Size
k	Number of centroids used for the k-means algorythm
L	Message Lenght (Payload)
L_{max}	Maximum length
L_{short}	Shortest Pacquet Size
$\mathbf{x}^{\mathbf{t}}$	A vector element from DB, the Data Base
m_k	Element counter for centroid k

m_{ω}^{k}	Element counter for centroid k on node ω
MTU_{size}	Maximum Transmission Unit size
n	Number of hosts
N_{iter}	Number of iterations
ω	Number of worker nodes
p	Number of processes
p_{base}	Number of processes for base line excution
S_p	Speedup using p processes
t_1	Time for single execution thread
t_{avg}	Average time
t_{base}	Base Line Execution Time
t_{byte}	Time to send a single byte
t_{comm}	Total Communication Time
$t_{comp\|\|}$	Computation part of $t_{\|\|}$
$t_{con.avg}$	Average time of running concurrent threads
t_{exec}	Execution Time
$t_{f\|\cdot\|}$	Time to compute a single Euclidean norm
t_{lat}	Latency Time
t_{load}	Loading Time
t_{navg}	Average time for end execution threads
$t_{\|\|}$	Parallel Execution Time
t_{seq}	Sequential Execution Time
t_{single}	Time for a single thread
t_s	Setup Time
t_{start}	Start time
t_{stop}	Stop time

ACRONYMS

3D	3 Dimensional
ALU	Arithmetic Logic Unit
AMD	Advanced Micro Devices
API	Application Programming Interface
bps	bits per second
Bps	bytes per second
BLAS	Basic Linear Algebra Subroutines
CFD	Compitational Fluid Dynamics
CMP	Chip MultiProcessors
CPU	Central Processing Unit
CPI	Clock Per Instruction
CUDA	Compute Unified Device Architecture
DMA	Direct Memory Access
DRAM	Dynamic RAM
EA	Evolutionary Algorithm
CFD	Computational Fluid Dynamics
FLOPS	FLoating point OPertaions per Second
GA	Genetic Algorithm
GCC	GNU C Compiler
GigE	Gigabit Ethernet
GPU	Graphics Processing Unit
GPGPU	General Purpose Graphics Processing Unit
GUI	Graphical User Interface

GNU	GNU is Not Unix
HPC	High Performance Computing
HPCC	HPC Challenge
HDD	Hard Disk Drive
HMM	Hidden Markov Model
HT	HyperTransport
IEEE	Institute of Electrical and Electronics Engineers
ILP	Instruction-Level Parallelism
I/O	Input/Output
Inf	Infinity
IP	Internet Protocol
ISO	International Standards Organization
L1	Level 1
L2	Level 2
L3	Level 3
LLVM	Low Level Virtual Machine
MIMD	Multiple Instruction Multiple Data
MMU	Memory Management Unit
MPI	Message Passing Inerface
MSC	Message Sequence Chart
MTU	Maximum Transmission Unit
NaN	Not a Number
NIC	Network Interfafce Card
NFS	Networked File System
UMA	Uniform Memory Access

NUMA	Non Uniform Memory Access
NFS	Network File System
OpenCL	Open Computing Language
OS	Operating System
PAPI	Performance Application Programming Interface
PerfDMF	Performance Data Management Framework
PDT	Program Database Toolkit
PtP	Point to Point
PXE	Pre eXecution Environment
PDT	Program Database Toolkit
PVQ	Parallel Vector Quantization
QPI	Quick Path Interconnect
RAM	Random Access Memory
RAID	Redundant Array of Inexpensive Disks
SATA	Serial Advanced Technology Attachment
SIMD	Single Instruction Multiple Data
SPMD	Single Program Multiple Data
SMP	Symmetric Multi-Processors
SRI	System Request Interface
SSE	Streaming SIMD Extension
TAU	Tuning and Analysis Utilities
TCP	Transmission Control Protocol
ZCAV	Zoned Constant Angular Velocity

INTRODUCTION

It is a well known fact that parallel processing is a multidisciplinary field of research where the computing infrastructure encompasses most of the electrical, software and telecommunication fields of engineering. And this is only for its implementation, to which we must add the disciplines proper to the environment being used, themselves covering a wide range of interests from Computational Fluid Dynamics (CFD) modeling (think weather forecasting) to biochemical engineering passing through genetics research. The intertwining complexity is amplified when one considers the Beowulf approach of High Performance Computing (HPC) where a wide range of configurations and heterogeneity of the hardware tends to transform traditional computational models into a complex mish mash of exceptions. If we also consider the widely varying computation characteristics of the code to be executed in such environments, ranging from embarrassingly parallel to highly cohesive (computation versus communications bound), the answer to *Which clustering solution is the best?* can simply not exist without intricate knowledge of the program and the underlying environment upon which the execution is to be performed.

To illustrate these intricacies, Figure 1 presents an overlapping view of the typical hardware and software components involved in the HPC parallel processing context. In this figure, we have also separated the domains of interaction whether it be hardware versus software or user versus system. The quadrants generated by this subdivision can each be interpreted as a field of specialization which can be further subdivided by the components from which they are composed.

Taking all these facts into account, one cannot claim the existence of a universal solution, hardware or software, which can be applied to all cases. Profiling of any computational task and/or of the underlying hardware is therefore a requirement for the attainment of performance maximization given a specific environment.

Even with such precise knowledge of the software, estimating its performance on different hardware can prove to be a daunting task which will tend to lead to false conclusions implying that the exercise of profiling is a task to be re-iterated each time new hardware is encountered.

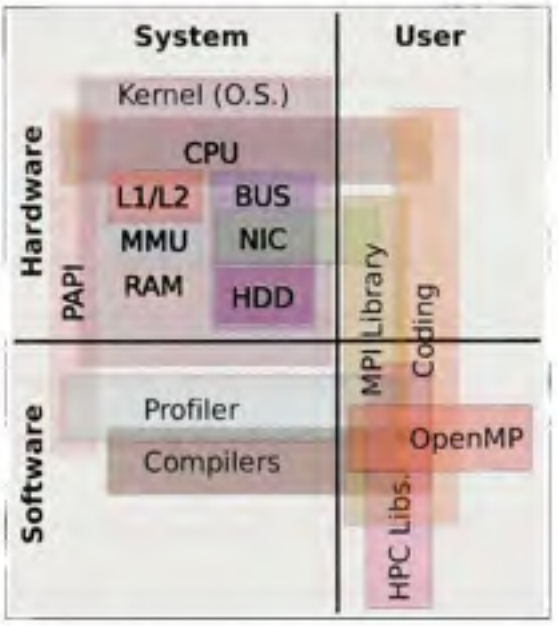


Figure 1 : An illustration view of the multiple elements and disciplines involved in parallel processing. Each quadrant represents a field of research with each underlying component being a specialization. The crossing of quadrants signify its multidisciplinarity and the overlapping emphasizes integration complexity.

Problem Statement

Research in the area of machine learning algorithms (including Evolutionary Algorithms (EAs)) is known to be computationally intensive and has been a growing user of parallel processing approaches to enhance its capabilities.

As an accepted fact, most of the processing payload resides in the fitness evaluation functions where a proposed solution is weighed. In the realm of EAs, the acceleration of this computation step can either lead to a faster or a better solution for a given problem. Fitness evaluators are problem-specific and cannot be generalized, which is why we concentrate on such a given

fitness evaluator, the K-Means statistical classifier, as was implemented in [44], Section 3, *Foreground-Background Feature Extraction (FBFE) Module*.

Given the nature of the K-Means algorithm, the most classic means of determining execution performance, the total run time, is of little use in itself. This is due to the fact that this iterative process terminates based on a convergence threshold, which is in turn affected by random initialization values and the number of participating nodes. In the case of a parallel implementation, it requires that other metrics than total execution time be used to gauge its performance such as scalability and efficiency. With the added complexity of a parallel execution environment, specialized tools are required to provide a concise view of the program's behavior and evolution. As algorithmic and/or code optimization techniques are applied, one must ascertain that the latter lead to an improvement and not a scalability bottleneck.

All these constraints, added to the aforementioned HPC parallel processing paradigms, require that a unified approach be used to guide implementers as to where efforts should be deployed to enhance performance. It is common that, in university research, the implementers (graduate students) have a short time to learn all aspects of their project, programming environment and the code base they will most probably be using and modifying. These three aspects tend to mutate, implying that the performance analysis infrastructure used has to be adaptive, flexible, and most importantly, relatively simple of use.

To demonstrate how this can be accomplished, we start by describing technically and empirically the hardware characteristics in Chapter 1. We then present the techniques and tools by which we probe the software being executed on this hardware in Chapter 2. A case study is then presented in Chapter 3, where we fuse the tools from Chapter 2 and the architectural knowledge from Chapter 1 which brings us to recommendations and future outlooks in the conclusion.

CHAPTER 1

HARDWARE CHARACTERIZATION

Although it may seem trivial or paradoxical to possess knowledge about a program to be executed in a given HPC environment [1], it is a key component to guide the proper profiling of any hardware platform. Ignoring the application domain can result in misguided concerns about a component that ends up being trivial for the targeted application. For example, concentrating on network fabric performance when, in fact, an application is memory or computationally bound, rather than communications bound, can turn out to be a waste of effort and resources. This fact is actually alleviated by the classic Beowulf rhetorical question :

What hardware should I use to build a cluster?

to which the non answer usually follows as :

It depends.

HPC coding requires intimate knowledge of the target hardware architecture as the implemented strategies depend on their characteristics. Starting from a superficial perspective, if the available hardware is in the form of an Symmetric Multi-Processors (SMP) machine, one would probably concentrate on applying approaches where communication costs can be neglected and where memory might be plentyfull. At the other end of the spectrum, the infrastructure might be composed of a mass of heterogeneous computers with varying specifications, interconnected using relatively slow links but possessing ample local storage. Digging deeper, one might find out that the second model proves to be more effective since each node would happen to have faster, less contentious memory access and demonstrate the ability to tap advantageous aggregated Input/Output (I/O) bandwidth thanks to local storage.

Obtaining knowledge of the target hardware architecture is a non trivial balance between theoretical models and supporting empirical data. The collection of such data is usually accomplished via micro-benchmarks and cluster gaging utilities [35]. Unfortunately, these remain

1. Which comes first, the software or the hardware, and is the profile on hardware X still applicable to hardware Y?

either too problem specific or too general to be of true value. For this reason, we will concentrate on characterizing the available hardware assuming some *a priori* knowledge of a problem to be optimized (in occurrence, the K-means algorithm detailed in Chapter 3), which exhibit vectorial computation features coupled with considerable data traversal and, in its parallel implementation, adds communications at each iteration [2].

We now present some of the basic concepts pertaining to computer architecture and communications fabrics. These elements will be useful when attempting to describe some of the characteristics and results of software profiling as presented in Chapter 2 and 3.

1.1 Basic Computer Architecture

Today's common computers are still loosely based on the what is commonly known as the *Von Neumann architecture* [23, 50] which means that they are essentially comprised of (at least) one of each of the following elements:

1) A control unit (for decoding the instructions and managing data flow);
2) An Arithmetic Logic Unit (ALU);
3) Main memory (such as Random Access Memory (RAM) more often referred to as Dynamic RAM (DRAM));
4) An Input/Output unit managed by the control unit.

1.1.1 The Control Unit and Arithmetic Logic Unit

The control unit and ALU are probably what characterizes a Central Processing Unit (CPU) core the most from the point of view of a compiler. It is in these components that mnemonics [3] are defined to mock up the instruction set and internal structure of a CPU. For the average user, these differences usually don't mean much but can have a significant impact in scientific computing.

2. The problems studied are embarrassingly parallel data mining applications which are typically memory bound.

3. Menmonics are the short textual words representing operations a CPU can execute (op-codes). They are the building blocs of the assembly language from which binary code (programs) are created.

For example, Advanced Micro Devices (AMD) has implemented a class of mnemonics which they have named `3DNow*` [4]. On their side, Intel has added their own class of mnemonics known as the Streaming SIMD Extension (SSE), which they have named `SSE*` and `SSSE*` [5]. In all cases, they are an implementation of Instruction-Level Parallelism (ILP), where performance enhancement is accomplished by applying a single instruction to multiple data elements loaded into independent registers of a given CPU core. This approach to low level parallelism is by definition known as Single Instruction Multiple Data (SIMD). The intent is that CPU cores would exhibit enhanced performance when dealing with vector intensive applications typical of multimedia and scientific computing. Nonetheless, proper use of these directives remains a daunting task for the compilers [15], which can benefit from some hints by the programmer, as we will see in Section 3.6.3.

One must not confuse the SIMD extensions with the advent of Chip MultiProcessors (CMP), which are part of yet another class of parallel architecture known as Multiple Instruction Multiple Data (MIMD). In this case, each processing stream (or program) is executing independently, implying a complete decoupling of instruction and data flow. The use of MIMD programming happens at the application level and does not exclude SIMD, the latter being implemented in each computing core. The only implication is that the program execution streams are independent in the case of MIMD and require explicit synchronization mechanisms. An automated implementation of such MIMD approach on CMP and SMP machines is the use of the OpenMP [6] compiler directives.

1.2 Caching in on The Main Memory

It is a well known fact that the DRAM performance curve is substantially inferior to the processor's speed evolution over the past decades [23], p.289. To compensate for this bottleneck, processors are built with on-chip caches [7] which help in speeding up memory access by

4. We use the * as a globing character to include all subsequent classes.

5. AMD now also supports the `SSE*` and `SSSE*` class of mnemonics. Note that the extra `S` means "Supplemental".

6. OpenMP is a specification which compilers are free to implement. For details, please visit `http://openmp.org/`.

7. Instruction and data caches can either be separate or common, depending on the hardware implementation.

prefetching data and instructions. The size and speed of these caches is dictated by its proximity to the processor core(s), which in turn is guided by transistor count limitations for a given physical space, heat dissipation and, of course, production costs [23]. This leads to the hierarchical memory layout of most computers where the processor's access to memory is a growing succession of caches, known as levels, who's efficiency is characterized by the ratio of *hits* and *misses* to each of these levels. These cache *levels* are organized starting from the Level 1 (L1) cache, characterized by its high speed but relatively small size [8]. Then follows the Level 2 (L2) cache, slower than the L1 cache but many times larger, it currently ranges from a few hundred kilobytes to a few megabytes. Now becoming more common, the Level 3 (L3) cache is larger than L2 (two to four times), and is mostly used for CMPs as a shared memory space between multiple processors [52, 6]. The last and slowest link down the memory hierarchy being the DRAM memory modules [9] with their ample capacity of a few gigabytes but with comparatively slow access time and bandwidth.

1.2.1 Accessing The Main Memory

The L1/L2, Memory Management Unit (MMU) and RAM blocks of Figure 1 are a gross representation of the actual processor to memory architecture now present in modern computers. The model becomes more complex as caches, processors and cores are added to a system. One constant remains, the Memory Management Unit (MMU), which plays a critical role in computer performance as it manages the data flow between the main memory and the processor and is reputed as the bottleneck of any modern system. The two major computer processor manufacturers, AMD and Intel, have diverged in this respect during the past years when comparing AMD's Athlon/Opteron and Intel's Pentium/Core 2 processors. AMD has opted for a Non Uniform Memory Access (NUMA) approach where each physical processor integrates its own MMU and possesses a local memory bank. Although the *local* memory of each processor is globally accessible, accessing it comes at a *varying* (Non Uniform) cost depending on the

8. Current processors generally posses an L1 cache close or below 128Kbytes

9. Note that we could push the memory hierarchy down into *virtual memory*, residing on Hard Disk Drive (HDD), but we won't address this case as it is an aberration to HPC and must be treated as an element that must not be used in such a context given HDDs are many orders of magnitude slower than RAM.

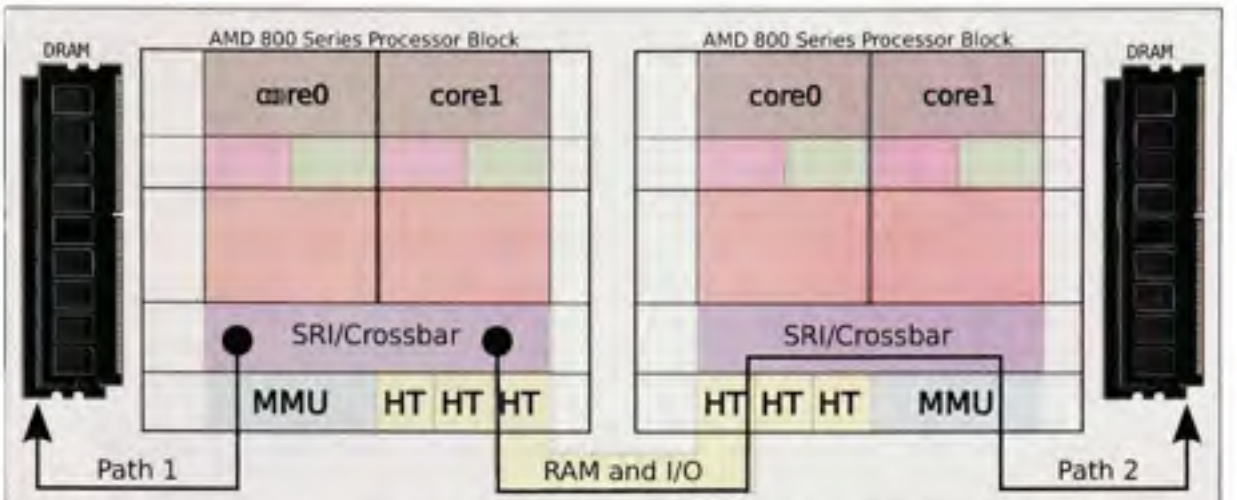
path required for Memory Access (hence NUMA). We illustrate this in Figure 1.1 (a) where a processor accessing its local memory has a direct path (depicted by Path 1) and accessing another processor's memory bank requires a more elaborate, thus longer, path (Path 2). Intel has typically kept the MMU as an external device, which implies a uniform access to the memory banks [10] as illustrated by Figure 1.1 (b).

Figure 1.2 (a) is a schematization of a typical AMD Opteron series of processor. It possesses an on chip MMU where the System Request Interface (SRI) interconnects multiple cores through the Crossbar (internal processor communications fabric). The Crossbar then selects between the MMU for local memory requests, or the HT link if the requested memory address is on a *remote* processor. This implies that access to local memory (going through the MMU) is *uniformly* shared by all cores of a single processor unit. Intel's approach implemented in the Core 2 series processors is depicted by Figure 1.2 (b) where we can see that L2 cache is shared and that the MMU resides on an external chip (usually called the North Bridge).

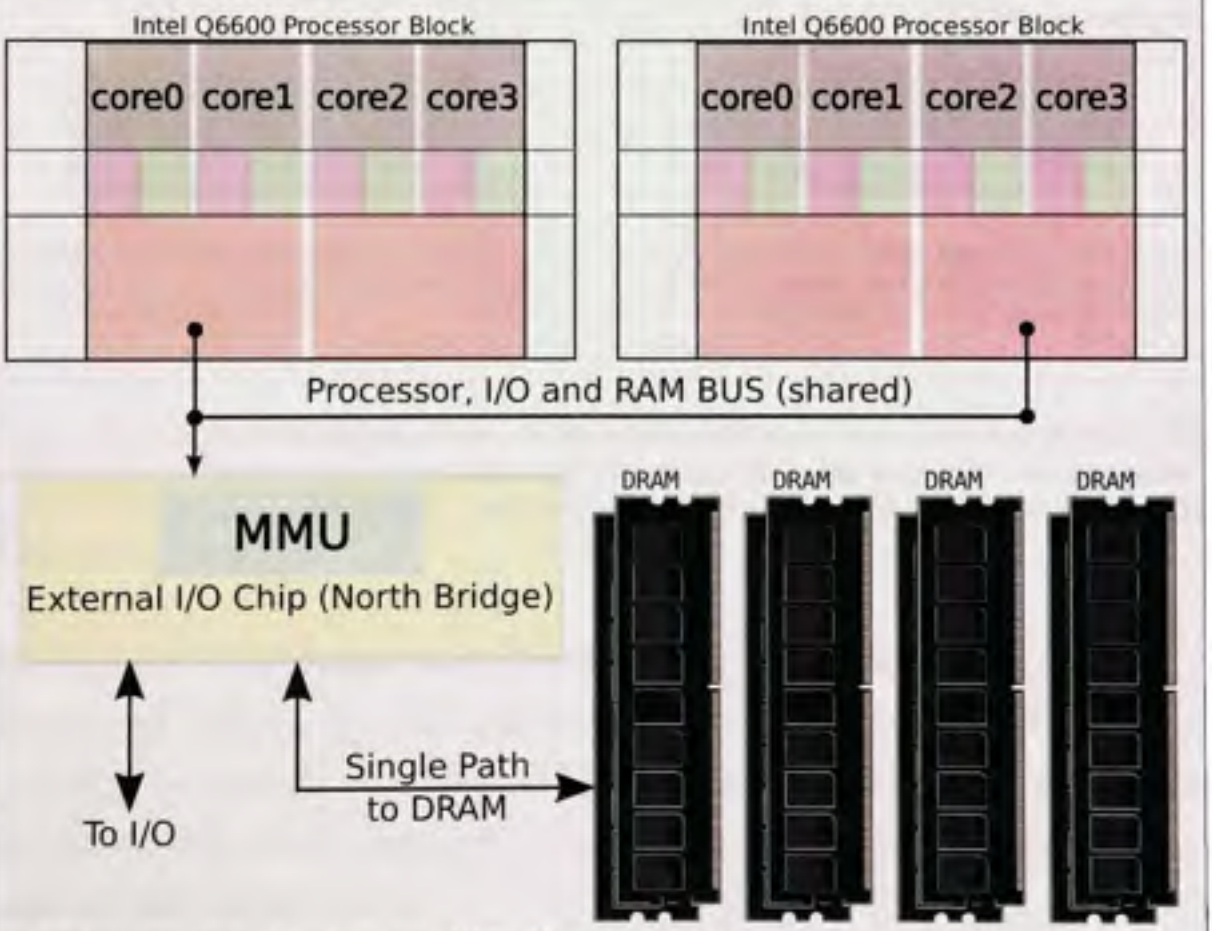
In the case of AMD's implementation, access to memory physically connected to another processor requires the use of the HT link [32, 28] and is typically NUMA in nature. In Figure 1.1 (a), Path 1 illustrates the local core's direct path to memory going through the SRI/Crossbar and MMU. Access to remote memory is illustrated by Path 2 where a request has to traverse the HT link as well as both processor's SRI/Crossbar logic, which adds latency and transfer delays. As processors are added to the system, more of these *hops* can occur, depending on the interconnection strategy used [28]. For Intel type CMP systems, the MMU is an external device and is dependant upon the motherboard implementer to select the interconnection strategy. Generally, these consist in using a single fast bus for I/O, inter-processors and memory (through a single MMU), as illustrated by Figure 1.1 (b).

The direct implications of the differing memory subsystems is that, apart from external hardware required to link Intel's processors, they must share the memory bandwidth evenly across processors and devices whereas AMD's processors each have their own local memory banks.

10. Although this will no longer be true with their Core i7 series, where they have opted to integrate the MMU into the processor die.

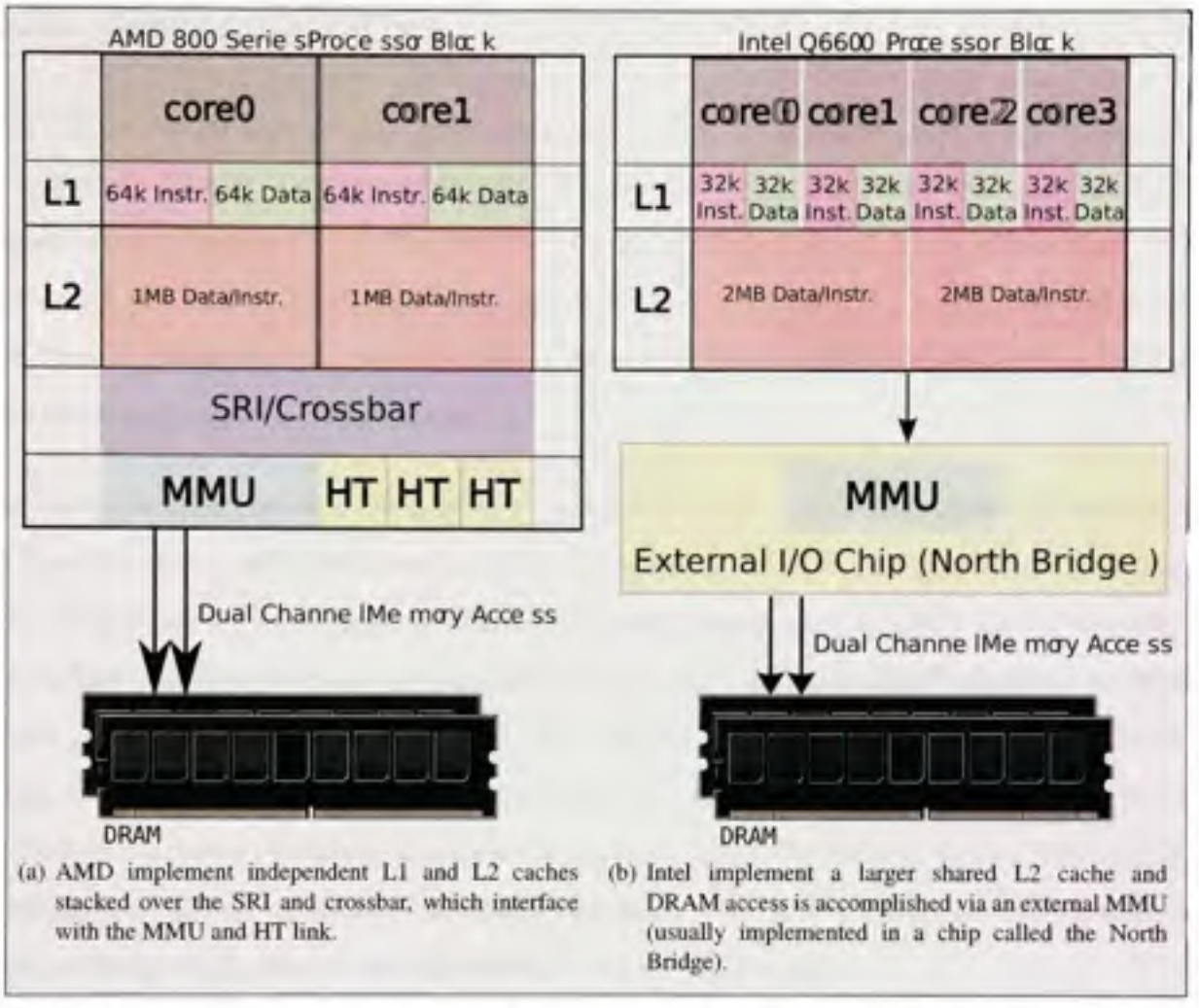


(a) Multi-processor implementations of the Opteron processors possess UMA characteristics where local cores access memory uniformly as seen by Path 1. Access to remote memory (physically attached to another processor) requires passing through the HT link and two controller stacks as demonstrated through Path 2, which is NUMA by definition.



(b) Intel's processors don't implement the MMU. Most CMP strategies available on the market implement it as a single chip (north bridge) which is accessed through a shared bus topology connecting all processors through which inter-processor communications, I/O and memory traffic is subject to contention.

Figure 1.1 : Multi-processor memory access strategies for both AMD and Intel processors. AMD possesses NUMA characteristics while Intel's implementation is essentially UMA.



(a) AMD implement independent L1 and L2 caches stacked over the SRI and crossbar, which interface with the MMU and HT link.

(b) Intel implement a larger shared L2 cache and DRAM access is accomplished via an external MMU (usually implemented in a chip called the North Bridge).

Figure 1.2 : These schematizations of the AMD Opteron Dual Core processors (800 series) and the Intel Core 2 Duo processors illustrates how the two core variants access DRAM. In both cases, the MMU possesses dual channel connectivity to DRAM for link bandwidth aggregation.

This implies that Intel's memory access is bound to memory bandwidth and bus contention as I/O traffic and processors are added to the system. In theory, AMD's on-chip MMU leverages its processors as the ideal candidates for embarrassingly parallel applications where aggregate memory bandwidth across multiple processors (not just multiple cores) is more important than single-threaded memory access.

1.2.2 Cache Size and Contention

Working at the processor's clock speed or a fraction of it, these caches are orders of magnitude faster than DRAM. Fetching and synchronization of the data between the caches and the main memory is managed by the processor's logic through different mechanisms which rely on easily predictable or repetitive (strided) data access patterns [31], p.300. The efficiency of these *prefetching* mechanisms is one of the most critical components for closing the gap between computation and data access.

Modern processors are now being built to contain many cores and possess a growing amount of L1 and L2 caches and some times L3 caches are added as the inter-core communications layer [6]. Depending on the strategy adopted by the manufacturer, the L1 and L2 caches can either be unique to each core or shared. Independent caches per core mimics SMP architecture where each processor is essentially monolithic and virtually interconnected with a high speed *bus*. This also implies that each core is constrained to only possessing a fraction of the cache that it otherwise would be possible to implement as a global cache. This strategy can be beneficial for independent data flows but could hamper performance when problem sizes are considerable or when data is locally shared amongst multiple concurrent threads.

As a reciprocal to this approach, Intel has implemented a large shared inter-core L2 cache strategy for it's Core 2 processors. This approach has the advantage of a large cache for single threads but shared cache for concurrent threads. Figure 1.2 compares both of these strategies where AMD's Dual Core Opteron 800 class of processors assign independent L2 caches and Intel's Core 2 Quad processor is composed of four cores with L2 caches organized in core pairs.

To demonstrate the different cache issues with concurrent and independent processes running on a CMP, was programmed Algorithm 1 in C. This Euclidean *computation kernel* is derived from our case study presented in Chapter 3. For our demonstration, we vary the vector dimension d between $128K$bytes and 2Mbytes per process in order to saturate the L2 caches when as many processes as cores are started (four processes for a quad-core CMP). Note that we kept

the problem size boundaries identical across experiments (not a function of the processor's cache size) to ease the comparison. We then compute the concurrent execution's *comparative* efficiency E_{comp}, which we define to be:

$$E_{comp} = \frac{t_{single}}{t_{con.avg}} \tag{1.1}$$

with t_{single} being the time for a single thread of execution on a given processor and $t_{con.avg}$ the average time of running concurrent threads [11] on that same system. This result is useful in identifying the interaction zones for concurrent execution of independent programs on a CMP.

1: Set d to maximum vector dimension ($\|X\|$)
2: Set $REPS$ to maximum repetitions
3: Initialization of vectors X and Y for Euclidean computation.
4: **for all** $i = 1$ to d **do**
5: Set $t_{start} = gettimeofday()$
6: **repeat**
7: Compute Euclidean norm such as $dist = \sum_{j=1}^{i} (\|x_j - y_j\|)$
8: **until** Computation has been executed $REPS$ times
9: Set $t_{stop} = gettimeofday()$
10: Compute average time as $t_{avg} = (t_{stop} - t_{start})/REPS$
11: **end for**

Algorithm 1: Memory contention test algorithm.

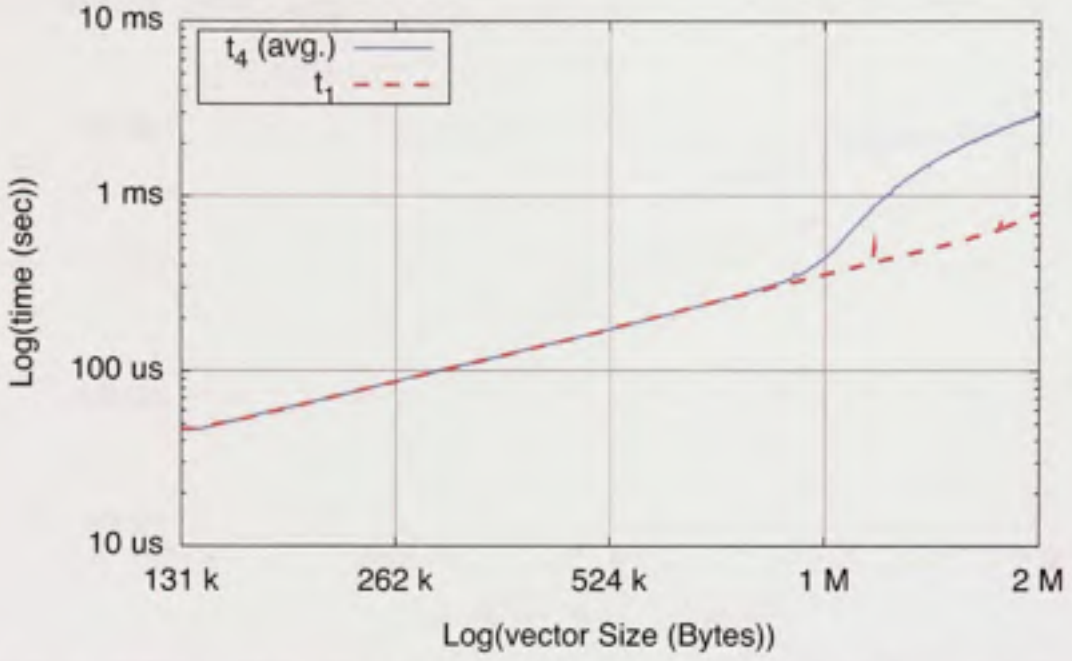
Our results for the Intel Q6600 processor are presented in Figure 1.3 . Execution times are presented in Figure 1.3 (a) where we observe performance degradation due to execution concurrency. The cause for the degradation is attributable to the zones identified in Figure 1.3 (b) which correspond to cache usage zones. Performance degradation begins when the vectors X and Y both reach sizes of about 760kbytes per process are reached. With four concurrent processes, this brings the total to about 6Mbytes. This induces cache *conflicts* as the total cache capacity is 4Mbytes for all threads. The processor is forced to move parts of working data out of cache for one or all of the executing processes. Cache *capacity* issues are then reached at

11. The number of threads is equal to the number of available cores on the system.

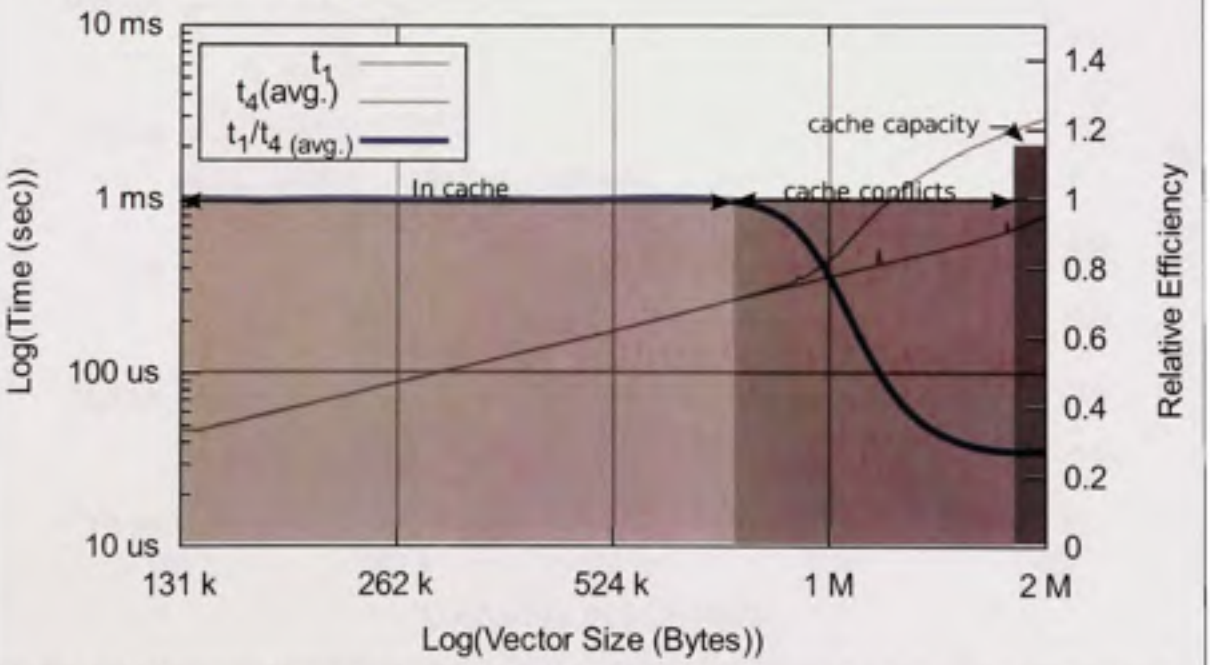
2Mbyte vectors, which is concurrent with the processor's 4Mbyte cache as both vectors for a single thread fill up the cache, leaving no space for the three other threads. At this point, each thread is executed at about 30% efficiency (close to four times slower). These results clearly demonstrate the importance of cache size for the execution time of large *memory bound* kernels as well as concurrency issues that may arise within multi-core processors.

The same observations are applied to an Opteron based SunFire $X4600$ machine [12] and presented in Figure 1.4 . Here we can see the significance of the NUMA architecture through the fact that the relative efficiency never gets even close to $1/14$ (0.07), which would be expected if all fourteen processes had to share a single path to the DRAM. Since each CMP have a direct path to local memory, the contention effect is limited to local processor and is not globally cumulative. This implies that this architectural approach is more scalable, as long as each problem is local to each processor and fits within the local DRAM banks.

12. Refer to Appendix III, section 3.

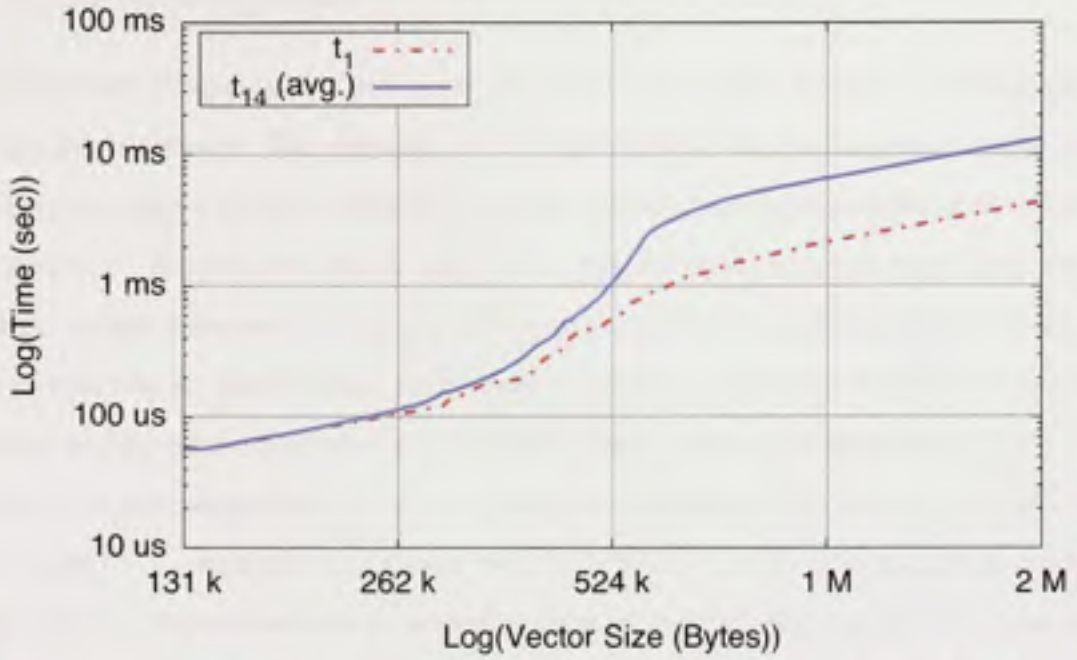


(a) Average concurrent execution time of 4 processes ($t_{4(avg.)}$) versus the vector size. Divergence starts around 1Mbytes problem size per process and is linearized around 2Mbytes. Our comparison baseline is t_1, the time for a single process.

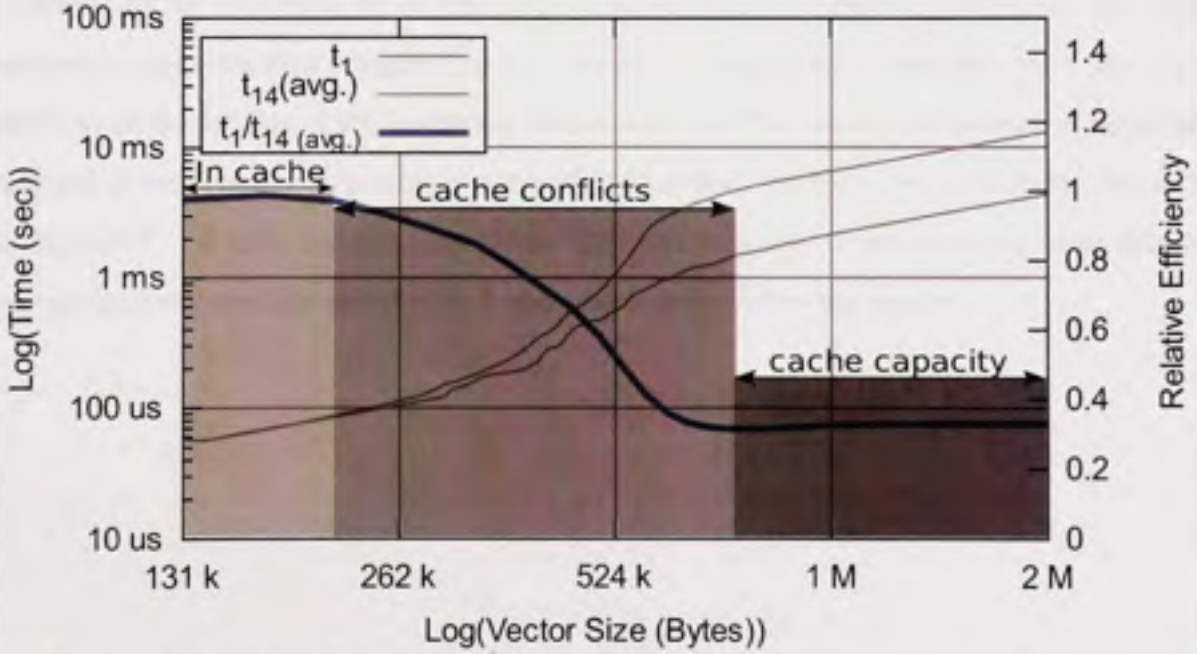


(b) An overlaid version of Figure (a), the execution's *relative efficiency*, computed as $t_1/t_{4(avg)}$, is used to identify different cache usage zones. Presented are: *in cache*, cache *conflicts* and cache *capacity* (saturation).

Figure 1.3 : Cache memory behavior on an Intel $Q6600$ (4 cores). Execution time characteristics are illustrated in (a). Cache usage zones are identified in (b).



(a) Average concurrent execution time of 14 processes ($t_{14(avg.)}$) is affected as the problem size grows. Divergence starts around $128K$bytes problem size per process and is linearized around 1Mbytes. Our comparison baseline is t_1, the time for a single process.



(b) The execution's *relative efficiency* curve is used to identify different cache usage zones. The green zone denotes optimal cache usage whereas red indicates *conflicts* are occurring. Values beyond cache *conflicts* fall into cache *capacity* issues (saturation).

Figure 1.4 : Cache memory behavior on AMD Opteron 800 series based processors using 14 cores of a SUN SunFire $x4600$. Execution time characteristics are illustrated in (a). Execution zones are identified in (b).

1.2.3 Processor Performance

There is no such thing as a *best processor* but rather a best match between a software problem and a hardware solution. The test case we have presented in this section uses a wide range of values and executes a single mathematical kernel, which is not representative of the entire program process [13]. Nonetheless, this isolation tactic and the use of aberrant cases (unconventionally large vectors compared to typical problem sizes) is of use to defining bound within which we can expect severe performance deterioration as well as scalability bottlenecks (concurrent execution performance degradation). Even if the figures indicate *better* scalability for a given platform, raw processing time will always prime over technical features and prowess. To this effect, Figure 1.5 compares the average execution time of concurrently executing 4 instances of Algorithm 1. The comparison is performed between Intel's $Q6600$ and AMD's Opteron 885 processors (on a SunFire $x4600$). With this current representation, Intel's $Q6600$ comes out as the best choice, even though its architecture is more susceptible to memory bottleneck issues. Additionally, the execution of 14 processes is included in the graph to emphasize the slight increase in execution time compared to 4 processes. It is important to note that, the 4 processes launched on the SunFire $X4600$ were not bound to CPUs. This means that each processes were assigned an independent processor and therefore benefited from full, non-contended access to the DRAM [14]. We must also note that the concurrent executions do not incur any inter-process communications, another aspect which we address in the following section.

13. Actually, each test *is* the result of the execution of the entire program, the point is that this program is useless in itself, typical of a microbenchmark.

14. Processor affinity, to force process to CPU assignments, was not available on the hardware at the time of writing.

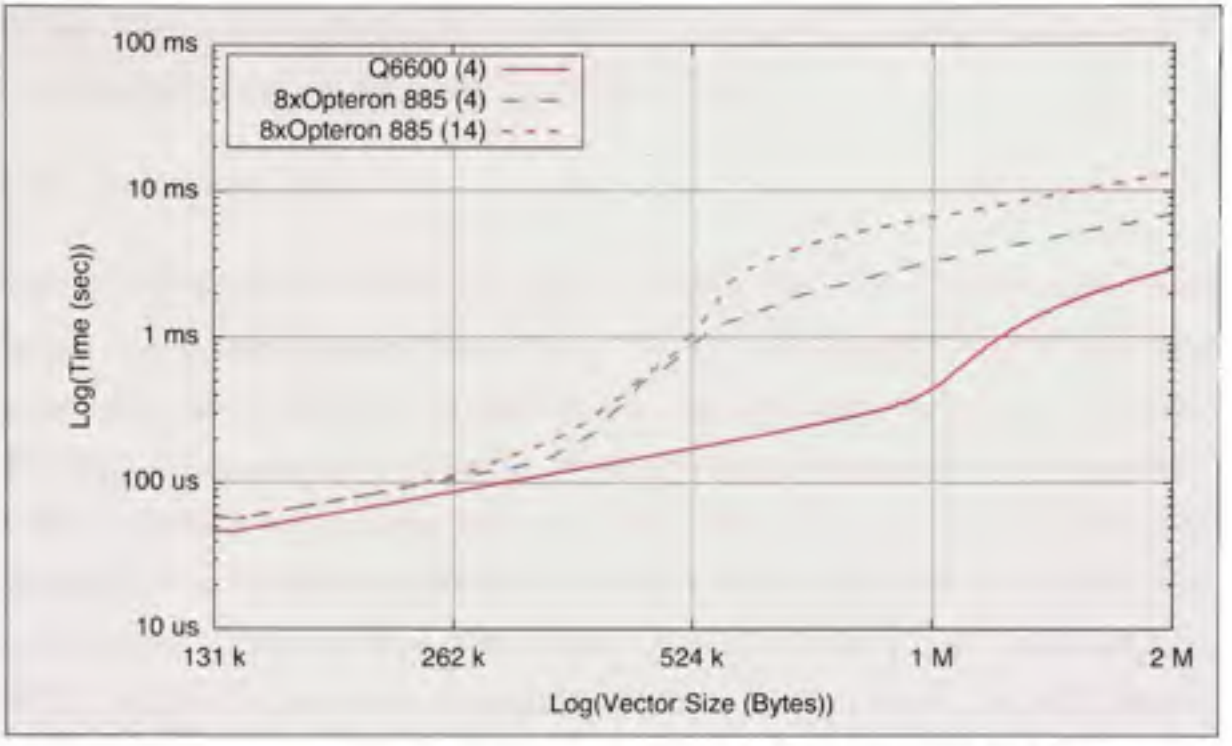


Figure 1.5 : Execution time comparison between Intel's $Q6600$ and AMD's Opteron 885 processors. The concurrent process count is in parenthesis. The raw computing power of the $Q6600$ outperforms the Opteron 885 for four processes. The case of 14 concurrent processes is presented to demonstrate the proportionally small impact of their simultaneous execution.

1.3 Communications

We have shown that raw processing power has to be coupled with an efficient mechanism for accessing the data that resides in RAM. The typical problem sizes, as addressed in Chapter 3, overcome the memory and processing capabilities of a single processor system. This introduces the problem of segmentation, thus parallel processing, which imply multiple processors and communications. Independent of the hardware nature of the latter, two principal characteristics, latency and bandwidth, come into play. This section aims at characterizing these two critical components as well as weighting their importance to our usage context, which are:

1) A network of computers forming a Beowulf style cluster interconnected using Ethernet based network fabric;
2) A monolithic SMP machine using HT as network fabric.

We start with the *Beowulf* approach to parallel computing to define bandwidth and latency. We then apply these two properties on HT based SMP systems.

1.3.1 Bandwidth

Bandwidth traditionally represents a bit count transferred over a unit of time, which is usually the second as denoted by bits per second (bps). This is the predominant feature of most fabrics, hence names such as $10/100/1000$ BaseT Ethenet, where the later is the name of the standard describing the physical medium. Taking 100 BaseT Ethernet as an example, its bandwidth is said to be 100Mbps *wire speed* or *at the wire*. This is because the figures given are for the raw bit transfer rates, ignoring all of Ethernet's protocol overhead, such as the headers which sums up to 38bytes [15]. Another feature of the Ethernet protocol is the Maximum Transmission Unit (MTU), which is the maximum *payload* allowed per packet. Historically, the MTU has been hard-limited to 1500bytes by the underlying hardware [16] which simply followed the Ethernet standard [17]. This upper bound to the size of each packet has a direct bearing on the efficiency of the communications as shown by Eq. (1.2), where BW_{useful} is the available bandwidth which is a ratio between the *useful payload* over the total bytes transmitted per packet. The total bytes transmitted is the sum of the MTU and the Ethernet headers (again, 38bytes). The *useful payload* is computed using MTU_{size}, the MTU, from which we substract $HDR_{TCP/IP}$, the TCP/IP headers.

$$BW_{useful} = \frac{MTU_{size} - HDR_{TCP/IP}}{MTU_{size} + HDR_{Ethernet}} \tag{1.2}$$

Taking into account the aforementioned values, the equation renders an available bandwidth of about 95%. The transfer rate in bytes of a 100 BaseT network becomes $100/8 \times 0.95 = 11.88$M bytes per second (Bps) [18]

With Gigabit Ethernet (GigE), 9kbyte MTU called *Jumbo Frames* were introduced to address the overhead issue and has now become common. Other than bringing the available bandwidth

15. We don't use VLANs (RFC802.1q), otherwise, this figure would be 40bytes.
16. Such as switch fabric, buffer limitation and Network Interfafce Card (NIC) implementations.
17. As described by RFC894.
18. This is a raw value, meaning that from this bandwidth one must also substract library overhead.

up to 99%, it has the effect of reducing the framing overhead of large data transfers. Unfortunately, this has no impact on small communications, where latency dominates. Which brings us to the following topic.

1.3.2 Latency

Latency is the delay imposed by hardware and software before establishing a link and actually starting the communication stream. For this reason, it is often modeled as if sending a 0byte packet. This overhead is very important for short communications. We define *short* communications as packets who's length (L_{max}) renders a transmission time less than the link's latency (t_{lat}). This value is simply obtained by multiplying the latency with the useful bandwidth, as in Eq. (1.3).

$$L_{max} = t_{lat} \times BW_{useful} \tag{1.3}$$

For example, if the latency for a 100 BaseT connection is of about $t_{lat} = 23\mu$seconds, given the theoretically *useful* bandwidth $BW_{useful} = 11.88$M Bps, we get $L_{short} \approx 273.24$bytes. This is one way of actually weighting the latency's cost on the communications.

These values were obtained thanks to empirical experimentation using a microbenchmark such as `mpptest` [22]. The reason why empirical data is more valuable than theoretical ones is made obvious in figures 1.6 and 1.7 where significant performance differences exist between Message Passing Inerface (MPI) communication library implementations across the communication spectrum.

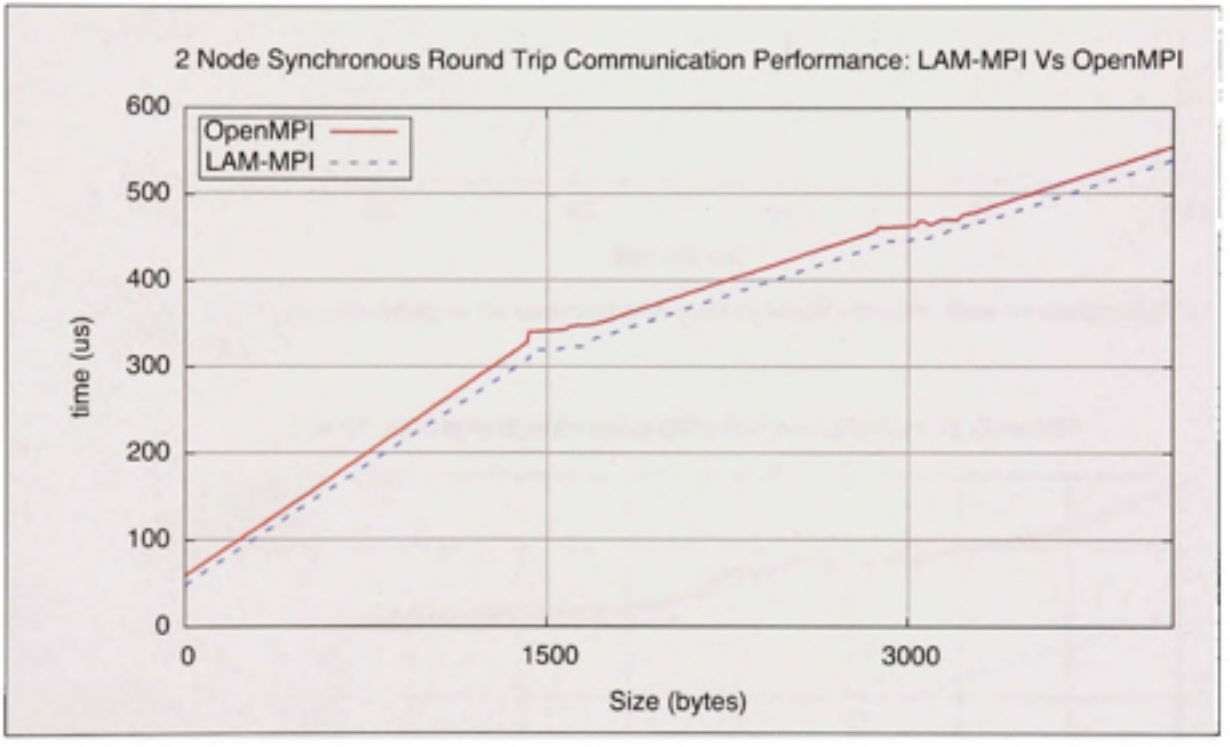
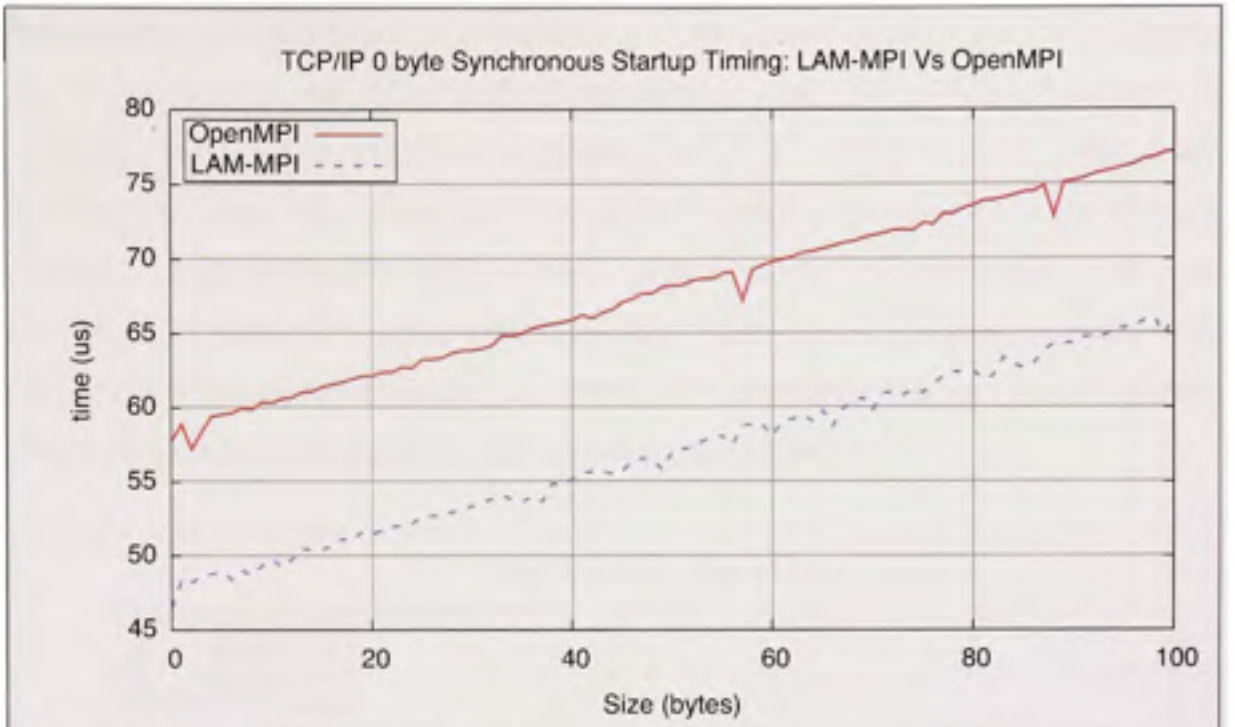
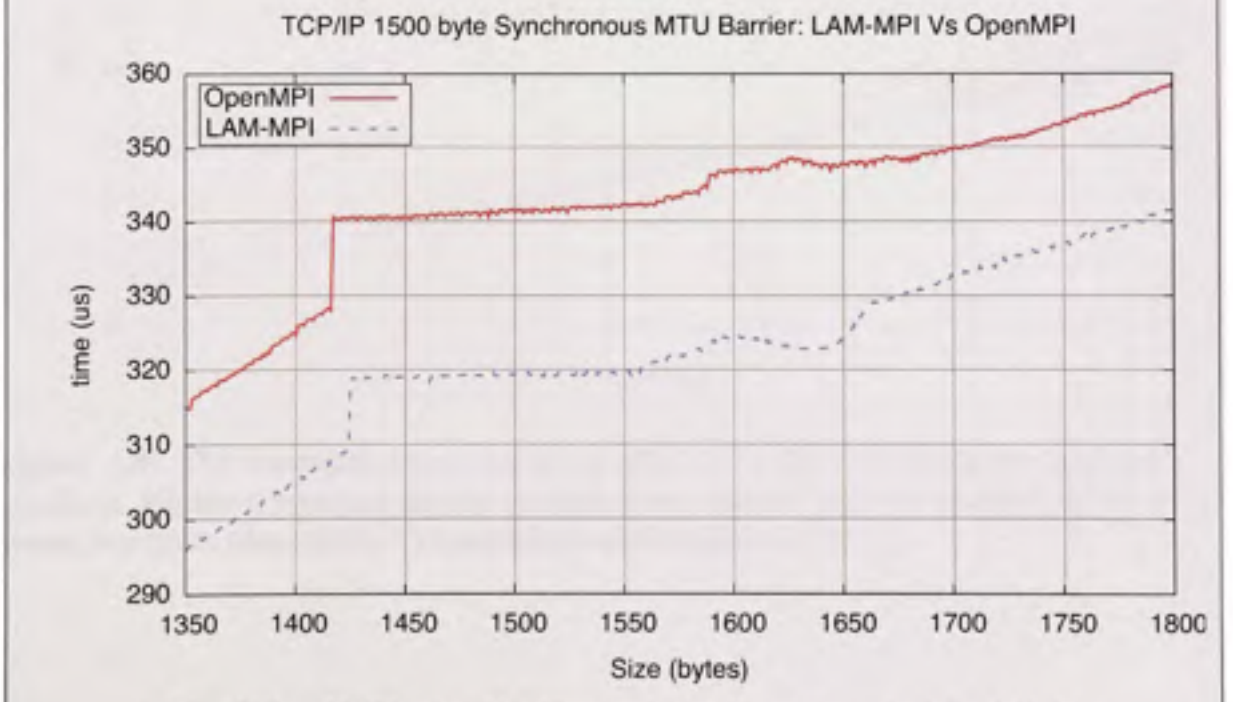


Figure 1.6 : LAM-MPI outperforms OpenMPI for any TCP/IP communications. The non-linearity are noted around the MTU barriers of 1500bytes.



(a) The setup latency differs depending on the communications library implementation. Here we compare LAM-MPI with OpenMPI.



(b) The 1500byte MTU barrier adds a 10μsecond delay due to communication re-initiation.

Figure 1.7 : The round trip communication times using MPI libraries surrounding the start up times in (a) and the MTU in (b). Since these are round trip figures, all values have to be halved when considering asymmetric communication patterns.

Computation and characterization of the latency is far less straight forward than bandwidth since it is a transitory state which is highly dependant on many characteristics external to the NIC such as processor, bus and memory speeds as well as network topology. We demonstrate these facts in Figure 1.8 , where processor speed as well as inter-connection topology (the addition of a hop between two nodes) all have a significant impact on the latency of the communications. A faster CPU renders lower latencies, which can very well be explained by its ability to service hardware interrupts more quickly. The addition of hops, through the addition of network switches between nodes, have non-negligible impact as well.

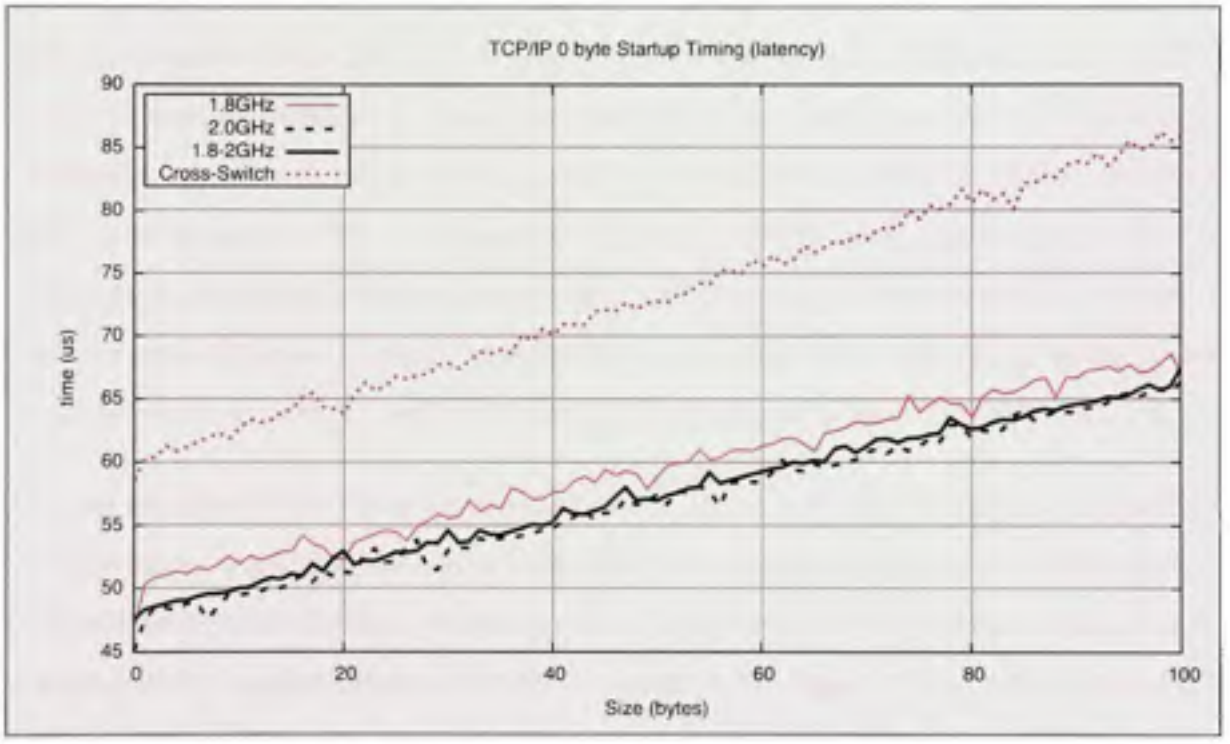


Figure 1.8 : The communications latency is affected by the CPU frequency and network topology. Higher frequency clearly renders lower latency and the addition of a hop between two hosts (denoted as Cross-Switch) adds significant delays.

1.3.3 The HyperTransport Interconnect

The HyperTransport link is the result of the HyperTransport Technology Consortium [19] which is formed by a group of more than 40 companies active in the computer industry. This fact, and the fact that the standard is open and accessible to all, might explain its currently wide adoption across the industry. Although it is not uniquely destined to be used as an inter-processor communication backbone [47], we will concentrate on this specific use for communications.

Of a totally different nature when compared to Ethernet, they present a relatively high speed [20] and low latency [47] path between processors. Although this approach doesn't require underlying communications libraries such as MPI, the libraries are still often used since they present a portable interface to a program's parallelization. For this reason, we still consider the libraries as part of the performance assessment of this fabric. The same latency and bandwidth paradigm apply to HT, even though the figures are orders of magnitude apart. Again, the choice of the underlying communications library can have a significant impact on the application's communication performance as is illustrated by Figure 1.9 , where performance varied greatly between versions 1.1 and 1.2 of OpenMPI's implementation of the MPI.

It is also important to note that topological considerations must still be addressed, especially when frequent communications are expected between computing *nodes* or processors. Proper to NUMA architectures, processor affinity (associating a process to a given processor or core) and data locality become issues when HPC is concerned. In Figure 1.1 (a) from section 1.2.1, we illustrated that the access to remote memory required passing through the HT link, another processor's Crossbar and MMU. Now consider Figure 1.10 , the physical layout of a Tyan $VX50$ machine, where a process residing on `CPU0` accessing memory on `CPU7` would have to perform, at best, 3 hops. This *twisted ladder* configuration is one of many possible connection strategies [28, 32] that can result in differing hop counts. It is the variance in these hops that charaterize the NUMA architecture.

19. `www.hypertransport.org`
20. Between 12.8GBps and 51.2GBps, depending on the implemented version of the standard.

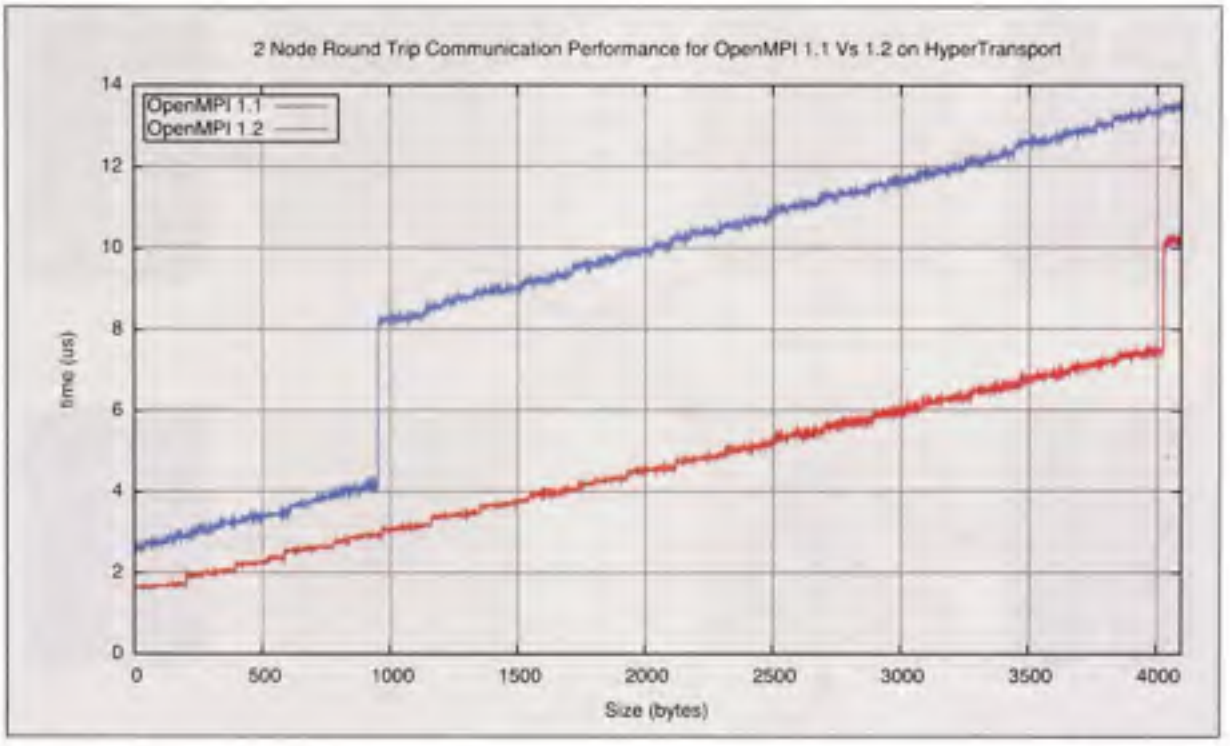


Figure 1.9 : Comparing OpenMPI versions 1.1 and 1.2 on HyperTransport by varying the message size passed to the `mpptest` micro-benchmark. The 1.1 implementations had performance issues characterized by a sudden jump in communication times around packet sizes of 1000bytes.

1.3.4 Benchmarking Network Communications

This section's performance assessment were obtained using *mpptest* [22], which uses the local MPI implementation for it's inter-process communications. Through our experimentation, we have confirmed that OpenMPI's ancestor, LAM-MPI, possesses better overall TCP/IP performance as seen in Figure 1.6 . This is a known issue and is due to OpenMPI's team concentrating on high bandwidth, low latency interconnects such as HT, Infinipath, Myrinet and others. This strategy also explains the improvements seen in Figure 1.9 where performance leaps were observed between the 1.1 and 1.2 release of OpenMPI running on HT links of a Tyan $VX50$.

We also note that, contrary to normal intuition, asynchronous (non-blocking) communications are actually slower than synchronous (blocking) communications in all cases of the `mpptest`

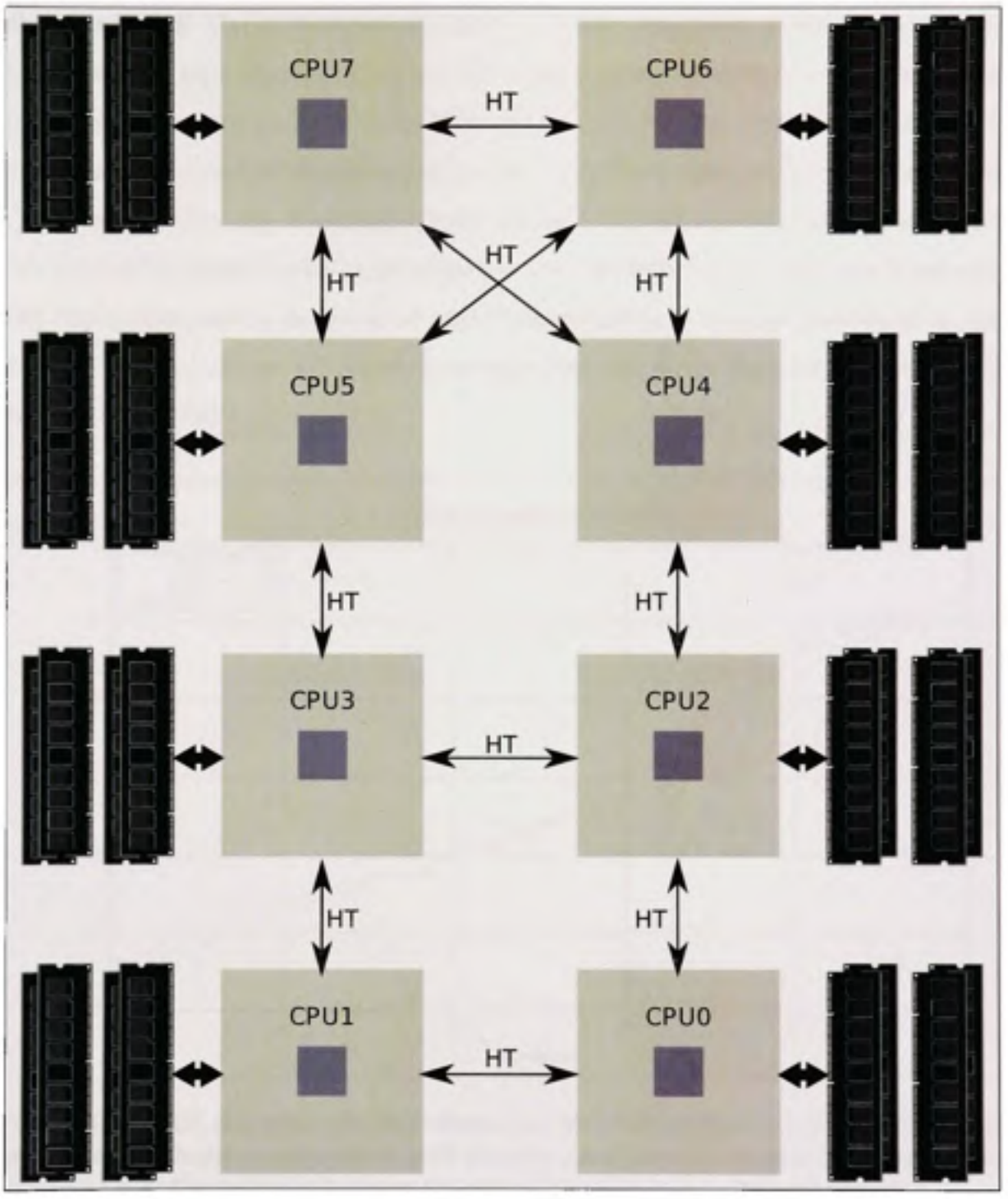


Figure 1.10 : The Tyan VX50 interconnection strategy for 8 processors using HT. This twisted ladder topology provides for an average 1.5 hop between processors and their farthest memory pages.

micro-benchmark. This is illustrated by Figure 1.11 where the fastest communications are of the synchronous type, followed by the persistent type (lagging behind by a few μseconds) and, finally, with almost 10 μseconds delay added are the asynchronous communications. This is due to the fact that the MPI libraries are only active when being called and executed actively by a program. The only way to guarantee this is during a synchronous call, where the execution path is linearized and forces the communications to complete before any other task is engaged. This implies that, barring the use of an explicit *helper* thread to keep the libraries *alive*, synchronous communications will remain faster than their asynchronous counterparts, even with the presence of CMPs.

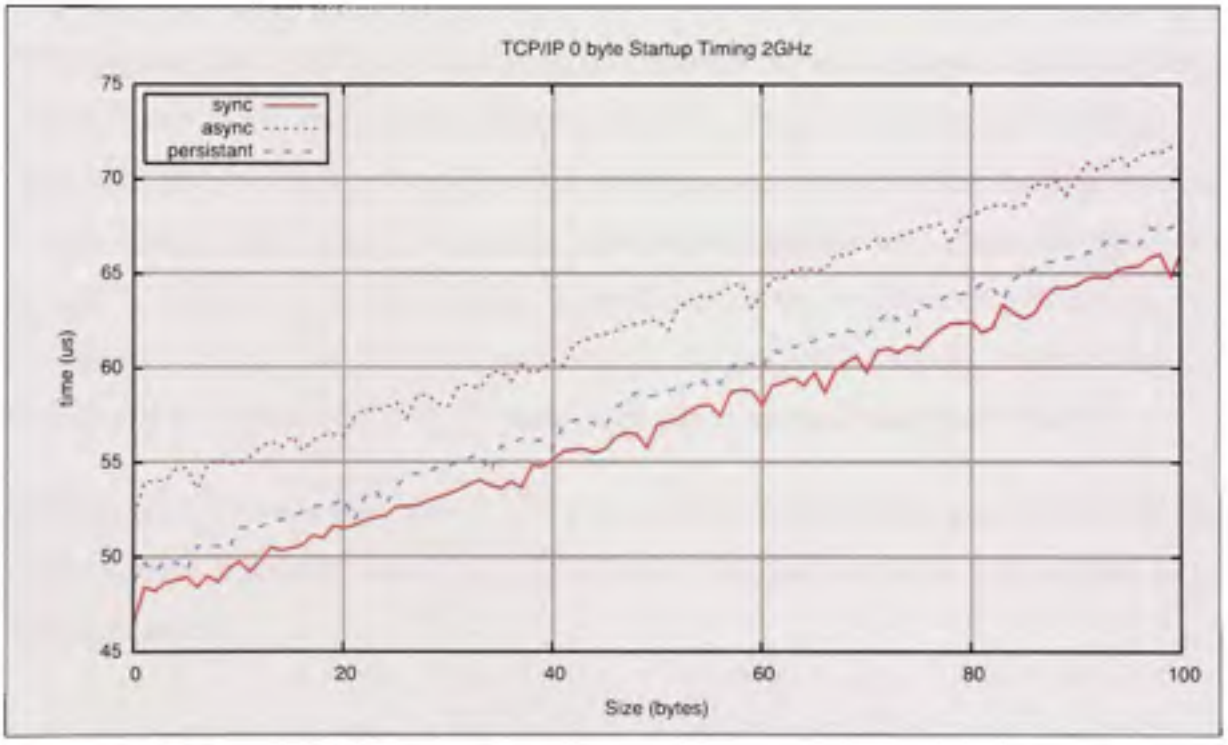


Figure 1.11 : MPI call types and their impact on the communication times. Synchronous (sync) communications outperform both asynchronous (async) and persistant ones as the processor is dedicated to performing the communication task in that specific case.

1.3.5 Theoretical and Empirical Model

Communication modeling is dependant upon logical and topological distributions. Nonetheless, Eq. (1.4) can be viewed as a generalized equation of Point to Point (PtP) communications[21] where t_{comm} is the total communication time which is t_s, the setup time (or latency), added to the cost per byte t_{byte} times the message length L (payload).

$$t_{comm} = t_s + t_{byte} \times L \tag{1.4}$$

To verify the validity of this generalization, we present Figure 1.12 , with which we are able to demonstrate that the theoretical model is adequate for packet sizes between 1 and 64 bytes and packets beyond 16 kbytes. The discrepancy between 64 and 16 kbytes can be explained with the non-linearity introduced by Ethernet's MTU, as they were presented in Figure 1.7 . The value of $t_s \approx 53\mu s$ is from `mpptest`, hence closeness of the initial theoretical values and this tool's results. Note that there is no *theoretical* definition for t_s, being ideally 0. We use $t_{byte} = \frac{1byte}{11.88Mbyte/s} \approx 84\eta s$, where $11.88Mbyte/s$ is the useful bandwidth as described in section 1.3.1. An arrow is inserted at $188Bytes$, the payload for sending a single vector of dimension $d = 47$ floats, a size which comes in handy in our case study in Chapter 3.

Although not all shown here, these results were cross-validated using popular microbenchmarks included in the HPC Challenge (HPCC) suite [35], the `mpptest` [22] and `netpipe` [53] applications.

21. These are the simplest and most common form of communications used.

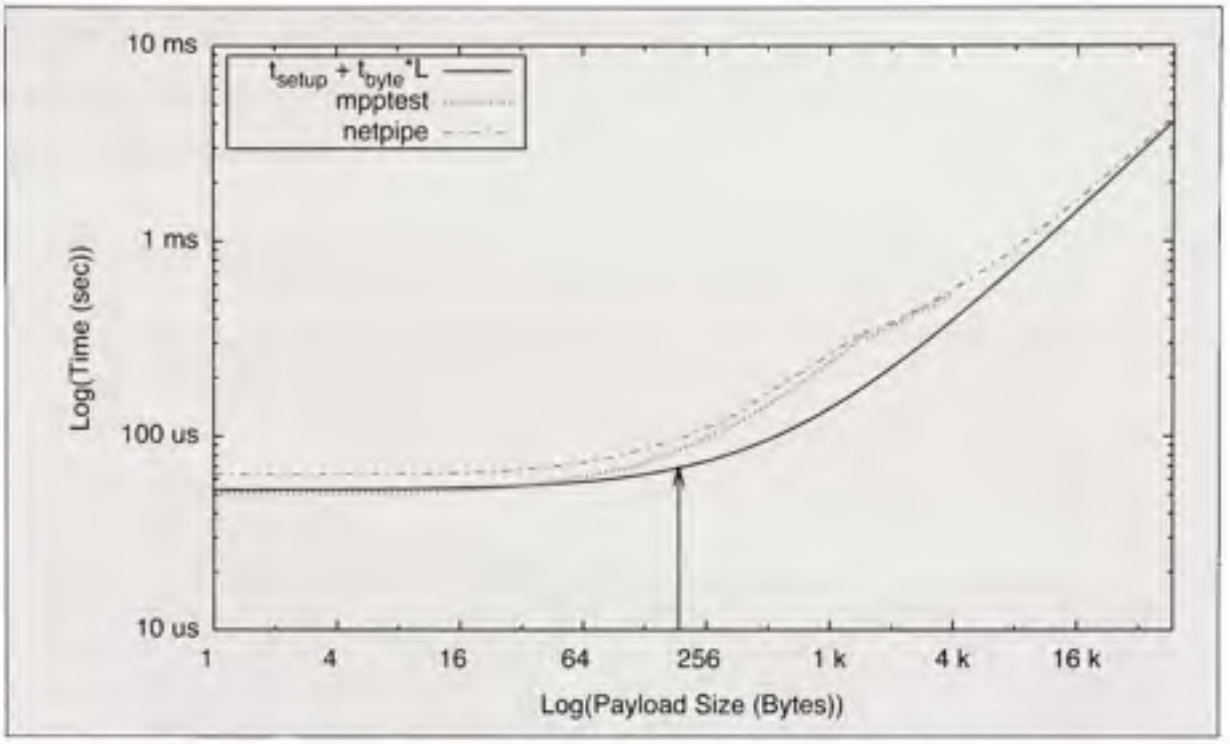


Figure 1.12 : Comparing the general theoretical communications model with empirical values for a $100BaseT$ Ethernet network. Results from `netpipe` are slightly higher than `mpptest`, indicating there might be additional overhead to his test suite. The theoretical value bases its t_s on results from `mpptest`, thus biasing it to be closer to that tool's results. An arrow is inserted at 188bytes as a point of reference for a vector of 47 floats, a unit which comes in handy in our case study.

1.4 Input/Output and Storage

Discussions concerning I/O and storage strategies are usually relegated to a transitory state of a program and judged as being non-essential or non-contributing to an application's overall performance given its single occurrence either at loading or termination of a given program. This type of assumption only remains true if $t_{load} \ll t_{exec}$, the loading time is significantly less than the total execution time. We investigate this assumption in Figure 1.13 where each function's time contribution is represented as a percentage of the total runtime. This stacked representation clearly illustrates each function's proportional shift as the number of processors augments for this parallelized algorithm. Bringing our attention to the loading function `load_samples()`, which accounts for less than 10% of the runtime for two nodes, we see that

it grows to a proportion beyond 40% when executed on 24 nodes. This is far from being negligible and brings about the importance of considering an application in its entirety when dealing with performance.

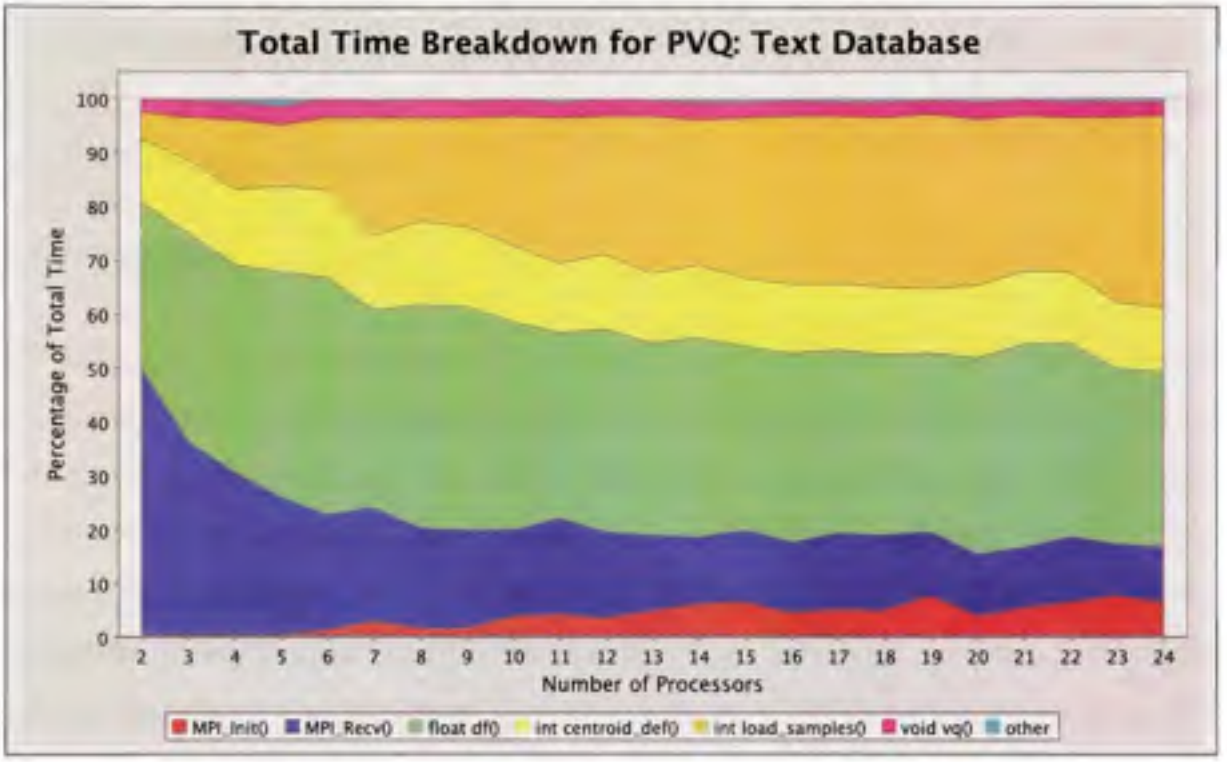


Figure 1.13 : An example of proportional breakdown of each task's contribution to the execution time for the PVQ implemented using a textual database (described in Chapter 3) and traversing its entirety at initiation. The loading of the data is performed by the `load_samples()` function and represents a significant portion of the total execution time.

1.4.1 Local Versus Remote Storage

In the context of Beowulf clusters [42], it is common to have nodes booted off a network share such as Network File System (NFS) as well as having the user's work directory mapped across the nodes through the same means. Given the usage simplicity provided by this approach, one might wonder if it remains relevant to use local storage used as *scratch space*. With local storage, a single path is drawn from I/O to RAM and the bottleneck resides in the slowest element, the HDD. This means the local bandwith $BW_{I/O}^{'node}$ can be expressed as being equal

to the HDD's bandwith ($BW_{I/O}^{node} = HDD_{I/O}^{node}$)[22]. In the case of an SMP machine, since we are dealing with a single task running at a given time, we can express this bandwith as a fraction of itself over the number of cores, thus rendering $BW_{I/O}^{node} = HDD_{I/O}^{node}/c$, where c is the number of cores. Noting that the available bandwith is shared among n hosts such that Eq. (1.2) becomes BW_{useful}/n, local scratch space remains beneficial as long as $BW_{I/O}^{node} > BW_{useful}/n$ or the remote server's own local bandwidth $BW_{I/O}^{node} > BW_{I/O}^{server}/n$, which must also be shared among all nodes.

1.5 Discussions

In this chapter, we have confirmed that processor caches are critical performance enhancing components used to mend the gap between processing speed and data access latency. Two of the cache's performance paradigms, *contention* and *capacity*, have been empirically identified and zoned for two common CMP processors, Intel's $Q6600$ and AMD's Opteron 885. Communication considerations were then brought up by our exploration of the available fabrics, notable the commodity 100 BaseT Ethernet and the high bandwidth, low latency HT. Issues with the underlying communications library, notably OpenMPI's implementation of the MPI standard were identified. More specifically, we demonstrated that the legacy LAM-MPI implementation of the Transmission Control Protocol (TCP)/Internet Protocol (IP) stack outperforms the one from OpenMPI on Ethernet based fabric. On the other hand, OpenMPI has concentrated their efforts on high speed fabrics, for which we have noticed marked improvements on their use of the HT links with considerable performance enhancements. The comparison of a theoretical model based on initial empirical data was shown to be adequate with slight divergences surrounding non-linearities imposed by hardware limitations such as the MTU. Data access and format issues were also brought up with an example of application scalability being hampered due to storage format. Furthermore, simple *rule of thumbs* were established to justify the use of local scratch space. Finally, the following recommendations can be made when applying this chapter's theory to our class of problem implemented using MPI:

22. We include all configurations of Redundant Array of Inexpensive Diskss (RAIDs) (and their redundancy/bandwidth enhancements) as part of the definition of HDDs for the sake of simplicity.

Processor selection:

- Problem size (or segmentation) must consider the processor's cache size or risk incurring significant performance loss;
- Memory access time remains important for performance in the case of memory bound processing, which is our case;
- When compared, it has been established that a larger cache with faster memory access is preferable for memory bound problems.

Communications fabric selection: The HT fabric is more efficient than Ethernet based solutions. Nonetheless, the cost of HT based SMP remains high compared to an equivalent Beowulf based cluster using commodity Ethernet fabrics such as GigE. The fact that CMP processors are now commonly available also emphasizes this cost factor since we are now seeing the emergence of clusters of SMPs. Since our application is rather memory bound than communication bound, we retain no benefits to the low latency brought by HT.

Data storage and location: Local data storage for *scratch space* comes out as a definite necessity as communications fabrics are rapidly overwhelmed by the amount of data to be transferred. And when it's not the communications fabric, the server's I/O path becomes an issue.

CHAPTER 2

THE PROFILING TOOLS

Although there are myriads of system based tools such as `dstat` (display of global system activity), `top` (per process statistics), and even some that are specialized for cluster monitoring such as `cacti` [1] (cluster wide equivalent of `dstat`), these can only convey an opaque view of the system usage. Fine grained specifics such as which function is using up all the processor or which system call is taking an abnormally long time to complete cannot be presented by such tools. This is where profiling comes in to automate the identification of functions and the collection of their execution statistics.

Profiling provides the contribution of a block code to some *execution metric* of a program. These metrics vary from call counts, time, and many other hardware accessible counters as will be presented with the use of Performance Application Programming Interface (PAPI). Profiles are meant only to convey a global statistical view of the total execution time. Even though a *call graph* [21] may be generated to interconnect code blocs of an overall execution, the sequence in which they are called in time cannot be reconstructed. This is because call graphs are based on aggregated profile data which is compiled in a *post processing* phase, after the program has been executed and has exited. Therefore, *time* measurements and call graphs collected by profiles do not permit a chronological reconstruction of events (function call sequence). This type of information is from the realm of program *traces* which we do not cover as they are more appropriate for code coverage analisys as well as time-sensitive troubleshooting (deadlocks in concurrent accesses and communications).

Given the many profiling tools available, we will concentrate on the ones freely available [2] since our main interest is their usage and not their comparison.

1. `http://www.cacti.net/`
2. This also excludes tools which are *free to try* during short periods such as a month or so.

The following chapter is organized as follows, we start by defining our use of the terms *Black*, *Grey* and *White Box* profiling, then briefly describe the context in which the tools are used (program and relevant parameters). Following are the tools themselves, starting with `gprof`, the classic GNU is Not Unix (GNU) profiling utility, where we identify its limitations in the context of parallel HPC. The Tuning and Analysis Utilities (TAU) suite is finally presented as a much more elaborate and appropriate alternative, applicable to the complex environment of parallel processing.

2.1 Black, Grey and White Box

In the realm of software engineering, the terms *Black Box* [40] and *White Box* [17] refer mostly to code coverage and reliability with the intention of identifying faults, failures and unexpected behaviors.

We adapt these terms for our specific usage to describe the conext in which the *performance profiling* is to be performed as well as its impact on the resulting program. These definitions are presented in Table 2.1 below.

Term	Description	Impact on the program	Impact on the source code
Black Box	The source code is unknown and only the compiled program (binary) is available.	None	Not applicable
White Box	The source code is known and the profiling is performed with explicit tooling, by the programmer, of the source code prior to compilation.	Some performance loss due to inserted profiling code.	The source code is tooled and the programmer is responsible for ensuring such tools can be switched off. It is also implied that the right functions are being profiled (a priori identification of the bottlenecks).
Grey Box	The source code is known but the profiling isn't explicitly performed within the original code. An external mechanism is used to add profiling code to the end program.	If the profiling is not performed selectively, significant performance loss is to be expected (all functions suffer from the profiling overhead).	None, the tools insert the profiling mechanisms in an intermediate step of the compilation.

Table 2.1: Black, Grey and White Box definitions.

As we present each tool, we will identify its capabilities as well as its use with regards to these different "Box" approaches to profiling.

Throughout this chapter, we will (ab)use the same program (PVQ) that is described and thoroughly analyzed in Chapter 3. The input and output parameters for the program execution are irrelevant in most figures that will be presented. In most cases, these parameters are nonessential and are merely set as such to provoke aberrant cases with the intent of providing visual material from a real program in its execution context. The input parameters of the algorithm, generally listed in the legend, can therefore be ignored as they were explicitly set to demonstrate specific use cases, caveats or aberrations. Most titles will include the label `GET_TIME_OF_DAY`, which is the generic label to indicate the displayed metric is time.

2.2 Sequential Profiling: Use Of `gprof`

The `gprof` [21] utility is a companion to the GNU C Compiler (GCC) for profiling sequential applications. A quick way of obtaining a profile when using the GCC is by enabling the `-pg -g3` directives where `-pg` enables profiling and `-g3` enables code symbols for the code annotation feature. When profiling, no optimizations higher than `-O2` should be enabled, otherwise the generated profile will be incomplete and back referencing to the code will not work. For example, the command `gprof --brief -p vq`, where `vq` is the application name, can then be used to extract the profile information. This information is contained in the output file, `gmon.out`, generated after a *sample* run of the application has been performed [3]. The resulting output is presented in Figure 2.1 with the following columns:

1) %: The time proportion of time spent in that function (percent of total execution time);
2) cumulative seconds: The time for executing this function while **including** the child function calls;
3) self seconds: The time for executing this function while **excluding** the child function calls;
4) calls: The total call count;
5) self s/call: The time (in seconds) per call while **including** the child function calls;

3. Profiling does have a non negligible impact on code performance and is generally not suitable for long runs.

6) self s/call: The time (in seconds) per call while **excluding** the child function calls;

7) name: The name of the function in question.

It is clear that `df` is the predominant function in the program both in time and call counts [4]. The source code of the functions can also be tagged with their call count using `gprof -A vq`. Figure 2.2 contains the two most called functions for our example program.

gprof Output

```
eric@taurisse ~/I_Files/I_STS/I_Matrise/Code/K-Means.old $ gprof --brief -p vq
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   s/call  name
 95.94   192.84   192.84 378184369     0.00     0.00  df
  3.35   199.57     6.72 1877828     0.00     0.00  centroid_def
  0.53   200.64     1.07        2     1.07     1.07  load_samples
  0.47   201.58     0.94        1     0.94   200.51  vq
  0.00   201.58     0.00        6     0.00     0.00  average_distortion
  0.00   201.58     0.00        6     0.00     0.00  mean_vector
  0.00   201.58     0.00        1     0.00     0.00  centroid_init
  0.00   201.58     0.00        1     0.00     0.00  save_centroids
  0.00   201.58     0.00        1     0.00     0.00  show_centroids
```

Figure 2.1 : Output listing from `gprof -brief -p vq`. The columns describe the following metric for each function (each line): *% time* is the proportion of total execution time, *cumulative seconds* is the inclusive execution time, *self seconds* is the exclusive time, calls is the total count. *Self* and *total s/call* are for the inclusive and exclusive time per call. Finally, the last column holds the function name.

A *visual* call graph [21] can be generated from the profile as demonstrated in Figure 2.3 . This is not a feature from `gprof` but the result of calling the sequence of code in Figure 2.4 where a python script, `gprof2dot.py` [5], translates the output from `gprof` into the Graphviz [6] `dot` file format. This call graph draws the execution path of this simple program. Each box represents a function and the arrows indicate the call sequence. The percentages indicate the *inclusive*, or cumulative, time as one walks down the graph. Exclusive times are indicated in parenthesis.

4. How such observations are to be addressed is the subject of Chapter 3.6, suffice to say that this function is a potential bottleneck or *hot spot*

5. `http://code.google.com/p/jrfonseca/wiki/Gprof2Dot`

6. `http://www.graphviz.org/`

df and centroid_def functions

```
1       /* distance function - Euclidian Distance */
2       float df(float *v1, float *v2)
3    3781643968 -> {
4        int i;
5        float dist, sum;
6
7        sum=0.;
8        for(i=0; i<T; i++)
9          sum=sum+(v1[i]-v2[i])*(v1[i]-v2[i]);
10
11       dist= (float) sqrt((double) sum);
12       return dist;
13      }
14
15      /* classification of a sample taking into account each centroid */
16      int centroid_def(int pos, float *d)
17    14772628 -> {
18       int i,index;
19       float mdist, dist;
20
21       mdist=99999.;
22       for(i=0;i<NC;i++) {
23         dist=df(centroids[i].feat,samples[pos].feat);
24         if (dist < mdist) {mdist=dist; index=i;}
25       }
26
27       *d=mdist;
28       return index;
29      }
```

Figure 2.2 : The annotated source code as per the use of `gprof -A vq`. Only the two most called functions form the source code are presented.

2.3 The Itch Of Measuring Time

Although *time* is the most popular and intuitive metric, others such as Clock Per Instruction (CPI), FLoating point OPertaions per Second (FLOPS), cache *hits* and *misses* can also reveal pertinent information about a program's efficiency. The measurement of time is a non-trivial task when precision is essential [51, 18]. Although it is more critical in the case of *tracing* where real-time event sequences are reconstructed [36, 8], it also applies to the quality of the information gathered for application profiling [10]. This information has historically depended on software counters provided by system calls such as `gettimeofday()` for which the precision varies greatly depending on the Operating System (OS)'s implementation [51, 18]. We address this issue in the following section.

2.3.1 PAPI: Time To Scratch Below The Surface

Given the time measurement variance due to systemic perturbations [48, 41] as well as other factors such as cluster heterogeneity, one must question whether it is valid to base performance assessments solely on *time*. The other metrics we mentioned earlier, such as CPI, FLOPS and

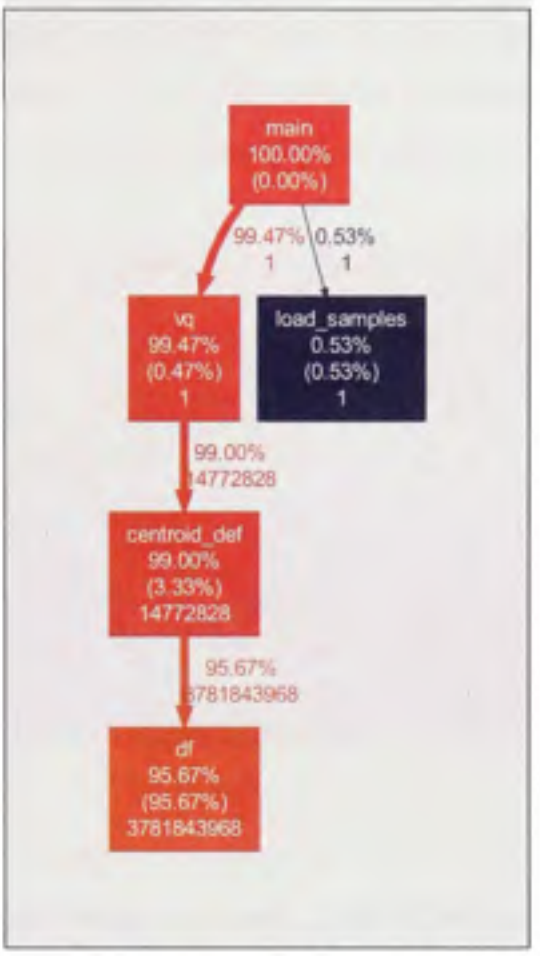


Figure 2.3 : The program call graph. This call graph draws the execution path of this simple program. Each box represents a function and the arrows indicate the call sequence. The percentages indicate the *inclusive*, or cumulative, time as one walks down the graph. Exclusive times are indicated in parenthesis.

Dot File Generation

```
gprof vq | gprof2dot.py | dot -Tpdf -o call-graph.pdf
```

Figure 2.4 : A sample use of `gprof2dot` to generate a `dot` file to be interpreted by Graphviz. The information is generated by `gprof`, then piped into `gprof2dot.py`, which itself pipes into the `dot` interpreter to generate the `call-graph.pdf` file.

all, are reputed as being generally more precise and useful [55]. They rely on the implementation of hardware counters [7] within a given processor or other peripheral such as sensors [11]. Accessing these metrics requires patching of the Linux kernel and software Application

7. Not to be confused with hardware *interrupts*.

Programming Interfaces (APIs), such as PAPI [3, 9, 10, 38]. Figure 2.5 is an adaptation of [10] which depicts the different software layers at which PAPI intervenes. We have added the explicit names of the support tools required by PAPI in parenthesis.

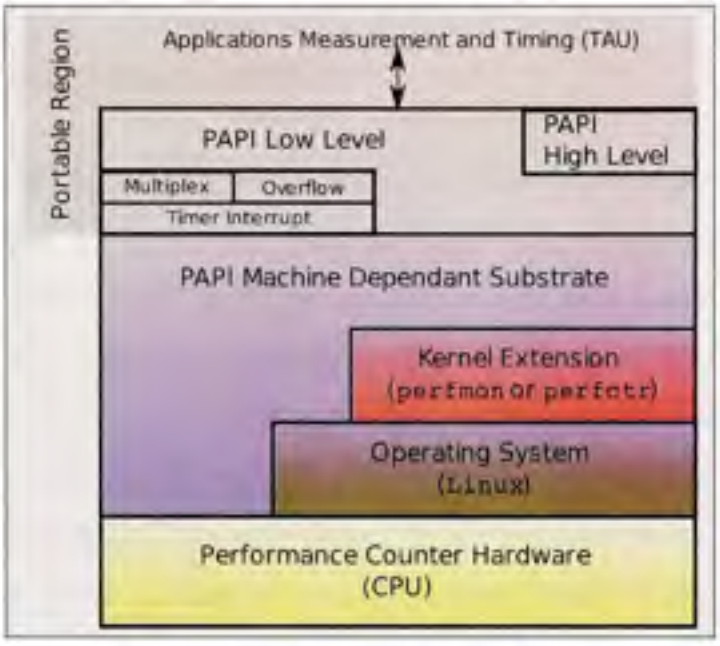


Figure 2.5 : The PAPI implementation scheme. Adapted from [10] to include the software components, in parenthesis, relevant to each layer used in our implementation.

Not all metrics can be counted nor are there as many counters as there are countable items [55]. For example, a processors being used may support only four simultaneous counters even though it is capable of probing well above 40 different events. A listing of such events, when PAPI is installed, is available for each machine in Appendix III. This is one of the limiting factors when selecting the desired statistic for collection. One must also note that some of these metrics are *derived*. These imply additional computation to be performed by the PAPI low level abstraction layer. The command `papi_avail -e <Name_of_PAPI_event>` can be used to display the metrics from which an event is derived. This adds to overhead to the profiling process [12, 55], which is detrimental to the quality of the resulting information. It is therefore suggested that non-derived metrics be chosen as to minimize their probing impact and that the derived metrics be computed as part of a post-processing mechanism. Such a feature

is well supported in the Tuning and Analysis Utilities, as we will demonstrate shortly. Lastly, since PAPI is an API, this means that one either has to insert tracing functions into their source code or use profiling tools with ability to use PAPI [33, 37], which inevitably brings us to the following section where we explore such tools that automate the tracing insertion process.

2.4 **Tuning and Analysis Utilities** (TAU)

We have demonstrated that using `gprof` is somewhat trivial but there are many drawbacks to this tool in our context. Firstly It only collects timewise and call count statistics. Secondly, and most importantly, it is not meant to be used in the context of parallel processing where one wants to keep track of all processes running on remote computers. This limitation renders `gprof` practically unusable. Also, we have presented PAPI, which provides an API for accessing the hardware counters present in modern processors. Unfortunately, this tool is not meant to automatically insert extra profiling and/or tracing functions into source code, such a burden being left to the programmer. Up until now, we have treated each tool individually and there is a clear need to consolidate these into a unified infrastructure to alleviate and make good use of each of their features.

> "Everything should be made as simple as possible, but not simpler."
> – Albert Einstein

This quote from Albert Einstein is shared with the TAU [46, 1] development team as the profiling and tracing of parallel processing application is a daunting task. Even more so when one adds the requirements of supporting multiple programing languages, compilers, hardware platforms and, above all, scalability [27]. But as we will demonstrate, the benefits of TAU's complex infrastructure is greatly outweighed by its features. Once the relevant features and components have been identified, its use provides simple yet powerful interfaces for both profiling and analyzing the collected data. Given this context, we will concentrate on using TAU's features which apply to our use, notably, the profiling of `C` and `C++` code, generated by GCC

with the ability to collect specific metrics thanks to availability of PAPI [8] and TAU's ability to use them [37].

2.4.1 Configuring TAU

TAU's features and the way it will probe a given program is selected when *compiling* TAU. For this reason, one usually generates multiple profiling and tracing configurations ranging from the simple and superficial profile *à la* `gprof` to more complex and elaborate probing configurations for trace and call-path reconstruction. There are essentially two categories for TAU's compilation options. The first category specifies which libraries, compiler and/or support applications (such as PAPI) will be used by the application to be profiled. The second category of options describe the type of profiling (measurements) to be performed [9]. Figure 2.6 is an example of how one would call upon the `installtau` script to automatically generate a general set of profiling configurations. Note that global options, such as enabling MPI profiling with `-mpi` on line 3, still have to be specified. In Figure 2.7 , lines 4 and 5 are examples of options describing the type of profiling to be performed. For example, the option `MULTIPLECOUNTERS` combined with PAPI's installation path (line 2), will result in a profiling configuration that will support he use of multiple PAPI counters.

TAU Automatic Configuration

```
1 ./installtau -prefix=$HOME/TAU/TAU -exec-prefix='uname -m' \
2 -papi=$HOME/TAU/PAPI/'uname -m' \
3 -mpi -pdt=$HOME/TAU/PDT/ \
4 && make -j4 install
```

Figure 2.6 : Automated general configuration of TAU using the `installtau` script.

The resulting libraries are named according to these flags as a mechanism of identification. Figure 2.8 presents a listing of the available configurations, each identified by their *stub Makefile*

8. Not applicable on all hardware platforms, refer to Appendix III.

9. As of version 2.18.1 of the TAU suite, the measurement infrastructure is being rewritten to make these options *run-time* selectable, therefore reducing the number of configuration stubs required.

TAU Manual Configuration

```
./configure -prefix=$HOME/TAU/TAU -exec-prefix='uname -m' \
-papi=$HOME/TAU/PAPI/'uname -m' \
-mpi -pdt=$HOME/TAU/PDT/ \
-PROFILECALLPATH -PROFILEPARAM \
-DEPTHLIMIT -MULTIPLECOUNTERS \
&& make -j4 install
```

Figure 2.7 : Manual configuration of specific features (lines 4 and 5) using TAU's `./configure` script.

names formatted as `Makefile.tau-<options>` where `<options>` relates to the aforementioned options. A specific configuration is selected by setting the environment variable `TAU_MAKEFILE` with the path to one of the *stubs*. The user then compiles his application using the *wrapper script*[10] instead of his usual compiler. Or, if the project has a `Makefile`, one includes the desired *stub* file and changes the compiler variable (typically `CC` or `CXX`) with TAU's wrapper script such that the compiler line becomes `CXX=$(TAU_CXX)`.

Available (compiled) TAU configurations

```
eric@h2 ~/TAU/TAU/x86_64/lib $ ls -1 Makefile.tau-*
Makefile.tau-callpath-mpi-compensate-pdt
Makefile.tau-callpath-mpi-pdt
Makefile.tau-depthlimit-mpi-pdt
Makefile.tau-mpi-compensate-pdt
Makefile.tau-mpi-pdt
Makefile.tau-mpi-pdt-trace
Makefile.tau-multiplecounters-mpi-papi-pdt
Makefile.tau-multiplecounters-mpi-papi-pdt-trace
Makefile.tau-multiplecounters-papi-pdt
Makefile.tau-multiplecounters-papi-pthread-pdt
Makefile.tau-param-depthlimit-multiplecounters-mpi-papi-pdt
Makefile.tau-param-mpi-pdt
Makefile.tau-pdt
Makefile.tau-phase-multiplecounters-mpi-papi-compensate-pdt
Makefile.tau-phase-multiplecounters-mpi-papi-pdt
Makefile.tau-pthread-pdt
```

Figure 2.8 : Example of TAU profiling options that were compiled at installation time. Following the `Makefile.tau-` filename prefix are the options selected at compilation time.

2.5 Profiling the Source Code

Profiling of the source code can be done in one of three ways:

10. Typically, `tau_cc.sh` replaces `gcc` for `C` and `tau_cxx.sh` replaces `g++` for `C++`

1) Automatically insert extra profiling functions using TAU's integration of Program Database Toolkit (PDT) [34];
2) Semi-automatically insert traces using a Graphical User Interface (GUI) editor such as Eclipse with the TAU integration modules [49];
3) Manually insert function calls to keep track of called events as well as the time spent in these events.

We will demonstrate the use of the first two approaches and leave manual integration of accessing TAU's APIs for other advanced projects such as auto-adaptive parallel codes [29]. In both cases, the original source code remains intact.

2.5.1 Automatic Code Insertions

If TAU is configured with the `-pdt` option, it is possible to let the wrapper scripts insert tracing code automatically. This approach is as simple as compiling the code normally with the exception of changing the compiler name and having the `TAU_MAKEFILE` environment variable set. Using this approach, the entire program will be profiled and, if the selected profiling configuration includes options such as `-mpi`, these function calls will be uniquely identified. This approach is probably the best one to use when performing initial profiling of an unknown code base (peering into the black box).

2.5.2 Semi-Automatic Code Insertions

It is possible to perform *selective profiling* of an application while preserving the integrity of the source code. This is accomplished via a selection file which is passed onto the wrapper script as an option. The simplest way [11] to accomplish this is to use TAU's Eclipse [12] modules [49] for selective profiling. Figure 2.9 is an example usage where we select a function (`df()`) for profiling. Two types of user-defined events can be selected, *start/stop* events and *atomic* events. The first one defines a type of counter that collects metrics at the entry and exit of the

11. From the user's point of view. The administrator has to go through the installation of multiple Eclipse modules to get such features working correctly.
12. `http://www.eclipse.org/`

selected region while the latter counts the occurrences of the selected region. Atomic events are tagged as *user events* in the generated profile and can help keep track of program dynamics such as *iteration counts*.

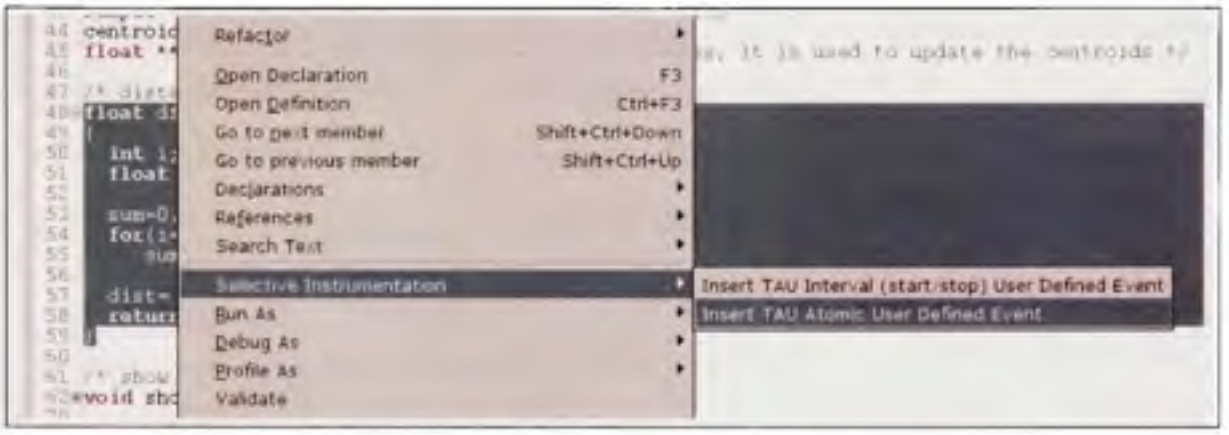


Figure 2.9 : Selective profiling using Eclipse and TAU's selective instrumentation interface. The `df()` function is selected and specific type of profile pattern is applied to it. The modules then automatically generates a `tau.selective` file to be passed to the wrapper script.

2.6 Executing the Profiled Code

In the case of parallel and distributed environments, the collection of profile information requires more attention that simply executing the program and running the profile viewer. Although doing *just that* will provide a valid profile with TAU, unexpected disaffects such as excessively long runtimes can be experienced depending on the selected profiling options. This is the case for options such as `TRACE` and `CALLPATH` where the resulting files tend to be quite sizeable [13]. Given the multitude of environment variables required to fine tune the profiling process, we present Figure 2.10 which is a sample script used for the profiled execution of a program.

13. A typical profiling run for one of our applications generates $500k$bytes of data while the same application will generate over $1.4G$byte of trace data.

2.6.1 Selecting The Profile Depth

The first variables of interest are `TAU_CALLPATH_DEPTH` and `TAU_DEPTH_LIMIT`, from lines 4 and 5, which guide the depth at which the profiling must take place. For example, a `TAU_DEPTH_LIMIT` of 2 applied to the call graph of a program, such as the one described by Figure 2.3 , would generate a profile containing the statistical information for `vq()` and `load_samples()` only, as these are the ones on the second level below the `main()` invocation.

Program Runscript.sh

```
1  #!/bin/bash
2  trap 'exit 1' 2 3
3
4  export TAU_CALLPATH_DEPTH=1
5  export TAU_DEPTH_LIMIT=2
6
7  # COUNTER1 enforced by the use of MULTIPLECOUNTERS and PAPI.
8  export COUNTER1=GET_TIME_OF_DAY
9  export COUNTER2=PAPI_L2_TCM
10 export COUNTER3=PAPI_L2_DCM
11 export COUNTER4=PAPI_TOT_CYC
12
13 for ITER in 'seq 1 30'
14 do
15    for I in 'seq 2 18'
16    do
17       PROF_DIR="PAPI_MPI_Trace_iter_${ITER}/PAPI_MPI_Trace_${I}"
18       EXP_DIR=/data/eric/$PROF_DIR
19       cexec -p "mkdir -p $EXP_DIR" >/dev/null
20       export TRACEDIR=$EXP_DIR
21       export PROFILEDIR=$EXP_DIR
22
23       orterun -x COUNTER1 -x COUNTER2 -x COUNTER3 -x COUNTER4 \
24       -x PROFILEDIR -x TRACEDIR \
25       --prefix ~/openmpi_i686/ -hostfile nodes -np $I \
26       ./pvq_i686_TAU /data/eric/fest_trainc_342910
27
28       mkdir -p $PWD/$PROF_DIR
29       cexec -p "mv $EXP_DIR/* $PWD/$PROF_DIR && rmdir $EXP_DIR"
30    done
31 done
```

Figure 2.10 : A sample script that sets up the environment for multiple runs of profiling.

2.6.2 Selecting The Desired PAPI Events

Another set of important variables are present between lines 8 and 11. These are used to define the PAPI counters that will be used by TAU to profile the application. The first variable (`COUNTER1`) has to be set to `GET_TIME_OF_DAY` as it is used as a reference to synchronize the individual traces provided by each independent node.

2.6.3 Controlling The Data Flow

As we have mentioned earlyer, the profiling and tracing of an application can generate sizeable amounts of data. In lines 17 to 21, we configure the variables `TRACEDIR` and `PROFILEDIR` to point to local storage and are identified based on the experiment's parameters. Lines 23 to 26 is the actual command line to launch the experiment. All environment variables are propagated to all nodes thanks to the `-x` option from `orterun` [5], OpenMPI's parallel process launcher. Once the application completes its execution, lines 28 and 29 transfer the resulting traces from the node's local storage to the server for post-processing of the data.

2.6.4 Storing The Data

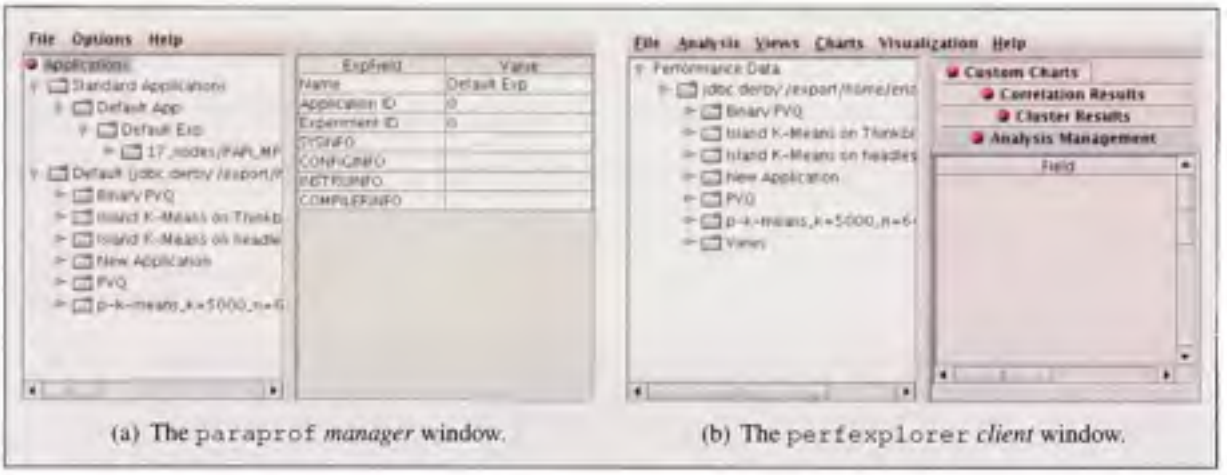
Performance Data Management Framework (PerfDMF) [27] is an interface to multiple types of databases [14], which leverages the use of the TAU suite for keeping track of the evolution of an application's performance as the codebase changes. Although it is not required for viewing profiles [15] per say, it is so for more elaborate analysis such as the ones that can be performed by the `perfexplorer` component [16], which can only access profile data through the PerfDMF interface. It also enables greater collaborative efforts as the standardized storing of the data eases distributed accessibility.

The tools provided by PerfDMF are command line oriented and meant to ease the configuration of the GUI tools and automate the insertion of trial data. Although injecting the data into the database can be accomplished through the use of `paraprof`'s GUI, which requires manual loading of each resulting profile for each experiment [17], the command line tool `perfdmf_loadtrial` is profiled to automate the process. Although this component does not play an active role in the profiling process, it plays a critical role in the decoupling of the profiling process from the analysis.

14. Version 2.18.1 of TAU supports PostgreSQL, MySQL, Oracle and Derby, a local file-based database.
15. We will present `paraprof` shortly.
16. Yes, also presented shortly.
17. A quick looks at Figure 2.10 reveals that one would have to perform $30 * 17 = 510$ manual insertions!

2.7 Paraprof and PerfExplorer: The Profiling Graphical User Interfaces

TAU provides two independent GUIs for interpreting the profiled application's data. The first we present is `paraprof` [2], used specifically for profile analysis. The second tool, `perfexplorer` [26], is used for performance and scalability analysis. In both cases, they are composed of a *main* window, as depicted in Figures 2.11 (a) and 2.11 (b), for selecting the data source.



(a) The `paraprof` *manager* window.

(b) The `perfexplorer` *client* window.

Figure 2.11 : Both GUIs possess a *main* window from which the data set(s) to be analyzed is selected. The selection is performed in the left pane where trials are presented in the form of a tree structure. The latter depends on how the data was imported using PerfDMF. We see in (a) that `paraprof` has an additional branch, which is used for the current folder's data and that (b) possesses an additional leaf named *view*.

In the case of `paraprof` (Figure 2.11 (a)), an additional tree is present since this application can be used in a stand-alone fashion (without the use of the database backend). Contrary to this, `perfexplorer` explicitly depends on the data being stored in a database, hence the single tree seen in Figure 2.11 (b). Note that `perfexplorer` has an additional leaf entry named *views*, this will be discussed with the use of `perfexplorer` itself.

2.7.1 The Paraprof Profile Viewer

Paraprof is a GUI component that provides a simple yet powerful presentation of the collected profiles. It has the ability to read the profile data from a multitude of different formats including `gprof` generated profiles. When used with TAU's suite, simply starting the application in the directory containing the output files is sufficient for the application to load and display the data. This implies that it can be used in a *stand-alone* mode, without requiring the connection to a database. The loaded profile can also be stored or retrieved to and from the database if so configured. Figure 2.12 is the first data representation displayed if `paraprof` is started within a directory containing profile data. By default, the bars are normalized, which makes the standard deviation (Std. Dev) seem disproportionate compared to the observed results.

Figure 2.12 : A normalized profile view of all processes including the global mean and the standard deviation (Std. Dev.) of each functions. In this case the metric is the time proportion as per `GET_TIME_OF_DAY`. Each color represents a specific function and its length is proportional to the total execution time on that specific node.

What makes this GUI interesting is the fact that any presented information has a contextual menu granting access to additional information. One can start from a global view of the entire execution and iterate down to the source code. This is also true for the hardware's *metadata* which is accessible through the contextual menu presented when hovering above the node

names. Such information becomes important when performance analysis is performed as well as for keeping track of the historical evolution of a given program.

The presented data can also be filtered by selectively hiding functions, or group of functions, through the function (or function group) legend windows. This is presented in Figure 2.13 where all functions are enabled but one group is selected, which adds emphasis to the selected group in the bar graph window. We also changed view configuration to present de-normalized and unstacked bars, as an alternative to the one in Figure 2.12 .

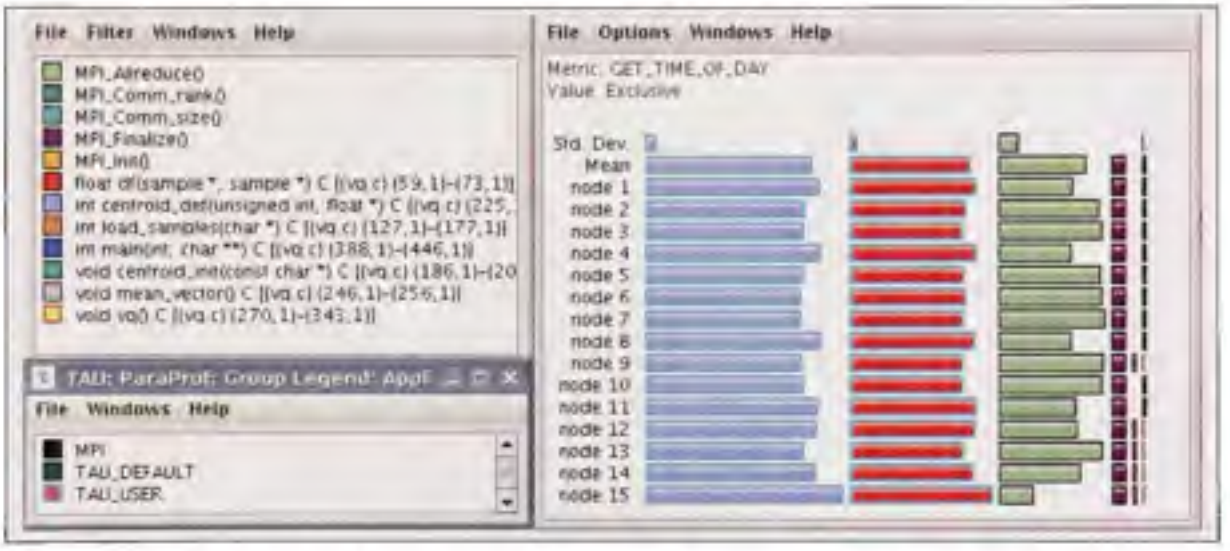


Figure 2.13 : Individual functions and group of functions can be selected to focus the displayed statistics. Here, the `TAU_USER` group is selected in the *Group Legend* pane (bottom left), which highlights the relevant functions in the main window (right). Note that we have de-selected the stacked bar presentation for the main window to present an alternative to the normalized stacked bars from Figure 2.12 .

The 3 Dimensional (3D) view, in Figure 2.14 , provides a more intuitive and dense analysis of the collected statistics when compared to Figures 2.12 and 2.13 . This view should therefore be the first one to be used to gain a rapid perspective of the program's behavior. The barcharts can *then* be used for a more in-depth analysis as they provide a complete mechanism for accessing all the data relating to each element of the program within its context [18].

18. There is no contextual menu in the 3D presentation linking a given component to its metrics nor to its section of source code.

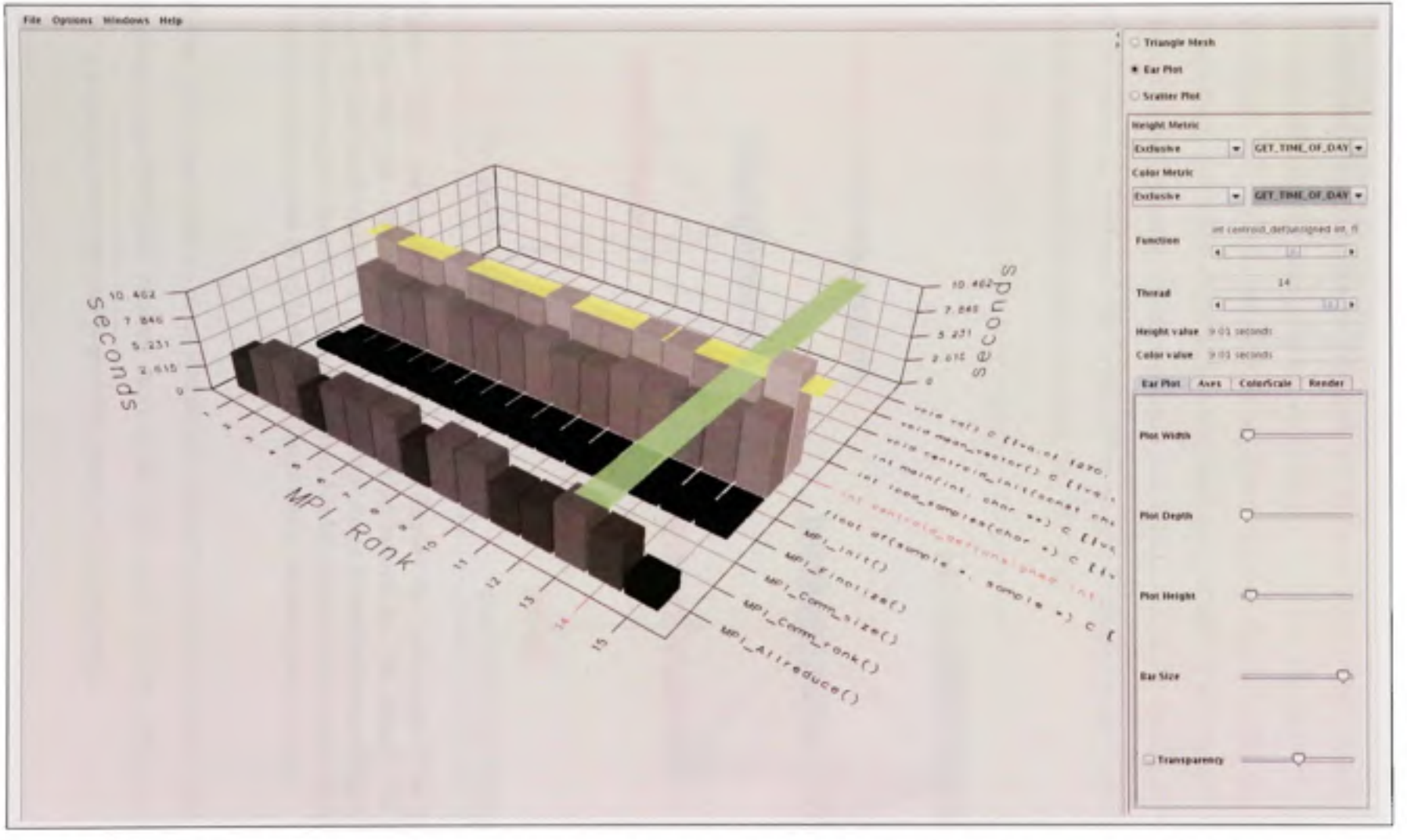
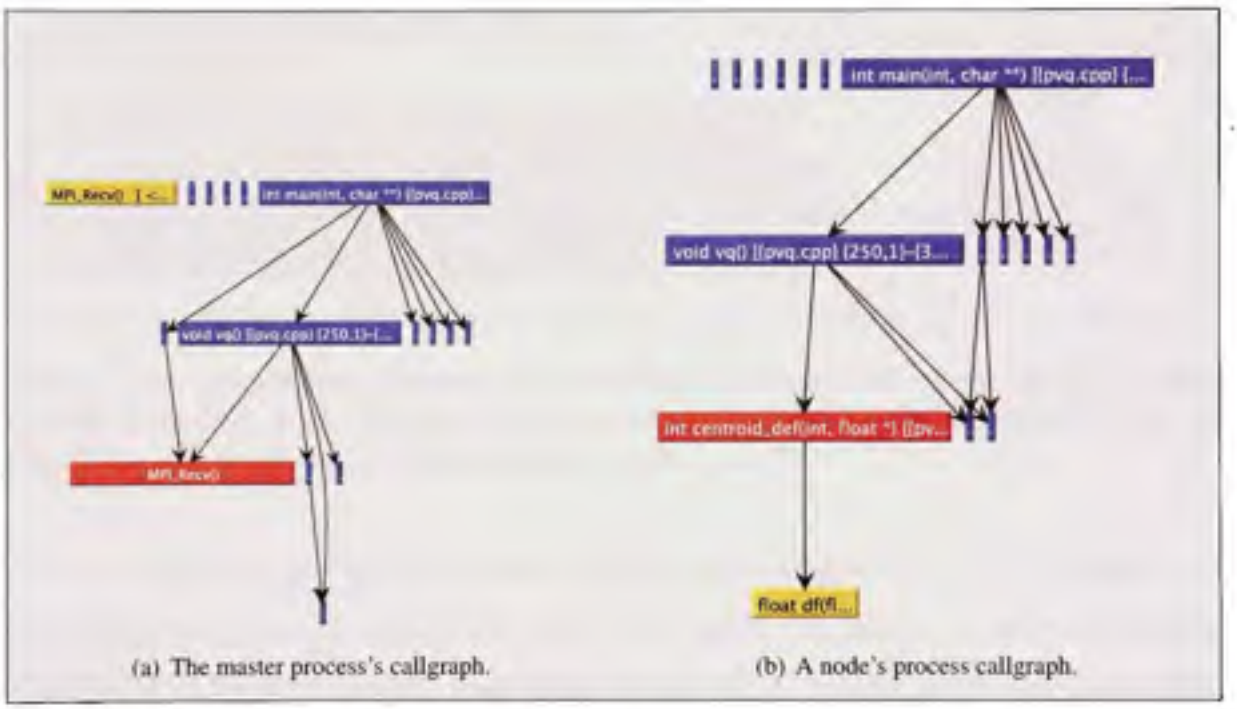


Figure 2.14 : An alternate representation of the data in 3D. This view provides a more intuitive view of the data through a landscape representation or a series of bars (as shown). The bar height and color intensity can relate to any of the collected PAPI metric or one of the derived metrics created by the user.

It is also possible to visualize the callgraph of a given execution thread. Figure 2.15 presents two such graphs for the same program where Figure 2.15 (a) is the graph of a master process while Figure 2.15 (b) is the one for a slave process. The box size and color are guided by their relation to selected metrics such as inclusive time and exclusive times. As with the barcharts, contextual menus grant access to contextual information such as the source code of a function and its statistics. Selecting one of the functions also highlights it in all other `paraprof` windows displaying this function (except for the 3D view).



(a) The master process's callgraph. (b) A node's process callgraph.

Figure 2.15 : `paraprof` has the ability to display the call graph if the program was profiled with the `-PROFILECALLPATH` option turned on. By default the box width is proportional to the *inclusive* times and the box color is selected according to the *exclusive* runtime of a given function. Both programs are the same but it is clear that the call path from the master node in (a) is different from one of the slave nodes in (b).

Figure 2.16 is an example of an interaction sequence presenting the path from profiled data contained in the PerfDMF database down to the source code from the profiled application [19].

19. A path to the source code must be provided is none is currently configured.

Figure 2.16 : An example of an analysis sequence in `paraprof`. From top left, circling counter-clockwise, is the sequence from `paraprof` manager window, through the bar charts, the call graph and then to the source code.

In all cases, `paraprof` can only be used for the punctual analysis of a given execution with a fixed context (such as the number of nodes). This implies that it is not the preferred tool for scalability and efficiency analysis and should be used for the performance analysis of a fixed context.

2.7.2 The PerfExplorer Performance Analyzer

As a sister application to `paraprof`, `perfexplorer` [26] is a more elaborate graphical front end specially created for statistical and efficiency analysis of parallel profiles. This tool is geared at providing an insight on a parallel program's scalability and efficiency, given some real-world runs of a program under possibly differing conditions. As such, it is expected that multiple profile runs will be executed with per-run variations such as the number of used nodes.

Given that scalability is a central concern to parallel processing, this tool is geared at giving as much information as possible in that respect so that one can quickly identify scaling issues.

Although `paraprof` provided a complete view of a single profile, it lacked the ability to convey tendencies that can only be obtained by comparing different profiled runs of a program. These tendencies are the application's *speedup* and *efficiency*.

2.7.3 Application Speedup

The *speedup* (S_p) is traditionally defined as per Eq. (2.1), a ratio between t_1, the time for a single process execution and t_p, the time for executing its parallel counterpart with p processes. This evaluation of the speedup holds true as long as the *program scaling* is *strong*, meaning that the computational load is not changed as machines are added. In such a case, ideal speedup is in direct proportion to the number of processors. Simply put, if there are p processors, the ideal speedup (and ultimate goal) is that a parallel application should run p times faster than its sequential equivalent. Linear speedup is seldom possible to attain unless the application is *embarrassingly* parallel, meaning that it will perform computation more than anything else for any given p processors. Although the *"anything else"* is historically bound to communications, we will demonstrate later that there are other factors that influence the application's speedup, and therefore, scalability.

$$S_p = \frac{t_1}{t_p} \tag{2.1}$$

The programmers of `perfexplorer` don't assume the reference execution of a program to be a single process and define a *baseline* execution time t_{base} of a given program that is based on the first available timing sample, which is not necessarily executing sequentially on as a single process. This leads to what they call *relative speedup*. This leads to a slight redefinition of Eq. (2.1) as Eq. (2.2) by replacing the unit time t_1 with a *base* reference time t_{base} which is not bound to a single thread execution.

$$S_p = \frac{t_{base}}{t_p} \tag{2.2}$$

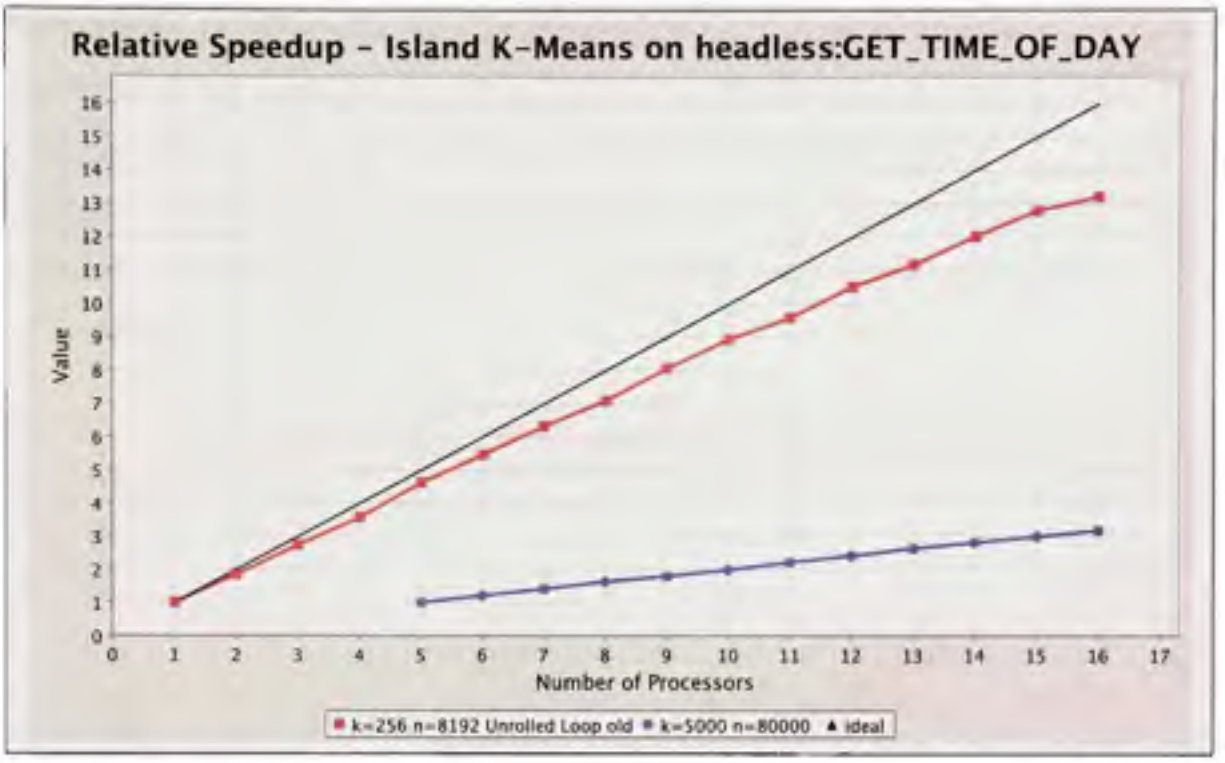


Figure 2.17 : The top line shows the *ideal* speedup, based on the experimental data right below it, which starts with $t_{base} = t_1$ (1 processor) up to the timing for $p = 16$ processors. The bottom line seems to have poor speedup as it is far from the *ideal* line (also drawn). For this curve, the baseline time t_{base} is based on the execution with $p = 5$ processes. This induces a distortion in the speedup representation as the two series have a different reference for t_{base}.

The direct implication is that an ideal speedup is not necessarily equal to the number of processors but rather a scaled factor of t_{base} by p_{base}, the number of processor used for the base run. To illustrate this, we ran the test program with artificially low and high computation characteristics and with a different number of worker nodes to start with. We then draw the speedup for both execution cases in Figure 2.17 . The ideal speedup is drawn top most, then the first experiment with low computational load starting at $p = 1$ and finally, the lower most curve is for he experiment starting with $p = 5$ processors with very high computational load. Both runs are executed up to using 16 processes. Although the bottom most run would seem to possess a lower speedup, a closer look in Figure 2.18 indicates that the application is in fact exerting ideal speedup characteristics according to `perfexplorer`.

Given relative speedup is being used, to be able to compare both executions, the program would have to permit the scaling of t_{base} by p_{base2}/p_{base1} for a true *comparison* to be possible when presented within the same graphic. A normalized result is obtained by setting $p_{base1} = 1$, which would give a scaling factor of 5. This does confirm that the second experiment, which had displayed a seemingly poor speedup of 3.2, actually has an ideal speedup of 16 ($3.2 \times 5 = 16$) after rescaling.

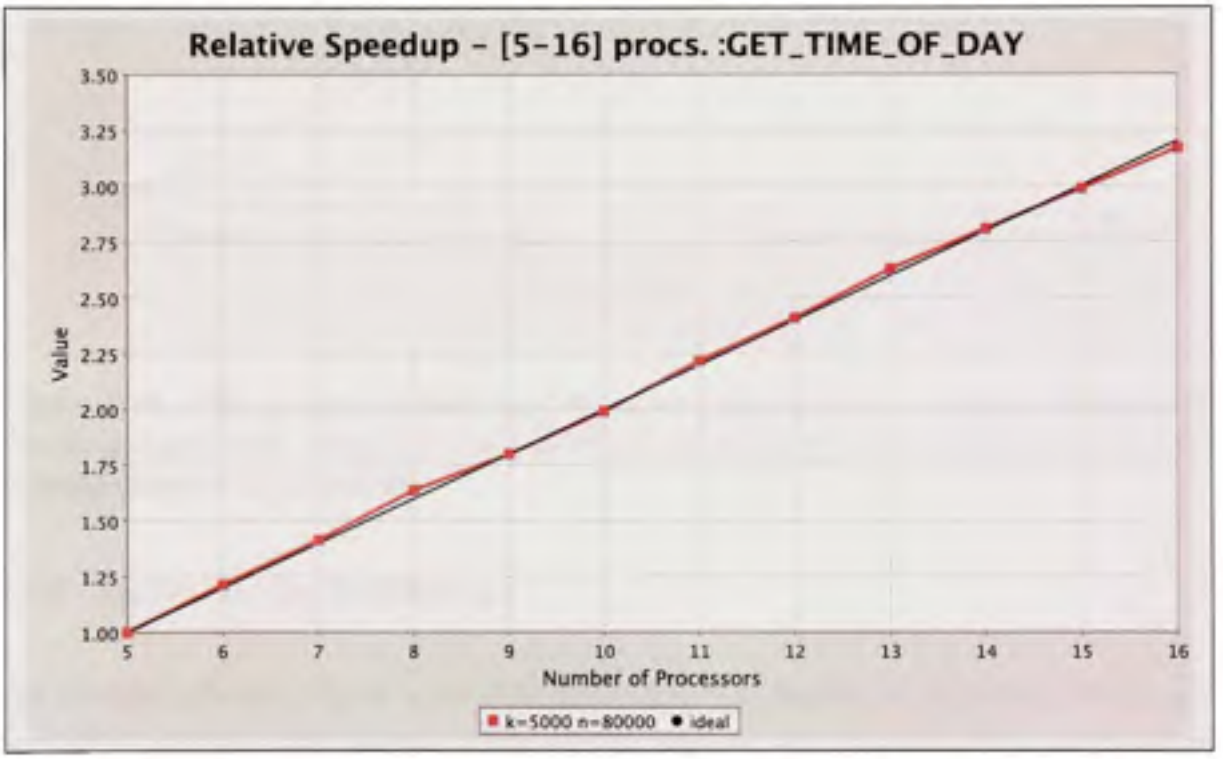


Figure 2.18 : A closer look of the experiment having a baseline time t_{base} with 5 processors demonstrates that it actually exerts *ideal* speedup according to `perfexplorer`'s guideline.

Another feature is the ability to display per-function speedups, which can help identify functions that will become problematic as processors are added. Figure 2.19 is one such example where the functions falling off below the ideal speedup curve are the most probable candidates of becoming scalability bottlenecks.

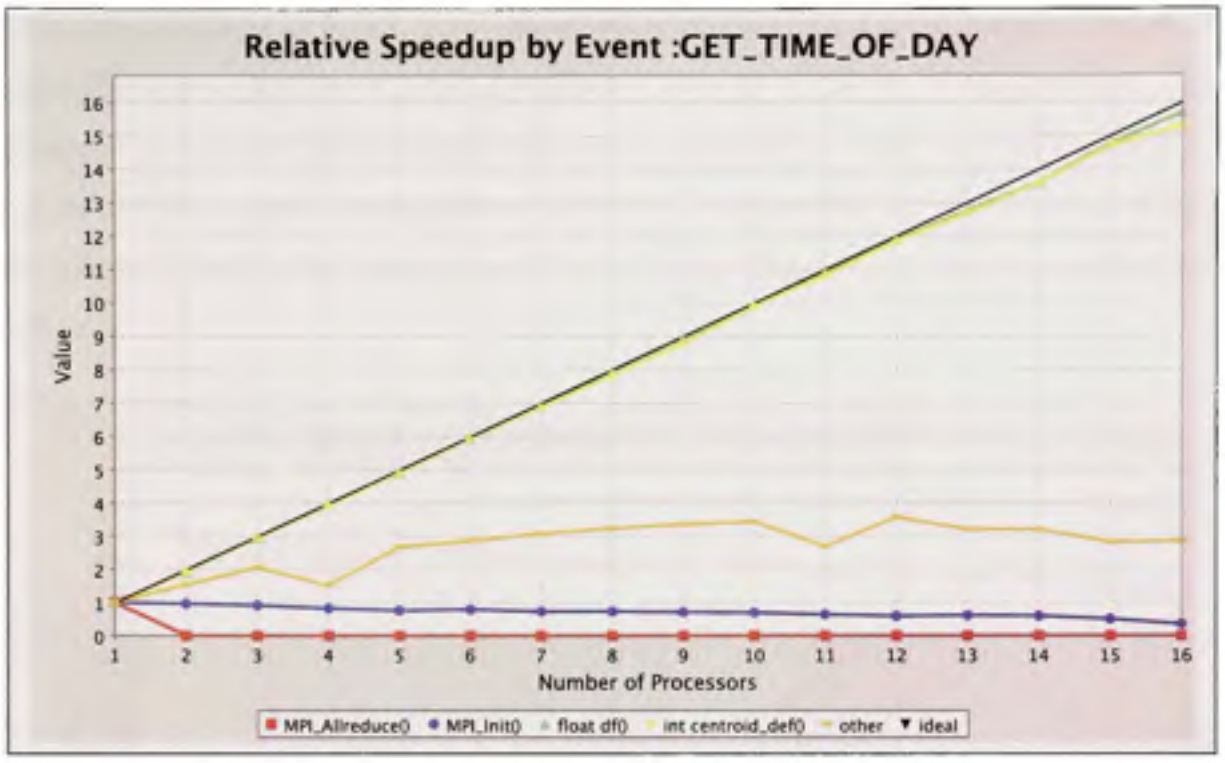


Figure 2.19 : The speedup of each event is drawn independently to isolate the functions that do not scale well. Functions that fall off the ideal *speedup* reference line are the most probable barriers to scalability.

2.7.4 Application Parallel Efficiency

The *parallel efficiency* E_p of a parallel application is a measure of its ability to use processors as they are added to the execution environment. Its general form is in Eq. (2.3), which is the speedup from Eq. (2.1) normalized over the number of processes p_{count}. Since `perfexplorer` doesn't assume the baseline time t_{base} is for a single process (or $t_{base} = t_1$), Eq. (2.3) is redefined as *relative* efficiency in Eq. (2.4). This is once again a variation on the speedup, this time scaled by p_{base}/p where p_{base} is the count of processors used for the baseline execution.

$$E_p = \frac{t_1}{(t_p \cdot p_{count})} \tag{2.3}$$

$$E_p^{rel} = \frac{t_{base} \cdot p_{base}}{t_p \cdot p} \tag{2.4}$$

As it is demonstrated in Figure 2.20 , the relative efficiency is a more appropriate measure of scalability when comparing different algorithms in differing contexts. The implementation that seemed to have initially poor speedup is in fact more efficient. The ideal is to keep the efficiency to 1 (or 100%) as p grows. As it is the case with the speedup graphs, it is also possible to display the relative efficiency for each element of the executing program, as demonstrated in Figure 2.21

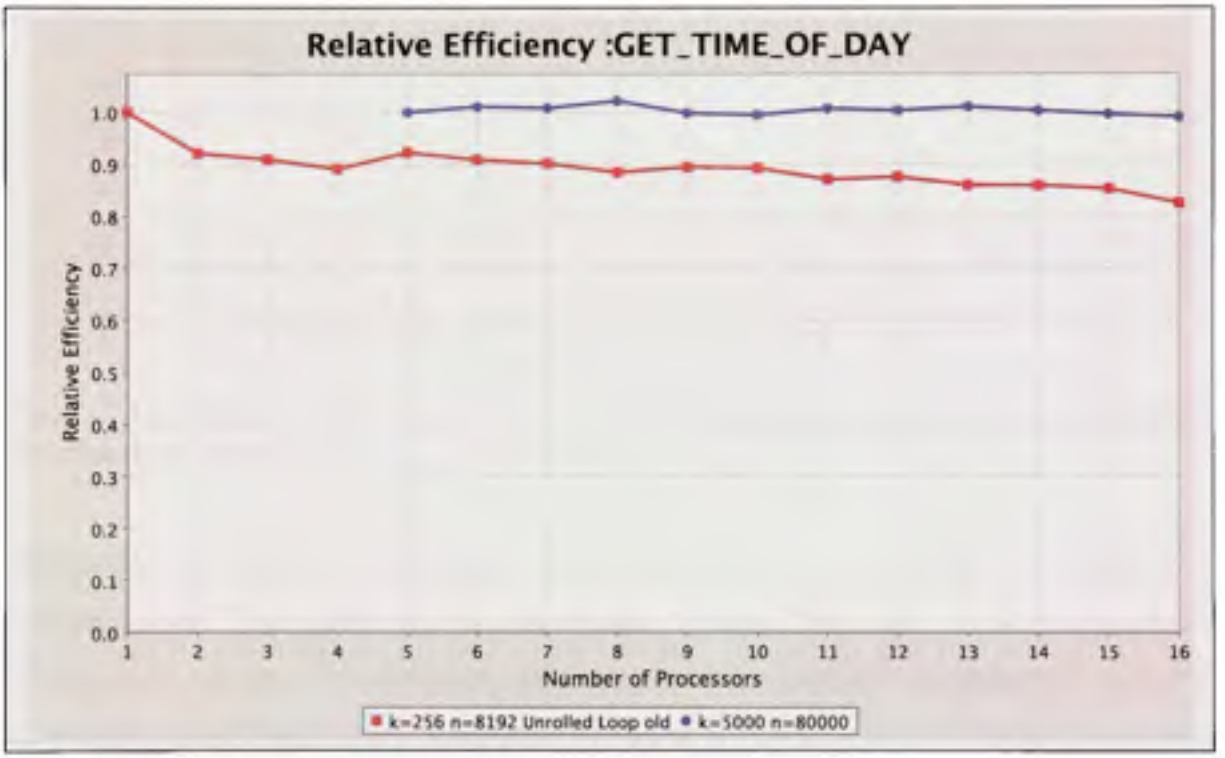


Figure 2.20 : Relative efficiency is not affected by the baseline's processor count p. The most efficient implementation (top line), averaging at 1, was originally presented as having comparatively poor speedup in Figure 2.17 .

2.7.5 Runtime Breakdown

The speedup and efficiency graphs are the *de facto* metrics to characterize a parallel process in its execution environment. As a complement to these representations, the *runtime breakdown* is composed of stacked areas representing each function's proportional contribution to the total execution time. This is a more intuitive view of the per-component performance progression

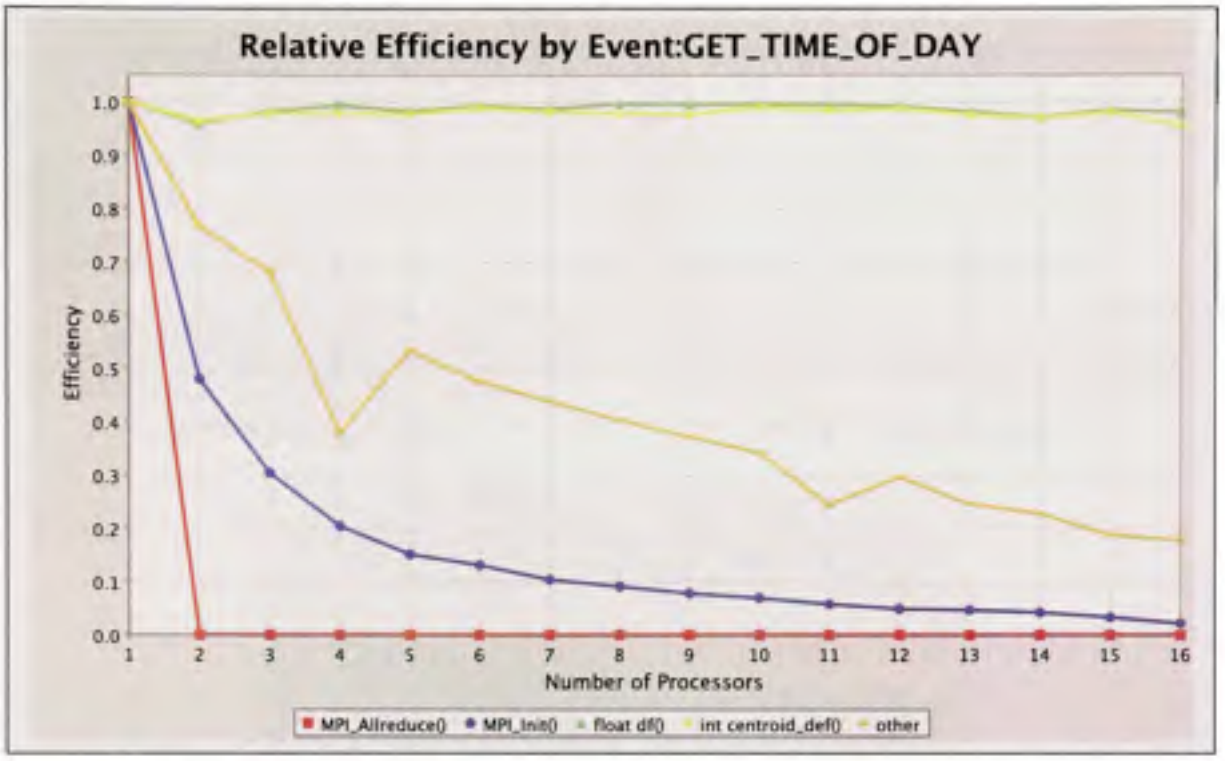
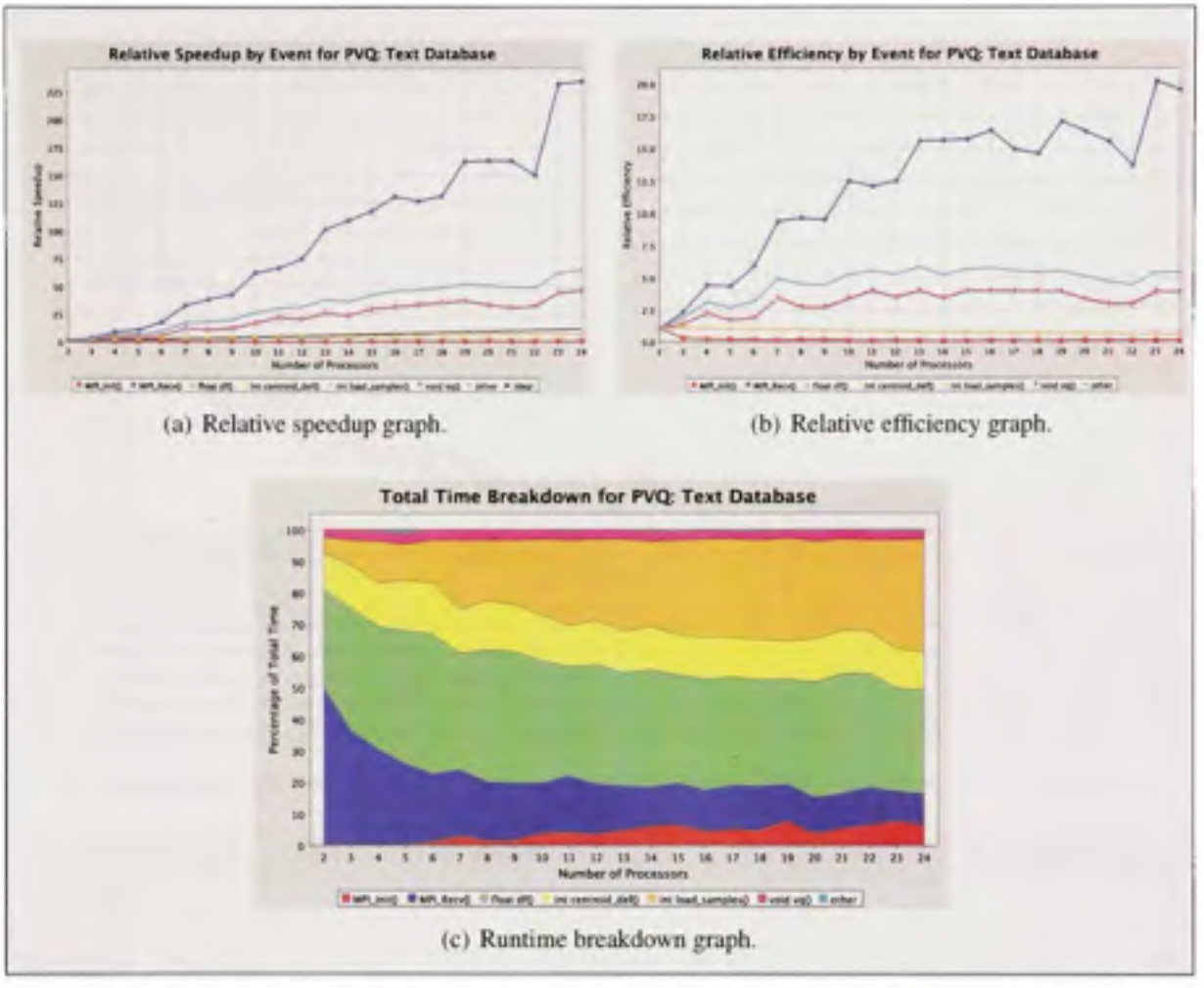


Figure 2.21 : Relative efficiency *by event* can help identify functions with poor scalability. The ideal is to remain close to 1 as processor count grows.

as processors are added. As an example, Figure 2.22 presents the same problem under three different angles. The relative speedup and efficiency graphs from Figure 2.22 (a) and Figure 2.22 (b) don't convey a view as intuitive as presented by the runtime breakdown in Figure 2.22 (c). Functions that present poor scalability will grow in surface area as processors are added. A quick glance at the runtime breakdown identifies these problematic functions quickly and efficiently.

2.7.6 Views

An additional feature of `perfexplorer` is its ability to create views based on the data present in PerfDMF. For example, if we have 15 iterations of the same experiment, displaying the resulting runtimes as in Figure 2.23 (a) isn't practical. Creating a view consolidating all iterations into a single experiment enables `perfexplorer` to display averaged statistics as



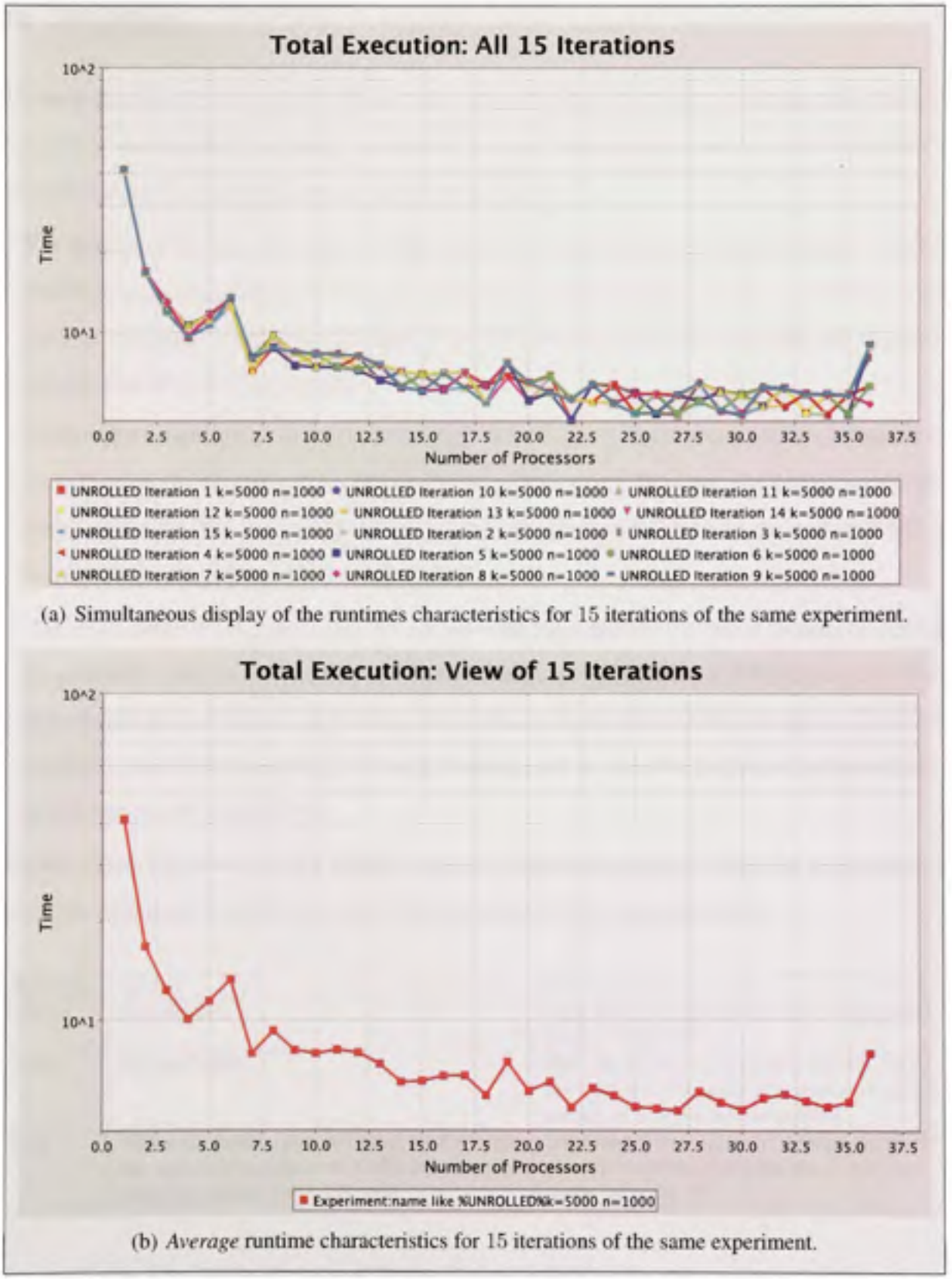
(a) Relative speedup graph.

(b) Relative efficiency graph.

(c) Runtime breakdown graph.

Figure 2.22 : Comparing three representation of the same profile run using relative efficiency in (b), relative speedup in (a) and a runtime breakdown graph in (c). The intuitive display from the runtime breakdown eases the identification of functions becoming problematic as processors are added. Simply put, a widening cone such as the second predominant layer from the top, is indicative of a growing bottleneck. A tightening cone, on the other hand, means that the function looses proportional importance in the overall execution time. Parallel or constant area are signs of linear (ideal) speedup of a function.

we see in Figure 2.23 (b), where the drawn cureve is the average runtime of the same 15 experiments.



(a) Simultaneous display of the runtimes characteristics for 15 iterations of the same experiment.

(b) *Average* runtime characteristics for 15 iterations of the same experiment.

Figure 2.23 : The use of `perfexplorer` views help consolidating experimental data for a better analytical perspective. All 15 experiments are presented in (a) whereas an averaged view is presented in (b).

2.8 Discussions

We have skimmed the surface of the vast and complex field of program profiling, especially in the case of parallel and distributed processing. The following points are our main observation concerning the tools we have mentioned in this chapter:

- The generic form of `gprof` is of little use to the parallel processing community when it comes to performance assessment. External tools are required for basic visualization tasks such as callgraph generation and there is no obvious means of consolidating the collected information across multiple runs.
- Modern processors provide internal counters which convey much more relevant information on the execution of a program other than time of execution. Performance enhancement and better understanding of a program's dynamics is accessible thanks to the use of PAPI in conjunction with profiling tools such as TAU.
- TAU is a complex yet powerful profiling suite that consolidates the entire process of manual or automatic code trace insertions, execution tuning, data collection and displaying of the results thanks to specialized GUIs oriented towards profiling and statistical analysis. This suite provides one of the most integrated and complete set of tools for program characterization in the context of parallel HPC.

Finally, Table 2.2 associates the different profiling approaches supported by the preseted tools/tollsets. It is more than obvious that TAU wins in all respects, hands down.

Box Type	`gprof`	TAU
Black	Not capable.	Only MPI calls through runtime interposition (library wrapers) [21]
White	Not applicable	One can manually insert TAU specific profiling calls for a fine-grained control over which portion of the code is to be profiled.
Grey	This is the *modus operandi* for `gprof` through the `-pg` option passed onto GCC. It is inappropriate for parallel HPC programs.	Supported in two ways: 1- through the use of selective profiling definition file 2- fully automated using the PDT.

Table 2.2: Black, Grey and White Box capabilities for the presented tools.

CHAPTER 3

CASE STUDY: PARALLEL K-MEANS ALGORITHM ANALYSIS

Unsupervised learning has become a popular field of study ever since it's infancy. One of the tried and true algorithms that keeps re-surfacing in one form or another is the k-means clustering algorithm. This data mining algorithm uses an iterative learning approach. Its training phase can prove to be very time consuming depending on the size of the dataset n, its dimension of vectors d, the number of centroids k and the number of iterations N_{iter}. The latter is a stop condition determined by an imposed convergence threshold δ. Given its computational complexity, represented by Eq. (3.1), it is not surprising that means to accelerating the k-means algorithm has been a point of interest since it's inception.

$$O(ndkN_{iter}) \tag{3.1}$$

We concentrate an implementation of the k-means used for unsupervised learning of segmented handwritten numeral strings as proposed by [44], Section 3, *Foreground-Background Feature Extraction (FBFE) Module*. In this context, k is varied using an EA in an attempt to obtain an optimal Hidden Markov Model (HMM). This requires many repeated learning phases, implying that any means by which the process can be accelerated can lead to a higher quality classifier and/or in less time.

Having access to a sequential and a parallel implementation (master-slave) of the algorithm, we will proceed as if we were performing a typical migration from the sequential to the parallel version as a means to validate the approach. With the help of *Grey Box* profiling results, we then propose a restructured version of the parallel algorithm (island model) which has both a simpler and more efficient implementation, statements which will be supported through comparative profiling of each key functions.

We therefore start by studying the sequential algorithm's profile, after which, we identify the potential parallel approaches. General parallelization considerations are presented for the al-

gorithm. The master-slave version is then analyzed to ascertain its adherence to the identified approaches and pin point probable scalability bottlenecks. An alternate parallel model is then proposed, the island model, with a restructured communication scheme which both simplifies and optimizes the code. In all cases, TAU is used to perform the analysis.

3.1 The Sequential k-means Algorithm

Given an unalbelled database DB of elements of dimension d represented as $\mathbf{x}^\intercal = \{x_1, x_2, \cdots, x_d\}$, we randomly select [1] k elements that are to become the centroids defined by $\mathbf{c}^\intercal = (c_1, c_2, \cdots, c_d)$, thus rendering the table of centroids C. For each elements in DB, we then identify the closest centroid $\mathbf{c_j}$ through Euclidean norm [2] (as denoted by $\|\cdot\|$). Once the *owning* identified (say, the j^th centroid), each element's values are summed into an intermediate value $\mathbf{c'_j}$ while keeping the count of elements in m_j. The new centroids are then computed by mean vector such that $\mathbf{c'_j} = \frac{c'_j}{m_j}$. The process is iterated until the convergence threshold $dist$ is lower than the allowed distortion, defined as δ. This convergence threshold is computed as the *average distortion*, which is the sum of the Euclidean norm between each element and its centroid over $|DB|$, the database size. This process is summarized in algorithm 2.

3.1.1 Empirical Evaluation of the Algorithm

As with most parallel programs, our implementation of the k-means first started as a sequential version of itself. A profile of the application is used to identify the most time consuming sections of the algorithm as well as its computational characteristics. Parallel strategies, such as partitioning, depend on the computational granularity of a program, a measure we will be able to obtain through profiling coupled with some knowledge of the implemented algorithm. Our profile information is obtained by compiling the program with TAU as presented in Figure 3.1 [3]. The parameters for this execution of the algorithm are $d = 17$ (vectors dimension is

1. Our implementation uses the elements located at each $|DB|/k$ interval so that the results would be deterministic between experiments, therefore comparable.

2. The Euclidean norm is the one we chose among many other *distance* computations as it is the one mostly used in our current experiments.

3. The extended version is avalable at Figure II.1

1: Random initialization of C by selecting k elements in DB
2: **repeat**
3: Clear out intermediate values $C' \leftarrow 0$, $dist \leftarrow 0$ and $m \leftarrow 0$
4: **for all** $\mathbf{x_i}$, where $1 \leq i \leq |DB|$ **do**
5: Identify closest centroid as per $\arg\min_k (\|\mathbf{x_i} - \mathbf{c_k}\|)$
6: Save the element's distance from the centroid $dist = \|\mathbf{x_i} - \mathbf{c_k}\|$
7: Add $\mathbf{x_i}$ to intermediate centroid: $\mathbf{c'_k} := \mathbf{c'_k} + \mathbf{x_i}$
8: Increment the centroid's element counter $m_k := m_k + 1$
9: **end for**
10: **for all** c'_j, where $1 \leq j \leq k$ **do**
11: Compute the new mean vector $\mathbf{c'_j} := \mathbf{c'_j}/m_j$
12: **end for**
13: Assign new centroids as current $C = C'$
14: Compute distortion $dist := dist/|DB|$
15: **until** $|dist| < \delta$

Algorithm 2: The Sequential K-Means

47), $k = 500$ (we want 500 centroids), $|DB| = 67879$ (we will use 67879 samples from the database). With a convergence threshold of $\delta = 0.001$, 9 iterations were required for the program to complete.

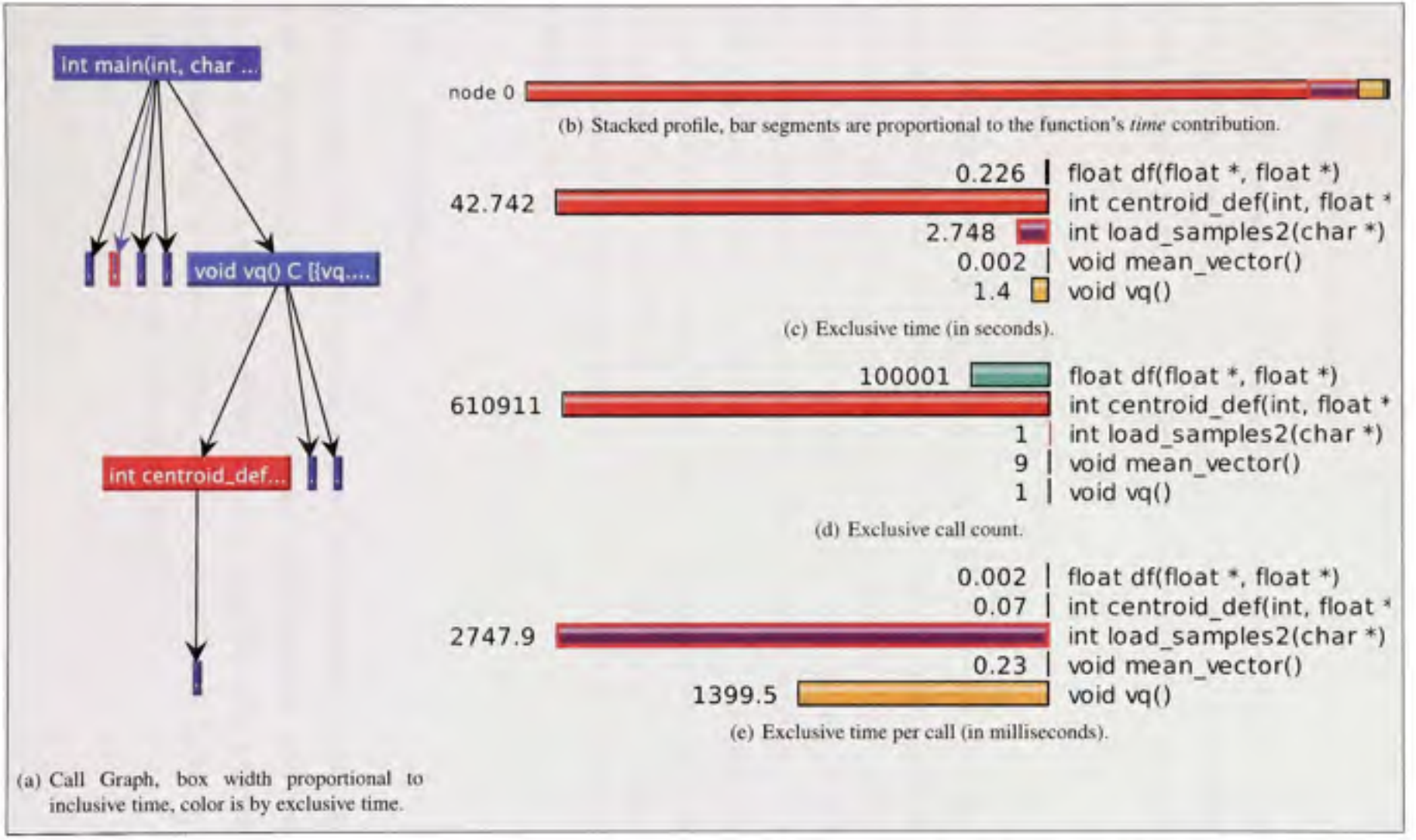
Sequential k-means profiling

```
$ export \
TAU_MAKEFILE=~/TAU/TAU/x86_64/lib/Makefile.tau-callpath-pdt
$ tau_cc.sh -optCompile="[snip]" vq.c -o vq_SIMD
$ ./vq_SIMD ./featg_col.dat $((6787955/100))
[snip]
```

Figure 3.1 : Profiling and execution of the sequential k-means algorithm using TAU**. The program is then started by specifying the reference database and the number of samples to load from the database. Here we load** 1% **of the entire database. The `[snip]` tags indicate output truncation.**

We present an analysis of the resulting profile in Figure 3.2 . First, the call graph in Figure 3.2 (a) indicates that the execution time (width of the boxes) runs down a direct path to the predominant function `centroid_def()`, being as wide as the `main` function. This observation is confirmed by Figure 3.2 (b) and Figure 3.2 (c), where the length of the bars representing

each functions are proportional to their time contribution. The function call counts presented in Figure 3.2 (d) come in handy with regards to a program's eligibility for parallelization as a high count and significant total execution time (from Figure 3.2 (c)) are indicative of probable partitioning for a given function. Since we are observing profiles and not traces, we have no means of establishing the level of each function's cohesion between calls. Nonetheless, we do have *a priori* knowledge of the implemented algorithm and know that the computation from the principal loop, at line 4 of Algorithm 2, can be performed on all n elements independently. Lastly, the time per call metric, from Figure 3.2 (e), gives a hint as to which functions might be scalability bottlenecks. This information leads to scrutinizing the `load_samples()` and `vq()` functions as their high time per call metric might indicate a single long task, susceptible of not being parallelized.



(a) Call Graph, box width proportional to inclusive time, color is by exclusive time.

(b) Stacked profile, bar segments are proportional to the function's *time* contribution.

(c) Exclusive time (in seconds).

(d) Exclusive call count.

(e) Exclusive time per call (in milliseconds).

Figure 3.2 : Each graphic is a window from `paraprof`, used to present a specific view of the sequential k-means profile. The call graph in (a) clearly shows that the execution time is mostly attributable to `centroid_def()`. The stacked bargraph in (b), and its deconstructed version in (c), also indicates this proportional importance. The call counts from (d) help identify potential *partitioning* areas as well as its *grain*. (e) is useful for identifying highly *cohesive* functions (many short calls), thus potential communication bottlenecks.

3.2 The Parallel K-Means Algorithm

As we will be presenting two parallelization approaches to the k-means algorithm, let's start by presenting common concepts for both approaches. We will start by presenting how the k-means may be subdivided followed by the implied communications required by the selected strategy.

3.2.1 First, Divide: The Segmentation Strategies

A typical approach to accelerating the resolution of massive amounts of loosely coupled calculations is to divide the calculated data into more manageable segments. This parallelization technique is applicable to the most basic form of the k-means algorithm, where, by definition, $|DB| \gg k$, meaning that the size of the database $|DB|$ is much more important than the number of desired centroids k. This implies subdividing the reference database DB amongst the ω workers, which we incrementally identify as $pid = 0, 1, 2, \cdots, \omega$. This approach is known as a coarse grained segmentation strategy of the problem with communications only performed between iterations. This is possible since the centroids aren't updated until a complete pass on the element database has been performed. The computation complexity form Eq. (3.1) therefore becomes Eq. (3.2).

$$O\left(\frac{|DB|}{\omega} dk N_{iter}\right) \tag{3.2}$$

3.2.1.1 Strided Segmentation

Two approaches to the segmentation are presented here, the first one is from [43] and is presented in Algorithm 3. It consists in assigning the $|DB|$ elements of the database to each ω worker by strides in a round-robin fashion. This is accomplished using a modulo function of the ω node count as the database is being traversed . In essence, this strategy ensures that the workload is subdivided as evenly as possible to all nodes.

1: **for all** x_i, where $1 \leq i \leq |DB|$ **do**
2: **if** $pid = i \mod \omega$ **then**
3: $DB^j_{pid} = x_i$ {the i^th sample is assigned to pid's local database}
4: **end if**
5: **end for**

Algorithm 3: Segmentation by Strides of the Database

3.2.1.2 Blocked Segmentation

The other approach, implemented for our island model, consists in subdividing the database into $|DB|/\omega$ large blocks. The rational behind this approach is that it provides a predictable access pattern for the hardware and therefore eases optimization through prefetching of the data. It also makes it possible to consolidate the loading of the data into a single system call. This strategy is described by algorithm Algorithm 4 where DB_{pid} is a local process's database to be initialized with a block of elements within the interval delimited by $pid \times |DB|/\omega <= i < (pid+1) \times |DB|/\omega$. In the case where there is a remainder to $|DB|/\omega$, they are assigned to the last processor. Figure 3.3 is a graphical representation of these two segmentation strategies applied to a database.

1: **if** $pid = \omega$ {If pid is the last process} **then**
2: $DB_{pid} := DB_i \mid pid \times |DB|/\omega <= i <= |DB|$
3: **else**
4: $DB_{pid} := DB_i \mid pid \times |DB|/\omega <= i < (pid+1) \times |DB|/\omega$
5: **end if**

Algorithm 4: Blocked Segmentation of the Database

3.2.1.3 Hardware Considerations: Load Balancing

The two presented approaches do not take into consideration possible variations in hardware characteristics between the computing nodes. These variations can come into play when there is a significant difference in processor performance where a slower node would slow down the process in whole. A common method of compensating such situations is to assign segments proportionally equivalent to the processing power of each node. This can be accomplished

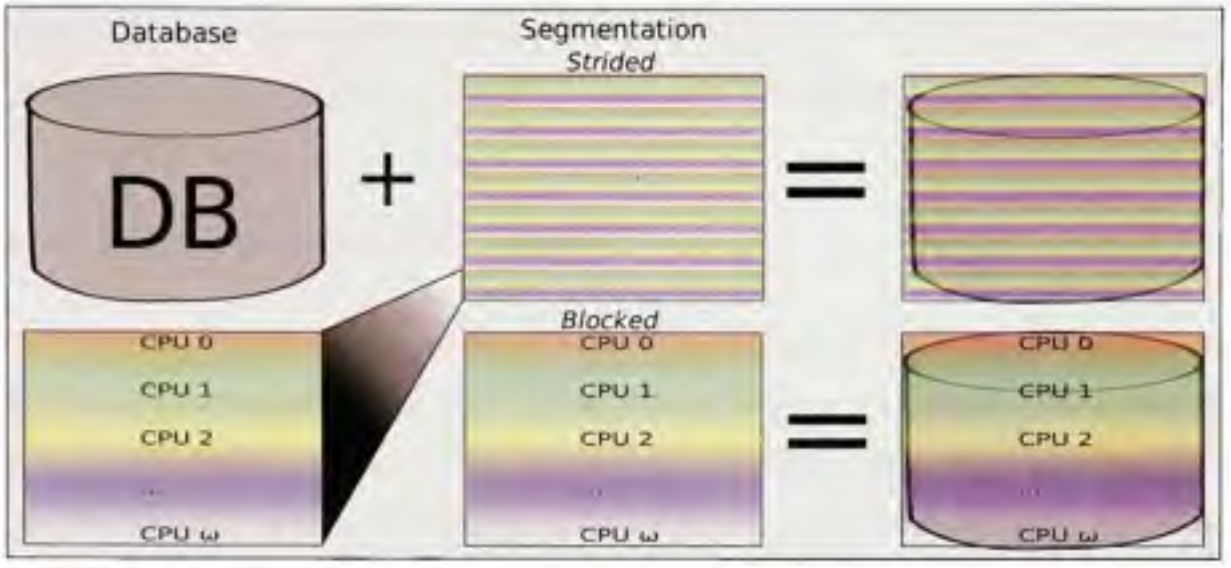


Figure 3.3 : Database segmentation strategies: TOP- *Strided* segmentation (fine grained) is used by the master-slave algorithm where each element of the database is assigned to one ω worker node in a round-robin fashion. BOTTOM- *Block* segmentation approach (coarse grained), assigns equal *consecutive* chunks of the database to each worker as per $|DB|/\omega$ with the remainder assigned to the last worker.

statically (at processing start up) or dynamically, through a scheduling scheme which assigns computational tasks as the processing evolves. The former typically requires little communications whereas the latter usually implies a continuous stream of communications from a *task manager* towards the nodes. This *queue* based approach has been explored for the k-means algorithm by [58] with less than optimal results. We note that, in that specific case, $|DB|$ and d were significantly small compared to our typical use cases [4].

3.2.1.4 Hardware Considerations: Physical Limitations

Another issue that can come about are the actual hardware limitations for each processing node. For example, the available RAM a node has can dictate the maximum size of each segment. Such considerations will also gain much relevance with the growing use of Graphics Processing Units (GPUs). Such technology impose stricter segmentation guidelines as the processing units share limited amounts of video memory [7]. This in effect limits the amount

4. Their largest performance test cases had at most $|DB| = 100k$ samples with $d = 2$ whereas we have $|DB| = 6,5M$ and $d = 47$.

of data sets that can be loaded at a given time as well as require a different program structure to be used. The gain in performance remains important in the realm of classification algorithms as demonstrated in [16][5].

3.2.2 Then Tell Everyone: Communications

Let's recall that the core element of the k-means algorithm is to compute the Euclidean norm between a given element and its representative centroid. We loosely represent this computation as t_{comp}. The end of each computation cycle N_{iter} is punctuated by a communication stage, identified as t_{comm}, where all partial results are amalgamated[6]. These two components lead to the general parallel tim ($t_{||}$) equation presented in Eq. (3.3). This is the prized crude representation [14] of parallel processing time, which, as we will demonstrate, can easily lead to distorted expectations. For example, other *times* such as initialization (t_{init}) and loading (t_{load}) times will come into play as important contributors to be dealt with.

$$t_{||} = N_{iter} \cdot (t_{comp} + t_{comm}) \tag{3.3}$$

It was demonstrated implicitly by [57] and then explicitly by [43] that the number of nodes ω, used in the parallelization of the computation process, is theoretically limited by the inter-iteration communications. Since the modelization of the communications is dependant upon the logical and topological distribution, we will present them in more details in their respective sections.

3.3 And Conquer: Master-Slave Model

Although there are many ways one can implement the master-slave k-means algorithm [13, 43], their topology can be generalized by Figure 3.4 . The approach we present here is from [43], where the master is responsible for computing the intermediate steps of the algorithm such as the centroid update and total distortion as well as propagating the new centroids. We

5. Again, we must be weary of the dataset size as their experiment's dimensionality is not comparable to ours.
6. We consider the consolidation as part of t_{comp} given its computational insignificance.

depict this in Algorithm 5 where the ωorkers (top half) are numbered such that $\omega = [0, p-1]$. The segmentation strategy is described by line 3 of Algorithm 5 using a ternary opertator where the local database (DB_ω) is assigned the element DB_j if the remainder of applying the modulo operator to j with the node number ω matches the node number itself, otherwise, the element is skipped[7].

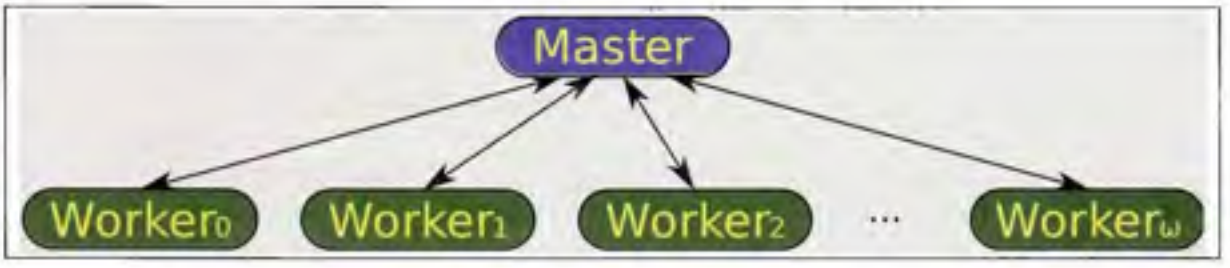


Figure 3.4 : A typical master-slave topology. All communications originate and terminate on the master. The nodes do not communicate between each other.

3.3.1 Master-Slave Communications

We identify communications using **bold and underline** in Algorithm 5. In this version of the algorithm, the k centroids are initialized by having each node select its first k/ω elements and send these to all other nodes. This process is performed using a modulo operator, applied against the node's identity ω. with each element being sent as they are selected. This requires $k \times (\omega - 1)^2$ PtP communications.

After an iteration, each node sends its local centroid table C'_ω, element count $\mathbf{m}_\omega$ and distortion $\mathbf{dist}_\omega$ vectors, to the *master*. This, in turn, represents 3ω PtP communications, each having a respective payload of kd for C'_ω, k for $\mathbf{m}_\omega$ and $\mathbf{dist}_\omega$ elements[8]. The *master* then computes the new centroids table C and global distortion $dist$ and sends them back to the workers for the next iteration.

7. Note that some *off by one* adjustments were not included for clarity. The actual implementation is available in Appendix I.

8. Where each element is the size of a `float` (4 bytes).

1: **if** $pid > 0$ {This is a slave process} **then**
2: Initialization of C by selecting the first k/p elements from each pid in a round robin fashion and **sending each locally selected element to all other workers**
3: Initialization of local database: $DB_{\omega_j} \leftarrow (j \mod \omega) = \omega \ ? \ DB_j \ : < skip >$
4: **repeat**
5: Clear out intermediate values $C' \leftarrow 0$ and $m \leftarrow 0$
6: **for all** $\mathbf{x_i}$ in DB_ω **do**
7: Identify closest centroid as per $\arg\min_k (\|\mathbf{x_i} - \mathbf{c_k}\|)$
8: Save the element's distance from the centroid $dist_\omega = \|\mathbf{x_i} - \mathbf{c_k}\|$
9: Add $\mathbf{x_i}$ to intermediate centroid: $\mathbf{c}_\omega^{\mathbf{k'}} := \mathbf{c}_\omega^{\mathbf{k'}} + \mathbf{x_i}$
10: Increment the centroid's element counter $m_\omega^k := m_\omega^k + 1$
11: **end for**
12: **Send partial results**, m_ω, C'_ω and $dist_\omega$, to master process and wait for new C and total $dist$.
13: **until** $|dist| < \delta$

(ω Workers)

(Master)

14: **else if** $pid = 0$ {This is the master process} **then**
15: **repeat**
16: **Wait for partial results**, m_ω, C'_ω and $dist_\omega$, from slave processes.
17: Combine partial results such that $m^k = \sum_{j=1}^{\omega} m_i$ and $C' = \sum_{j=1}^{\omega} C'_i$
18: **for all** c'_j where $1 \leq j \leq k$ **do**
19: Compute the new mean vector $\mathbf{c'_j} := \mathbf{c'_j}/m^j$
20: **end for**
21: Assign new centroids as current $C = C'$
22: Compute total distortion $dist = \frac{1}{|DB|} \sum_{j=1}^{\omega} dist_j$
23: **Send new C and $dist$ to slaves**.
24: **until** $|dist| < \delta$
25: **end if**

Algorithm 5: The Master-Slave Parallel K-Means

Recalling the general communications model in Eq. (1.4) (from section 1.3.5), the *master-slave* communications overhead is modeled by two phases of communications. The *worker* to *master* communications are represented by Eq. (3.4), which is composed of three distinct PtP communications (hence $3t_s$) and a total payload of $4k(d+2)$. The *master* to *worker* communications is described by Eq. (3.5) which is also composed of three distinct PtP communications with a

payload of $4(k(d+1)+2)$.

$$T_{collect}^{worker} = 3t_s + t_{byte} \cdot 4k(d+2) \tag{3.4}$$

$$T_{update}^{master} = 3t_s + t_{byte} \cdot 4(k(d+1)+2) \tag{3.5}$$

We have adapted these equations to reflect the actual source code implementation of the communication primitives. These are comprised of matching pairs of `MPI_send/MPI_recv` function pairs as illustrated by Figure 3.5 . Variable names are chosen to concur with Algorithm 5.

Workers send partial results

```
//Send parital results to Master
MPI_Send(&C   , (K*D), MPI_FLOAT, master, tag+1, MPI_COMM_WORLD);
MPI_Send(&distc, K  , MPI_FLOAT, master, tag+3, MPI_COMM_WORLD);
MPI_Send(&m_w , K  , MPI_FLOAT, master, tag+2, MPI_COMM_WORLD);

//get results from Workers
for(j = 1; j < w; j++)
{
        MPI_Recv(&C   , (K*D), MPI_FLOAT, j, tag+1, MPI_COMM_WORLD, &status);
        MPI_Recv(&distc, K  , MPI_FLOAT, j, tag+3, MPI_COMM_WORLD, &status);
        MPI_Recv(&m_w , K  , MPI_FLOAT, j, tag+2, MPI_COMM_WORLD, &status);
...
}
```

Figure 3.5 : The workers send their partial results to the master

Master updates workers

```
//Master
for(i = 1; i < totalnodes; i++)
{
        MPI_Send(&C    , (K*(D+1)), MPI_FLOAT, i, tag , MPI_COMM_WORLD);
        MPI_Send(&dist_ant, 1    , MPI_FLOAT, i, tag+5, MPI_COMM_WORLD);
        MPI_Send(&dist  , 1    , MPI_FLOAT, i, tag+6, MPI_COMM_WORLD);
}

//receive from the Master
MPI_Recv(&C    , (K*(D+1)), MPI_FLOAT, master, tag , MPI_COMM_WORLD, &status);
MPI_Recv(&dist_ant, 1    , MPI_FLOAT, master, tag+5, MPI_COMM_WORLD, &status);
MPI_Recv(&dist  , 1    , MPI_FLOAT, master, tag+6, MPI_COMM_WORLD, &status);
```

Figure 3.6 : The master updates the workers with the new values

The Message Sequence Chart in Figure 3.7 illustrates an *idealized*[9] communications between the master and workers. This emphasizes the fact that all communications originate and terminate on the master.

3.3.2 Master-Slave Empirical Modelization

As it was demonstrated in Chapter 1, performance characteristics vary significantly even within the same class of hardware. The simple communications model, described in section 1.3.5 from that same chapter, will be used as a basis to define the communication times of this master-slave implementation. The model is completed using TAU [46] to extract the computation times of important parts of the application. The Beowulf cluster used for our tests is described in detail in Appendix III.

A gross estimate of the parallel computation time $t_{comp\|}$, for one iteration t, is presented in Eq. (3.6) where $t_{f(\|\cdot\|)}$ is the time required for a single Euclidean norm computation, k is the number of centroids, n is the number of samples in the database and ω is the number of workers. We chose $t_{f(\|\cdot\|)}$ for the computation times as it is the smallest token of computation in the k-means algorithm but also the most called upon.

$$t_{comp\|} = t_{f(\|\cdot\|)} k \frac{|DB|}{\omega} \tag{3.6}$$

We have measured that $t_{f(\|\cdot\|)} \approx 1.8\mu s$. With $|DB| = 342910$ (the size of the entire test database), $k = 10$ and $\omega = 10$, we get a modeled $t_{comp\|} = 0.6172$ seconds per iteration. Eq. (3.4) and Eq. (3.5) are then used to estimate t_{comm}. Given the vector size of $d = 47$, we get

$$\begin{aligned} t_{comm} &= \omega \left(T_{collect}^{worker} + T_{update}^{master} \right) \\ &= 10 \left(6t_s + t_{byte} \cdot (4(k(d+1)+2) + 4(k(d+1)+2)) \right) \\ &= 6.4576ms \end{aligned} \tag{3.7}$$

9. Meaning that we neglect the possibility of *out of order* communications and collisions.

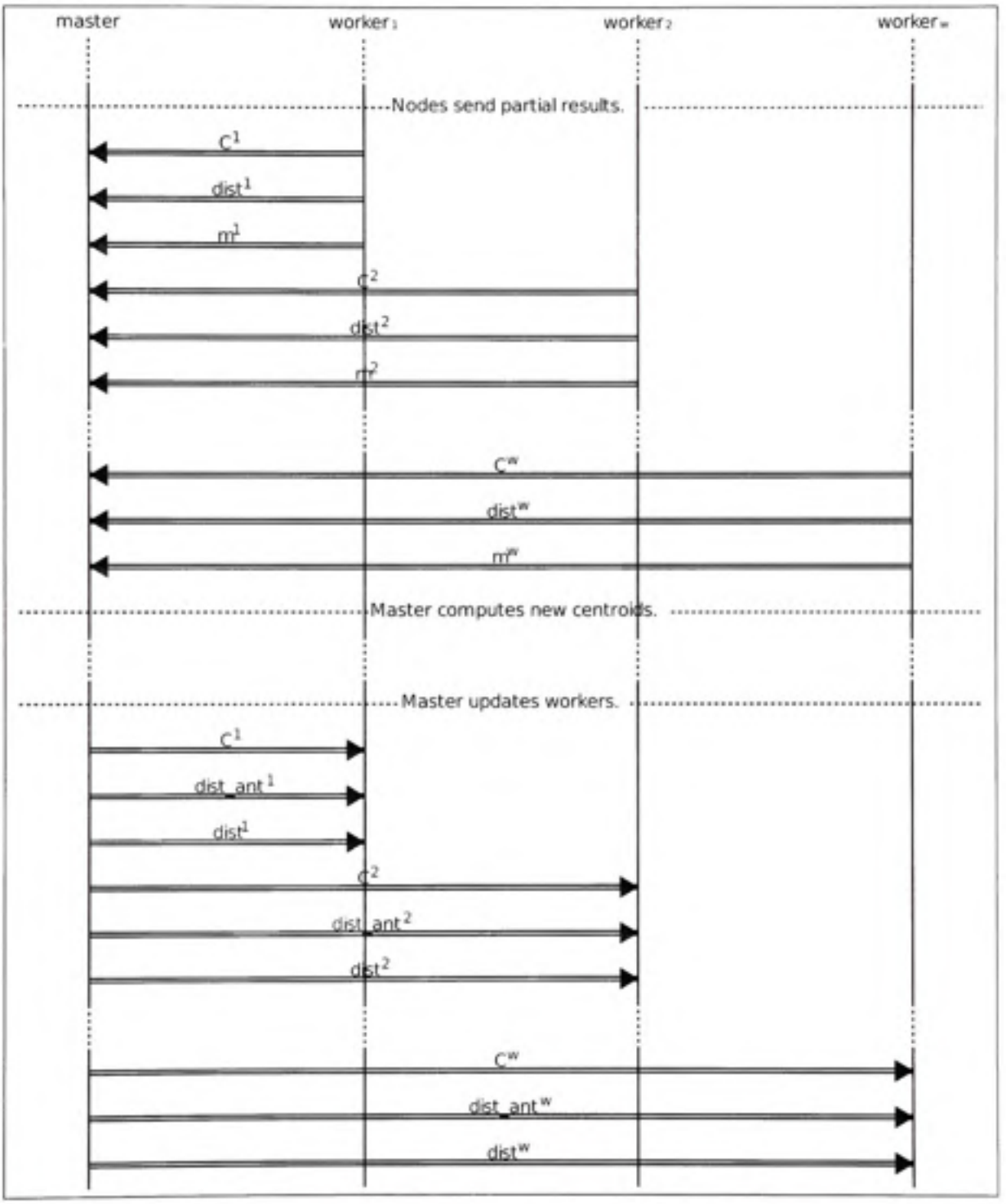


Figure 3.7 : Master-Slave Message Sequence Chart (MSC) for the inter-iteration communications. All communications are point to point and must be performed by all nodes.

The first thing one notices is that the model anticipates that the communications will be negligible [10] compared to the actual computation. Knowing that a trial run with the same parameters

10. Close to 100 times less.

took six iterations to converge, the general equation presented in Eq. (3.3) renders a total expected execution time of 3.74 seconds. Unfortunately, this figure proves to be overly optimistic as the actual measured execution time averages [11] around 9.23 seconds. The execution time is therefore more than threefolds the estimated value using the theoretical model based on empirical data.

To investigate this large discrepancy, we profile the entire application using TAU. Our observations start with the 3D view from Figure 3.8 where we have isolated the two functions of interest, the communications (the first row identified as `MPI_Recv()`), and the computation (the second row which is identified as `float df()`). The height of the bars represent the time spent in each function, the color represents the time per call of each function. As expected, the master node (node zero) spends most of its time in the `MPI_Recv` communication call as it waits for the workers to complete their part of the computation. What is also revealed is the heterogeneity of the cluster [12] with nodes being more powerful than others. This is put forth by the varying heights and lighter coloring [13] of the communication bars for the faster computers. The 3D view makes it easy and intuitive to correlate the computation time and communication times seen by each worker node. It is obvious that faster computers end up waiting longer in the communication calls by simply looking at the bar height. These observations alone are sufficient to invalidate our assumptions about the communications model, the most notable one being that one must account for delays imposed by *all* other nodes for a communication to be considered completed in a master-slave topology using PtP communications. In other words, the present communication model is designed in a way that each PtP must complete in the correct order, and all delays imposed by slower nodes must be added. Note that this is not a limitation of the communication library but rather a design flaw in the code's use of the library by forcing a given sequence in the communications.

11. We ran 30 times the experiment within the same timeframe varying the node count from 2 to 24.
12. Described in more details in Chapter III, section 1.
13. Had the color been the same for all bars of the same row, this would have indicated that all calls took the same time for that row, therefore implying that higher bars are indicative or *more* calls, not *longer* calls.

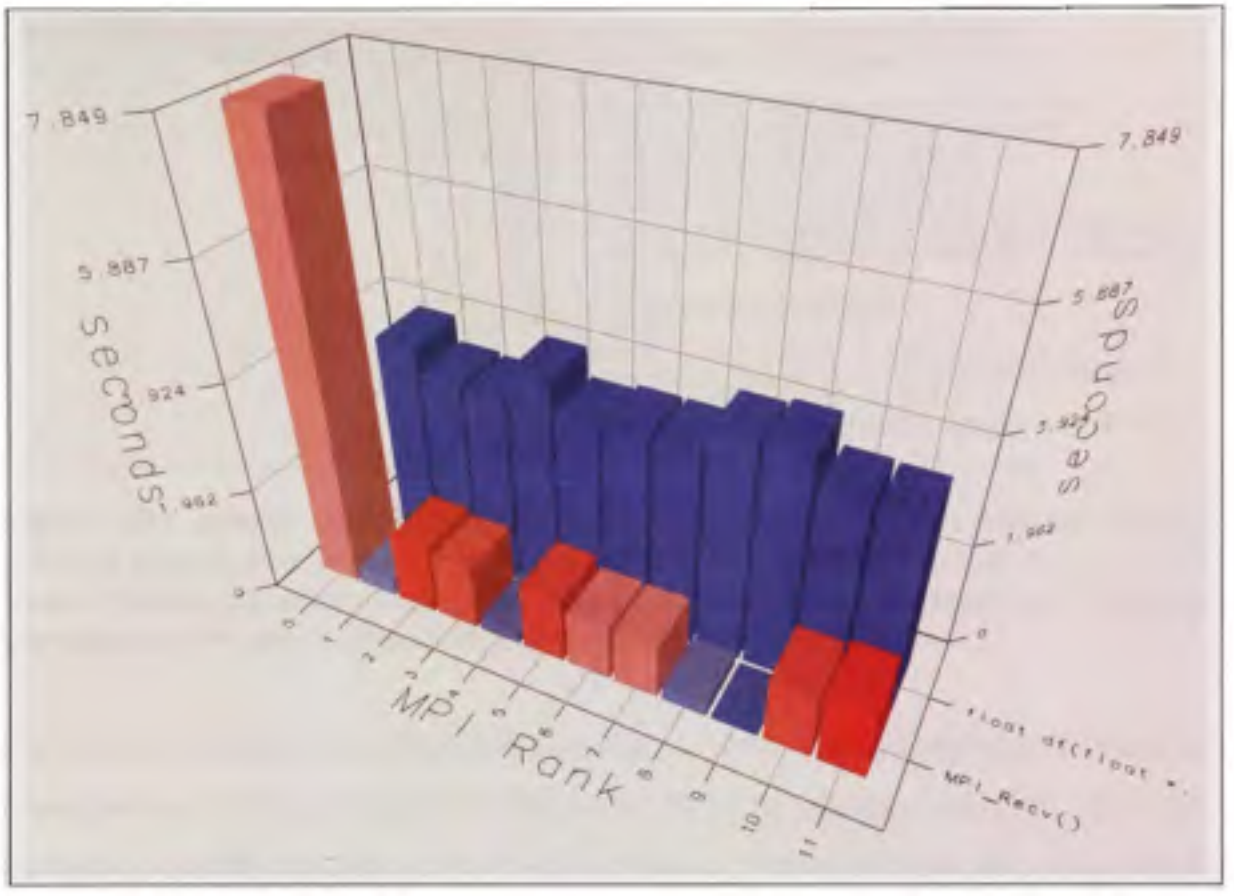


Figure 3.8 : The 3D view of the master-slave communications `MPI_Recv()` and computation cycles `df()` for all nodes. The master node (node 0) spends most of its time waiting for the results from the worker nodes. Columns are colored according to time per call for the function.

We now observe the tasks accomplished by a single worker. Figure 3.9 is a barchart presenting each function's cumulative contribution to the total execution time for a single worker. As expected, `df()` takes most of the time with 3.2 seconds. Once again, we notice the excessive time (compared to the anticipated model) for `MPI_*` function calls.

Furthermore, what the model fails to address is the fact that the loading of the samples database, the function called `load_samples()`, would come in second place with 2.47 seconds. This last observation is one of the often neglected considerations in parallel performance models, a point brought up by Foster in [14] when discussing execution profiles.

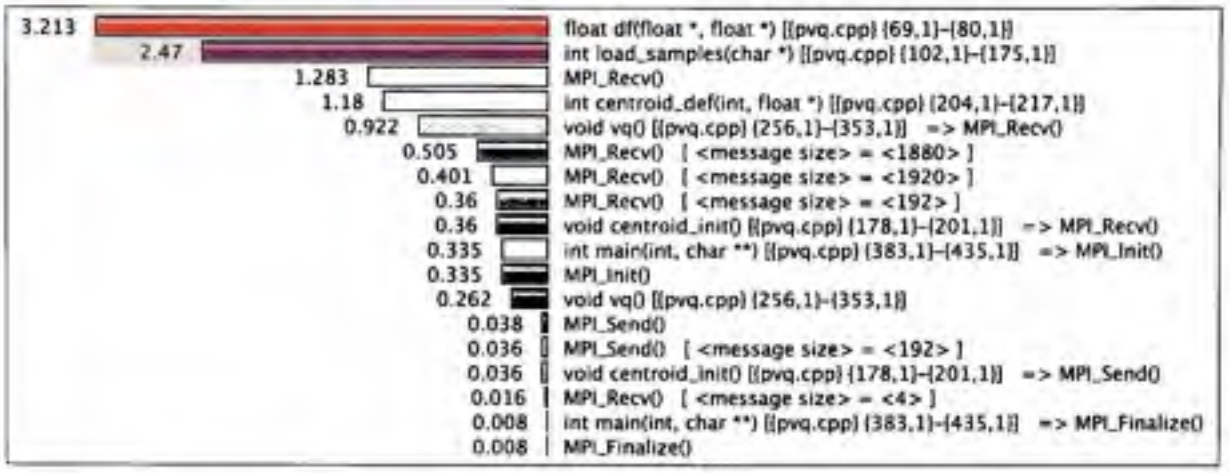


Figure 3.9 : Average time spent by all nodes in each function. Each calls are sorted by order of contribution importance. Calls under 0.008 seconds aren't shown for clarity. Braces indicate the source file and line numbers, bracket information specify which call parameters were used and function call paths are indicated using '=>'.

The runtime correlation analysis graphic from Figure 3.10 is also used to correlate a function's execution time with the addition of worker nodes. Of all the functions, only `MPI_Init()` has a negative correlation coefficient ($r = -0.50$), meaning that its execution time is unrelated to the addition of worker nodes and might also become a bottleneck.

Finally, we performed a scalability analysis by varying the number of workers between 2 and 24. The resulting runtime breakdown graphic in Figure 3.11 shows that the `load_samples()` function is hampering scalability as its importance grows with the addition of computing nodes. An important note about `MPI_Recv()` is that it *seems* to scale quite well but, in fact, the presented proportion is biased by the fact that the baseline reference of two nodes actually has a single node computing and the master essentially spends close to 100% of its time in the `MPI_Recv()` call. Hence the 50% proportion allotted to this call when only two nodes are considered. We also confirm our observation from Figure 3.10 , where the negative correlation predicted that the `MPI_Init()`'s function time contribution would grow and therefore become a potential bottleneck as nodes are added.

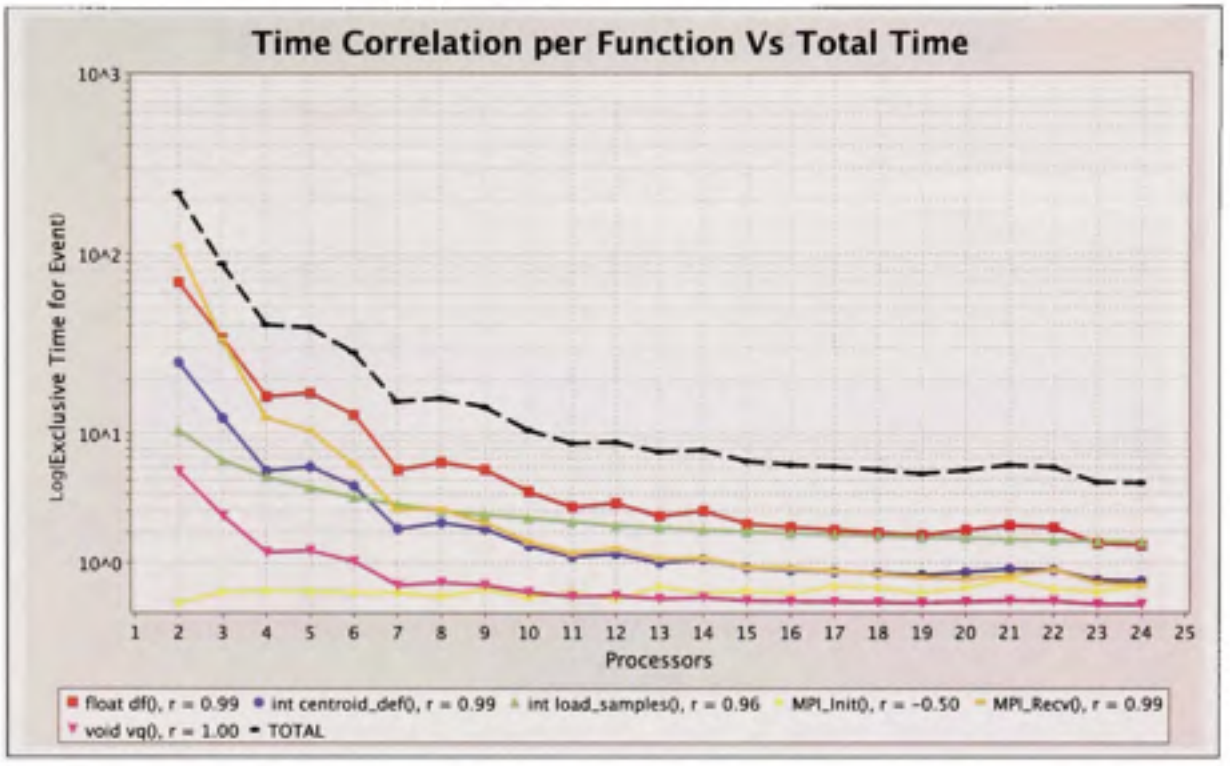


Figure 3.10 : Correlation analysis for $\omega = [2, 24]$. Each function's time contribution is drawn as the worker count grows. The correlation coefficient r, indicates the correlation between the addition of nodes and the execution time of the function.

With such information at hand, we move onto the following section in which we present an optimized version of the parallel k-means where we have implemented the *island* parallel computation paradigm.

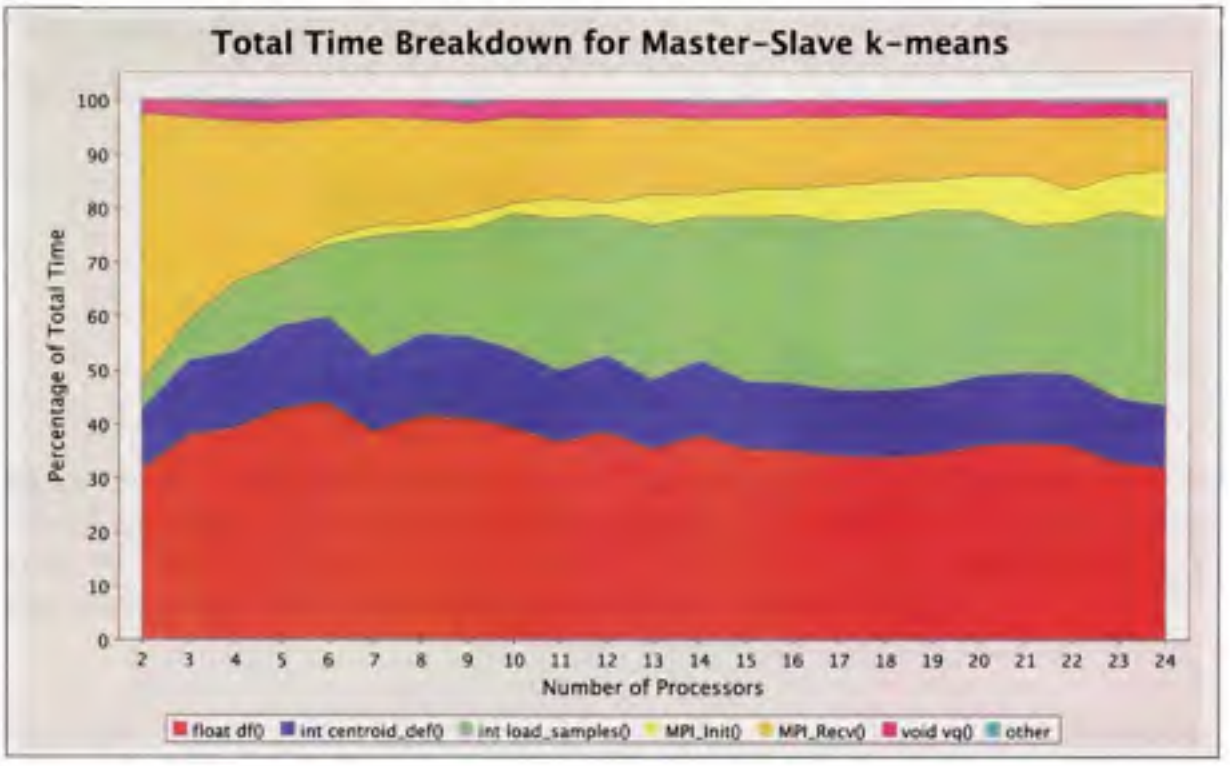


Figure 3.11 : Runtime breakdown for $\omega = [2, 24]$. Each function's proportional importance for the total execution time is depicted by its surface coverage as nodes are added to the computation. A perfectly scalable function would be represented by a constant surface area whereas a growing surface is indicative of poor scaling.

3.4 Or, Invade: Synchronous Island Model

In the master-slave model, all communications are performed from and towards a master node. Also, this node typically doesn't participate in the computational task other than communicating and computing intermediate parameters. Computation cycles are therefore lost while the master awaits the results from the nodes and, vice versa, the nodes are idle while waiting for the update from the master node. Also, given the PtP implementation, computation on the last node can only start after all other nodes have received their updates, which we have modeled as $(w-1) \cdot T_{update}^{master}$. As another well-known topological parallel paradigm is the synchronous island model which can be generalized by Figure 3.12 . In this model, all nodes participate to the computation and the communication paths interconnect all nodes. This implies the use of

a fully connected (or flat) network such as is the case in most Beowulf implementations using Ethernet networking fabric.

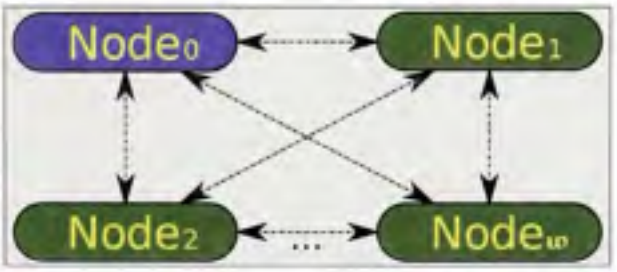


Figure 3.12 : A typical island topology. Communications originate and terminate between each node. This model implies a fully connected network where all nodes can see eachother (typical Ethernet configuration). The number of actual communications varies depending on the MPI implementation of the global communicators.

We use this model to address inefficiencies found in the master-slave model. Our implementation of the synchronous island parallel k-means is described in Algorithm 6. Again, the communications are in **bold and underlined** and, as with the master-slave model, are also the point at which all nodes are synchronized at the end of each iteration.

The most notable change between the master-slave and island model is the lack of distinction between a said master node and workers. Note that we have also implemented a simplified centroid initialization scheme where all nodes use a predefined pattern to initialize C using elements from X. This modification eliminates the need to communicate between nodes for this initial step and also ensures that the result is not dependant upon the number of nodes used [14].

14. The original master-slave implementation would use a modulo operator combined with the node count to select elements from DB, this would lead to variances in the end result and in the execution time.

1: Local initialization of C with elements from DB using a pattern known by all workers
2: Each local database DB_ω is assigned a chunk of size $|DB|/\omega$
3: **repeat**
4: Clear out intermediate values $C' \leftarrow 0$ and $m \leftarrow 0$
5: **for all** $\mathbf{x_i}$ in DB_ω **do**
6: Identify closest centroid as per $\arg\min_k (\|\mathbf{x_i} - \mathbf{c_k}\|)$
7: Save the element's distance from the centroid $dist_\omega = \|\mathbf{x_i} - \mathbf{c_k}\|$
8: Add $\mathbf{x_i}$ to intermediate centroid: $\mathbf{c}_\omega^{\mathbf{k}'} := \mathbf{c}_\omega^{\mathbf{k}'} + \mathbf{x_i}$
9: Increment the centroid's element counter $m_\omega^k := m_\omega^k + 1$
10: **end for**
11: **Exchange and combine partial results**:
Local values of m_ω, C'_ω and $dist_\omega$, are exchanged with all ωorkers
While they are being exchanged, combine partial results such that:
$m^k = \sum_{j=1}^{\omega} m_i$ and $C' = \sum_{j=1}^{\omega} C'_i$
12: **for all** c'_j where $1 \leq j \leq k$ **do**
13: Compute the new mean vector $\mathbf{c'_j} := \mathbf{c'_j}/m^j$
14: **end for**
15: Assign new centroids as current $C = C'$
16: Compute total distortion $dist = \frac{1}{|DB|} \sum_{j=1}^{\omega} dist_j$
17: **until** $|dist| < \delta$

Algorithm 6: The Island Parallel K-Means. All nodes execute this exact same algorithm.

3.4.1 Optimizing the code

As a general rule, code optimization requires that a baseline be established as a point of comparison to assess the enhancement or degradation of performance. Since we have performed a complete profiling of the application, we have access to per-function execution times with a given set of parameters such as node count, centroid count and loaded elements from the sample database. We can therefore work on individual functions and compare the optimized versions to the original ones.

We have established that the following functions [15], in order of importance as per Figure 3.9 , either require optimization or represent a significant enough portion of the execution time to warrant further investigation:

- `df()`: The Euclidean norm computation function;
- `load_samples()`: The database loading function;
- `MPI_Recv() [*]`: The MPI calls (we consider the sum of all of them);
- `centroid_def()`: The function that defines which elements of DB are clostest to the k centroid.

The three categories of optimizations are therefore to be considered, computation, I/O, and communications. In this section, we present the MPI communication primitives as they are the most abstracted from the hardware architecture, are tightly bound to the chosen topology (island), and will represent the most code architecture change. The I/O (`load_samples()`) and computation routines (`centroid_def()` and `df()`) are both applicable on a ordered basis and apply to any model. They will therefore be presented in their own section.

15. You may peer into their original implementation in Appendix I and their final implementation in Appendix II.

3.4.2 Island Communications

As we have just demonstrated, the island model is algorithmically simpler than the master-slave approach. The master's role in the previous implementation served the only function of collecting three partial results then computing and redistributing the new centroids and the current global distortion.

3.4.2.1 Overlapping Communications and Computation

In the island model, these three steps are accomplished by the three MPI collective calls as presented in algorithm Figure 3.13 . Not only do their semantic adhere more closely to Algorithm 6, but they also replace 15 blocking PtP calls [16] that were made in the original master-slave implementation. This approach also has the added benefit that it provides the necessary leeway for MPI-level improvements in the implementation of the global communicator [56, 4]. Another key benefit is that it also overlaps communications with computation [17] as well as simplifies the implementation which in effect reduces the probability of introducing deadlock conditions and simplifies debugging.

Island Communications

```
MPI_Allreduce(MPI_IN_PLACE, &distc, 1 , MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
MPI_Allreduce(MPI_IN_PLACE, c_int , K , MPI_INT , MPI_SUM, MPI_COMM_WORLD);
MPI_Allreduce(MPI_IN_PLACE, c_sum , K*T, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
```

Figure 3.13 : The three collective calls used to communicate and perform an element by element summation of all three intermediate variables.

The Message Sequence Chart (MSC) in Figure 3.14 illustrates the communications between all workers. The three collective calls are clearly separated by the horizontal dotted lines, which in effect indicate a communication *barrier* where all nodes must have completed their call to the communicator before moving onto the next call. These barriers will prove to be a limitation to the algorithm which we will address shortly.

16. Including the calls made during the initialization, otherwise the figure is 11 calls for the main computation loop.

17. The efficiency of the overlapping is dependant upon the MPI library implementation.

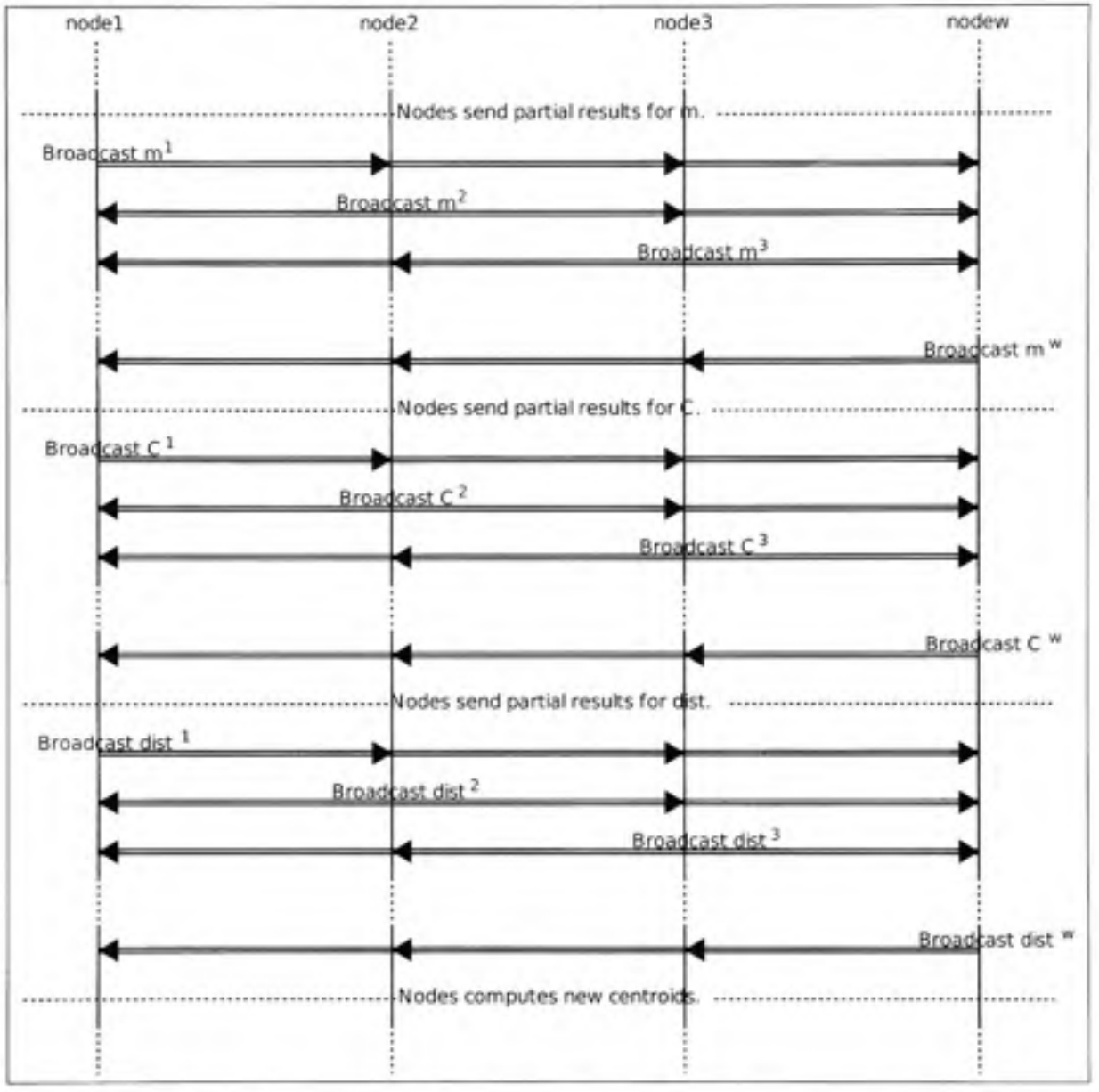


Figure 3.14 : Island MSC for the inter-iteration communications. Although drawn as sequential, collective communications can overlap within the same call to `MPI_Allreduce` but must complete within the same call (equivalent to a communication barrier). These barriers are depicted by the horizontal dotted lines. They must also be performed by all nodes.

But first, we observe a few results comparing the two communication approaches. In Figure 3.15 (a), the total average communication time sums up to about 4.45 for the master-slave algorithm. In Figure 3.15 (b), our island model presents a significant improvement with a given average of 0.51 seconds for its *unique* communication.

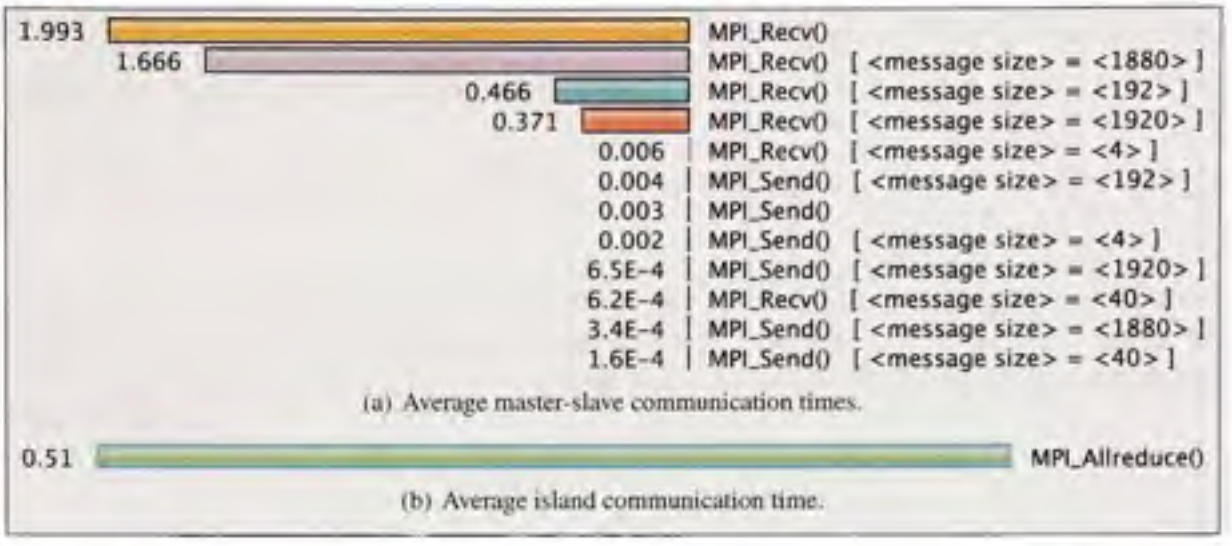


Figure 3.15 : Average communication times for both approaches. Master-slave communications are presented in (a) while the only communication for the island model is in (b).

3.4.2.2 Less Talk, More Work

As we have just mentioned, although the use of three separate collective communication calls is semantically identical to the algorithm, the introduced synchronization barriers add communication latency and prevent the overlap of computation for the successive collective calls. This effectively eliminates some of the advantages of the collective communicators.

There are two ways to address this. The first one is to create a custom MPI data type (structure), which consolidates the three elements into a single communication block. The creation of custom data types in itself isn't too problematic but their use with collective communicators that operate on the data adds the complexity of also having to create custom MPI operators. We opted to use a simple alternative which consists in using a single large vector to contain all three elements. This approach requires less code modification and proved to be much simpler to implement. It is also possible because the operation to be performed on all exchanged elements is the same (a summation) and that the datatypes are compatible. The only variable assignment that required some modification is for the centroid ownership counter m which was changed

from `int` to `float`. The three calls from algorithm Figure 3.13 are therefore merged into a single call as presented in algorithm Figure 3.16

```
Merged Island Communications
MPI_Allreduce(MPI_IN_PLACE, c_sum , (K*T + K + 1), MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
```

Figure 3.16 : A single collective call performs the exchange and summation of all intermediate values. The variable `c_sum` is supersized to include C, m and $dist$, hence the communication size of $K * T + K + 1$. Each variable simply points to its specific region within `c_sum`.

The Message Sequence Chart (MSC) for the communications therefore becomes much simpler as attested by Figure 3.17 where all communications are consolidated into a single call from each node. This approach has the potential [18] of generating as little as ω communications compared to the PtP approach with its $6 \cdot (\omega - 1)$ communications [19].

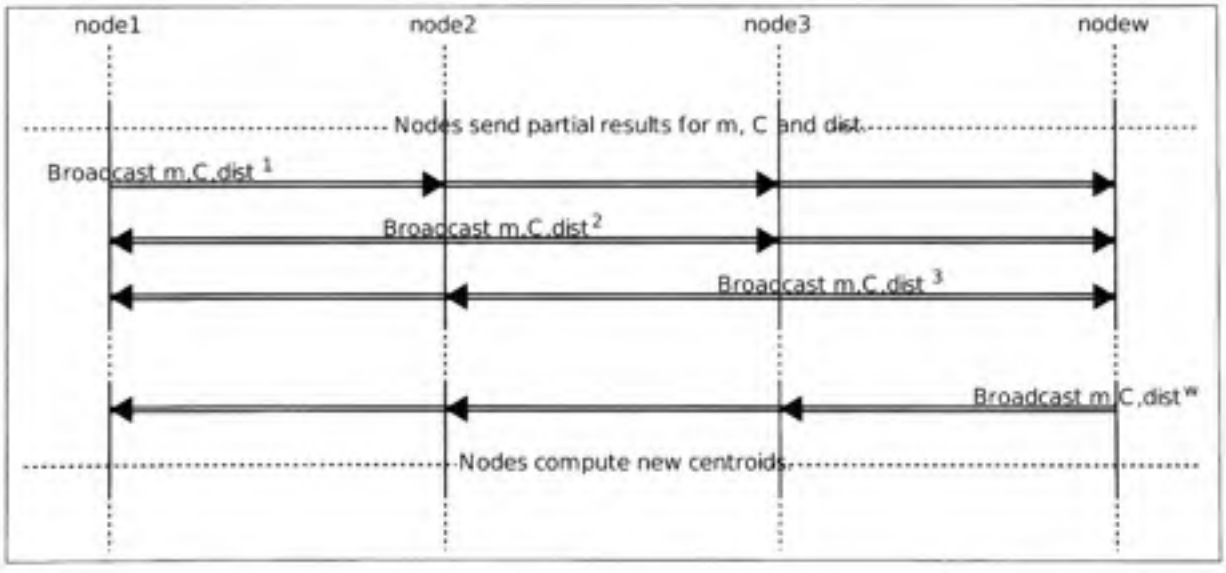


Figure 3.17 : Simplified Island MSC for the inter-iteration communications. A single collective call from each node communicates all intermediate values and performs their sum at the same time.

18. The MPI standard does not enforce that collective communicators be implemented efficiently. They can actually be a wrapped version of PtP communications

19. Recall that there are 3 send-receive pairs for each node in the master-slave model.

3.5 Optimization of I/O Routines

We have established that the `load_samples()` I/O function is hampering most of the scalability according to Figure 3.11 . Investigations into the `load_samples()` I/O routine reveals that the database is in fact an ASCII (text) file containing 47 columns of numeral data separated by spaces for each dimension d and each element on its own line.

Storing data in textual format, although human readable, represents a heavy burden as far as raw space and computation requirements are concerned.

For example, each element of a vector is represented by a character string (ie: 0.032352) to which we must add a space or the end of line character. This representation takes a total of 9 bytes for single number where its binary equivalent in `float` format only takes 4 bytes. Not withstanding a gain in precision, storing the data in binary format would therefore reduce the raw data transfer requirements down to 44% of the original figures. Furthermore, the textual representation of numbers have to be converted to `float` format, which implies that each and every byte of the file has to pass through the processor. This represents a considerable amount of processing which, in its binary format, isn't required.

Finally, performance enhancing mechanisms such as Direct Memory Access (DMA), allowing the direct transfer of data from disk to memory, as well as OS based file caching are impossible with the use of textual data. Even if the data is cached in main memory due to recent access, it will still need to be re-parsed by the processor the next time the program is called [20].

The above-mentioned reasons and the poor performance revealed by our performance profiling has lead our implementation to use a binary file format. The performance gain is more than significant, trial runs executed on 12 nodes using both approaches revealed that the text database took an average of 2.085 seconds to load whereas the binary version took 0.018 seconds to load [21]. This represents a considerable speedup, the binary version being over 115 times faster

20. Recall that the k-means of our case study is part of a Genetic Algorithm (GA) in which the k-means serves as a fitness evaluator, thus being called multiple times upon the same data.

21. The binary database was *in cache* as the previous system call forced a read of the entire file (a call to `md5sum`).

than the original code. Such speed gain is attributable to the fact that there is no longer a need to convert from the ASCII format, less data needs to be read from disk, the data can be loaded directly into RAM without passing through the CPU and the use of file pointer arithmetic is now possible, eliminating the need to read the entire database to load the node's portion into memory (we can jump to the right entry immediately). This enhances the program's scalability by reducing the read time proportionally to the number of workers (read time should be inversely propositional to the number of nodes).

3.6 Computational Optimizations: Coding for High Performance Computing (HPC)

Although the ultimate goal of most programming language is to provide an abstraction layer between hardware and software components, some considerations are to be taken into account when dealing with HPC. Such programming constraints are seldom applied unless there is a proven performance gain in the overall application, which implies that *hotspots* have been identified and that proposed techniques are known to have a significant impact.

In both parallel models, the `df()` and `centroid_def()` functions have prooven to be *hotspots*. They both possess a high call and cumulative time count (Figure 3.2 (d) and Figure 3.2 (c)). But what we have also noted is that these function calls are very short (Figure 3.2 (e)). Code optimization techniques are much more complex and require intrinsic knowledge of the underlying hardware to guide the applied techniques. We will use TAU and PAPI more extensively in this section to investigate the probable paths to optimizing the code. When possible, the compiler's implementation of the technique will be used when a performance gain is obtained. We will only revert to manual modifications of the code when absolutely necessary. This way, the code remains as close to the original implementation and doesn't get overfitted to a given hardware platform.

3.6.1 Compiler Directives

Compilers are the core component of any software development project, it is therefore essential to be aware of their capabilities and options as well as the impact using optimization flags. We study the impact of these in Appendix I and refer to the obtained results throughout this chapter.

3.6.2 Mathematical Libraries Versus Code

We have just demonstrated that most of the program's execution time is made up of small mathematical kernels called up repeatedly. Most basic mathematical functions, such as `pow()`, `sin()`, `cos()` are implemented via standard mathematical libraries. It is often debated whether or not these should be used when performance is concerned.

The `glibc` mathematical library has an Institute of Electrical and Electronics Engineers (IEEE) standard compliant implementation of basic trigonometrical, logarithmic, power and many other operations. Although our Euclidean computation kernel is quite simple, we notice that the `df()` function from Figure I.2 might be implemented with the `pow()` function [22] to compute the squared distance. We investigated the relevance of such a substitution of the explicit code with its equivalent call to `pow()`. As we can see in Figure 3.18 , the use of `pow()` renders code that is slower and less efficient than its hand-coded equivalent. In all cases, whether it be time, processor cycles, processor instructions, floating point instructions and vectorized floaring point instructions, the hand coded implementation is *always* faster, uses less processor cycles and instructions.

We attribute the performance loss to the fact that the library approach adds a function call and that the current GCC implementation does not yet perform propagation of optimizations such as defined by `-ffast-math` [23]. Note that optimization propagation such as ignoring error and boundary conditions down to the compiled library is defined in the C99 standard [24].

22. Note that the library documentation stipulates that any of the *functions* may in fact be defined as macros.

23. Referring to *"treatment of error conditions by math library functions (math_errhandling)"* at http://gcc.gnu.org/c99status.html for all 4.*x* versions of the GCC

24. As per `http://www.open-std.org/jtc1/sc22/wg14/www/standards`, *The lastest publically available version of the standard is the combined C99 + TC1 + TC2, WG14 N1124, dated 2005-05-06.*

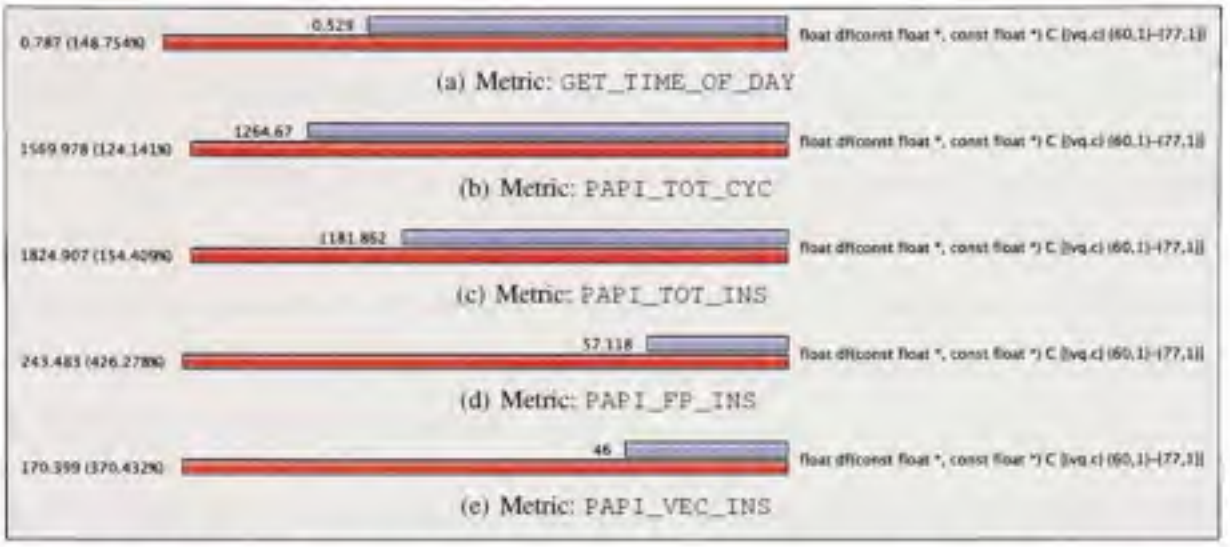


Figure 3.18 : Comparing hand coded squared function ($a \times a$) to the use of `pow()` on Intel $Q6600$. The metric used in all cases is the exclusive mean per-call values of the fucntion. In all figures ((a) to (e)), the top bar (in blue) uses the explicit definition while the red bar below uses the library call to `pow(a, 2)`. All the presented metrics point to the expanded version as being more efficient by consuming less total time (a), cycles (b), issuing less instructions (c) (total) and even less floating point (d) and vector instructions (e).

The performance *gain*, on the other hand, can be explained by the fact that the compiler was able to recognize the intended operation and generated code that would explicitly use hardware specific features such as SIMD instructions, some of the key features of the Intel $Q6600$. We detail their use and implication, coupled with loop optimizations, in the following sections.

3.6.3 Using Single Instruction Multiple Data

As we have discussed in Chapter 1, most contemporary processors have stagnated as far as clock speed is concerned. Other strategies such as ILP and data parallel operations are now being implemented to compensate for the lack of performance enhancements. This in effect is indicative of the rebirth of vector processing, mostly by adding SIMD instruction sets (mnemonics) or similarly purposed processing units [39]. These instructions, as their name indicate, perform a single instruction upon multiple data units. The main difference between processors are the available instructions, ranging from simple arithmetic to complex matrix

manipulations, the data width, such as single versus double float elements, and the element count (2, 4, 8, etc..) upon which they can operate simultaneously.

Their effectiveness is therefore dependant upon low level data parallelism and locality which typically occur when performing vector computation where the same instruction is to be applied to multiple consecutive elements (ie: consider the addition of two vectors). Their use has proven to generate code with significant speedup [15] but still require careful considerations with regards to memory access patterns [45].

To take advantage of these specialized instructions, the compilers need to be hinted both on the command line and through mindful coding practices so that the mathematical idioms are recognized by the compiler. As we have just demonstrated, the use of the generic implementation of `pow()` is to be avoided as it obfuscates the intended operation from the compiler and hampers optimization.

GCC's documentation states that hardware specific SIMD extensions are enabled through the option `-mfpmath=sse` coupled with a combination of flags such as `-msse`, `-msse2`, `-m3dnow`, and so on, depending on the hardware. In the case of more recent 64 bit hardware such as the `x86_64` based architectures[25], the extensions are enabled by default. By their nature, these instructions are typically used within loops and prove to be most effective when implemented in unrolled loops [15], our next topic.

3.6.4 Loop Optimizations

Probably some of the most popular topics in literature pertaining to HPC [25, 54, 31, 23], optimization of frequently called loops mostly consist in obtaining a higher computation versus control/branch ratio while reducing memory references to a minimum.

Instead of executing a single element of a loop and calling upon the indexing and break conditions, we execute multiple steps of the loop before, within and after the said loop. An example of one of these techniques, *loop unrolling*, is described in algorithm 7. In this case, we have

25. Which imply most current Intel and AMD processors.

unrolled the inner loop by a ratio of $4:1$ computations versus branch verification. The loop indice advances by steps of its unrolled power, four in this case, and the remainder of the index is executed in its regular form at the termination of the unrolled version.

```
1: for i = 0; i < (Size − 4); i+ = 4 do
2:    DATA[i ]=OP1[i ]+OP2[i ];
3:    DATA[i+1]=OP1[i+1]+OP2[i+1];
4:    DATA[i+2]=OP1[i+2]+OP2[i+2];
5:    DATA[i+3]=OP1[i+3]+OP2[i+3];
6: end for
7: if i mod Size {If some elements are left to be computed.} then
8:    for i < Size; i + + do
9:       DATA[i ]=OP1[i ]+OP2[i ];
10:   end for
11: end if
```

Algorithm 7: Loop Unrolling

The direct C code application of this technique is presented in Figure 3.19 where both the regular (left) and unrolled (right) versions of `df()` are presented.

Loop Unrolling

```
eric@thinkbigl ~/1_Files/1_ETS/1_Maitrise/Code/pvq $ diff --suppress-common-lines -y \
<(mpicc -E vq.c) \
<(mpicc -DUNROLL -E vq.c)
# 61 "vq.c"                                          | # 80 "vq.c"
inline float df(const float *v1, const float *v2)    | inline float df(const float *v1, const float *v2)
 float sum=0.0;                                      | float sum =0.0;
 int i;                                              | float sum1=0.0;
                                                     | float sum2=0.0;
                                                     | float sum3=0.0;
                                                     | int i=0;
                                                     |
                                                     | if (47>4){
 for(i=0; i<47; i++)                                 |   for(; i<(47-4); i+=4){
                                                     |     sum +=(v1[i ]-v2[i ])*(v1[i ]-v2[i ]);
 sum+=(v1[i]-v2[i])*(v1[i]-v2[i]);                   |     sum1+=(v1[i+1]-v2[i+1])*(v1[i+1]-v2[i+1]);
                                                     |     sum2+=(v1[i+2]-v2[i+2])*(v1[i+2]-v2[i+2]);
                                                     |     sum3+=(v1[i+3]-v2[i+3])*(v1[i+3]-v2[i+3]);
                                                     |  }
                                                     | }
                                                     > if (47%4)
                                                     >  for(; i<47; i++)
                                                     >  sum+=(v1[i]-v2[i])*(v1[i]-v2[i]);
                                                     >
                                                     > sum+=sum1;
                                                     > sum+=sum2;
                                                     > sum+=sum3;
```

Figure 3.19 : On the left, the original loop. On the right, the fourfold unrolled version of this same loop.

The self evident drawback of this approach is that it assumes the loop index to be high enough as to mask the added control latency imposed by this larger code base. Such manual modifications, other than inducing probable errors, make the code less legible and somewhat hardware dependent as the unrolling "level" is to be defined by the processor's characteristics such as data, instruction and address cache sizes. For these reason, it is preferable to let the compiler perform these optimizations.

Although most loop optimizations flags are set by the `-O3` general optimization level and, by their nature, should not impact the results, we have found that adding `-ffast-math` was *required* for the compiler to actually unroll the loops. This might be explained by the fact that, as it was mentioned in [15], code vectorization and loop optimization techniques tend to be tightly bound by nature of their application.

To inspect the use of the SIMD extensions and loop optimizations by comparing the assembly code for the `df()` function using both `-O3` and the combined `-O3 -ffast-math` flags. In Figure 3.20 , we see that the right-hand side has an unrolled version of the loop which also implements software pipelining prologue (lines $16-24$) before instructions are unrolled (lines $25-30$ repeated six times) and then all data is reconciled in the epilogue (not shown) with the added touch that the loop index is transformed into a decremented index (line 40), reputed to be a faster control approach on some hardware.

Being closely related to hardware, the impact of such optimizations will vary from platform to platform. This is well illustrated in Figure 3.21 where we compare above mentioned optimization approaches using the general flag `-O3`, then both `-O3 -ffast-math` and finally forcing the compiler to unroll all loops with `-funroll-all-loops`. Figure 3.21 (a) displays a time reduction of about 6% for the Athlon XP platform and we also note that forcing the unrolling of all loops proves to be detrimental to `df()`'s profiled time. Figure 3.21 (b) demonstrates that there is barely any gain obtained on the Intel $Q6600$.

In the case of the Intel $Q6600$, many reasons might explain the lack of performance gains. Apart from compiler adaptation to this recent platform, the hardware itself might actually be

Figure 3.20 : Pre-assembly output from GCC for an Athlon XP processor for `df()`. On the left, the code is compiled with explicit use of SIMD directives such as `-mfpmath=sse -msse -m3dnow`. On the right, the addition of `-ffast-math` has triggered unrolling of loop as well as additional use of the SIMD capabilities, generating more efficiently *vectorized* code.

more efficient and not require as much hand-tuning of the source code. Recall that most of the techniques pertaining to optimizing loops revolve around computation versus control ratios and data locality. Since the processors have grown dramatically in cache sizes, it is a fair bet that the $4M$ cache of the Intel $Q6600$, compared to the Athlon XP 's $512K$ [26], is having a significant impact which require that classic techniques be revisited and re-evaluated with regard to their implementation and pertinence.

Other approaches in HPC computing include hand coding the assembly. Although rather rare given the prohibitive efforts required to implement, there is a vectorial mathematical library by

26. Also noting that some of our models had $256K$.

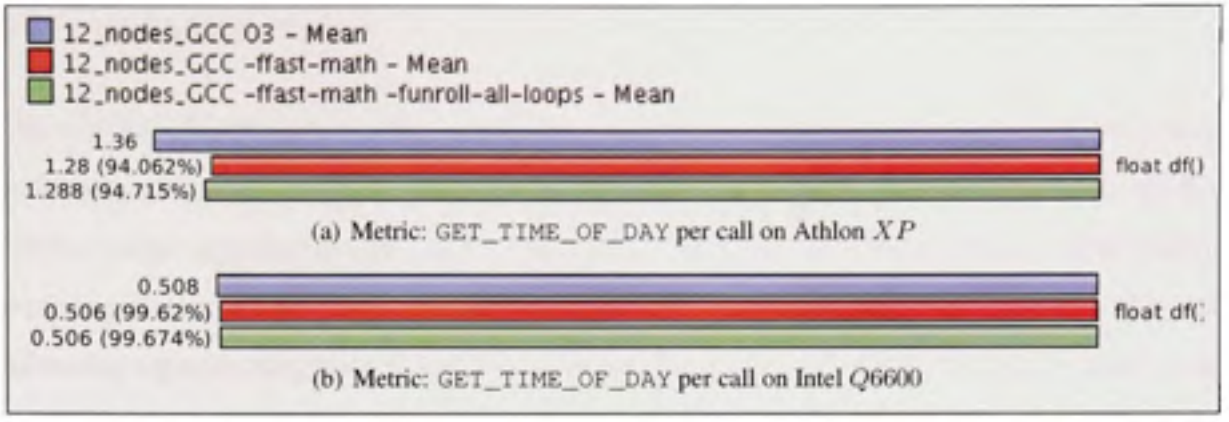


Figure 3.21 : Execution time comparison between using `-O3` (top bars in blue), adding `-ffast-math` (middle bars in red), and also addinf `-funroll-all-loops` (bottom bars in green). The (a) is for the execution time on Athlon XP processors where we can see that `df()` does not seem to benefit from `-funroll-all-loops` but does perform better with about 6% in time gain with only `-ffast-math`. (b) is on Intel $Q6600$ where very little differences are noted between the three approaches.

the name of its creator, Kazushige Goto, known as GOTO Basic Linear Algebra Subroutines (BLAS) which is the result of such strenuous efforts. We investigate its use in our next topic.

3.6.5 BLAS Libraries

The GOTO [20] implementation of BLAS is reputed to be the fastest since it has been hand written in assembler and fine tuned for all supported processors. We have replaced the Euclidean norm computation (the `df()`) function with its equivalent linear algebra mathematical representation using the Level 1 scalar-vector BLAS. This implementation is described by the following equation sequence where Eq. (3.8) performs copy of one of the vectors into a temporary *work area*, which is then added with the negated second vector in Eq. (3.9). The norm of the resulting vector is returned as a single scalar in Eq. (3.10).

$$Vdist \leftarrow v1 \tag{3.8}$$

$$Vdist \leftarrow -\alpha \cdot v2 + Vdist \tag{3.9}$$

$$ret \leftarrow \|Vdist\|_2 \tag{3.10}$$

This sequence translates into the code presented in Figure 3.22 , where each element of the original implementation are aligned with their equivalent BLAS call when possible. Note that the BLAS implementation actually performs the vector difference and norm in different steps while this is fused into a single line in the case of the C code implementation.

Using BLAS Routine

```
eric@thinkbigl ~/1_Files/1_ETS/1_Maitrise/Code/pvq > diff --ignore-blank-lines --suppress-common-lines -y 1
<icpic -E vq.c|egrep -v '^$' \
<icpic -DUSE_BLAS -E vq.c|
# 76 "vq.c"                                  | # 82 "vq.c"
                                             | float Vdist[47];
                                             |
inline float df(const float *v1, const float *v2)  | inline float df(const float *v1, const float *v2)|
{                                            |
 float sum=0.0;                              | cblas_scopy(47,v1,1,Vdist,1);
 int i;                                      |
 for(i=0; i<47; i++)                         | cblas_saxpy(47,-1.0,v2,1,Vdist,1);
 sum+=(v1[i]-v2[i])*(v1[i]-v2[i]);           |
 return sqrtf(sum);                          <  return cblas_snrm2(47,Vdist,1);
```

Figure 3.22 : The `df()` function using BLAS. On the left, the original loop. On the right, the BLAS version of this same loop. The operations on the right are aligned with the ones they (mostly) replace on the left.

We compare this use of the library in Figure 3.23 to our previously optimized version that used `-ffast-math`. As we can see, using the BLAS Level 1 library is detrimental to the

performance in our case from all points of view (time, computing cycles and all). According to [19], this is probably linked to limited loop unrolling capabilities in the Level 1 routines due to the lack of prior vector dimensionality knowledge, the same paradox faced by the compiler when unrolling loops. To investigate this further, we created a synthetic problem [27] calling upon the `df()` function repeatedly while varying the vector size. Our results, presented in Figure 3.23 (f), clearly demonstrate that there is no vector size where these libraries represent a performance gain.

It is therefore *not* recommended that *Level* 1 BLAS be used instead of plain C code.

27. this is the same program used to investigate cache saturation in Chapter 1, section 1.2.2

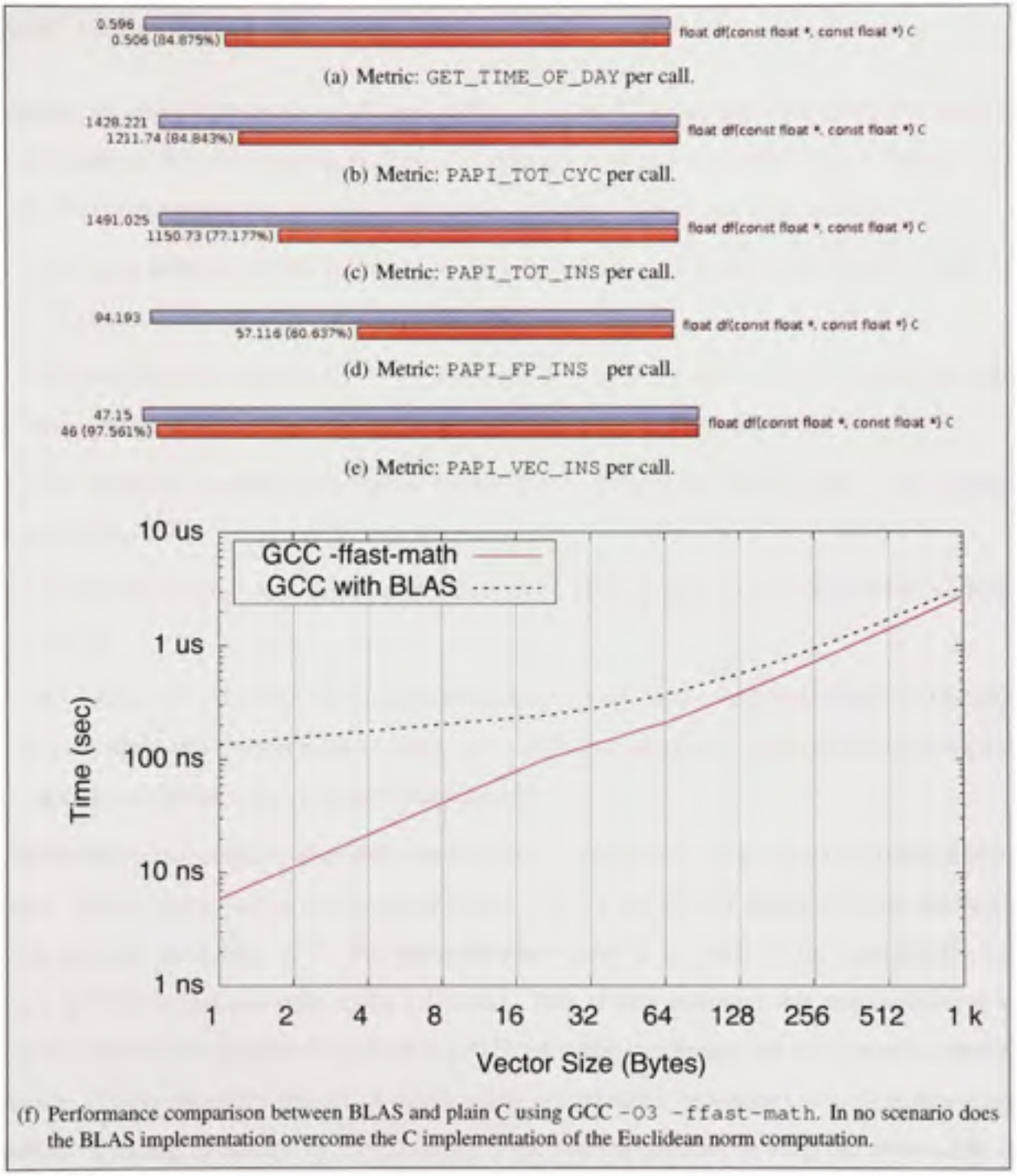


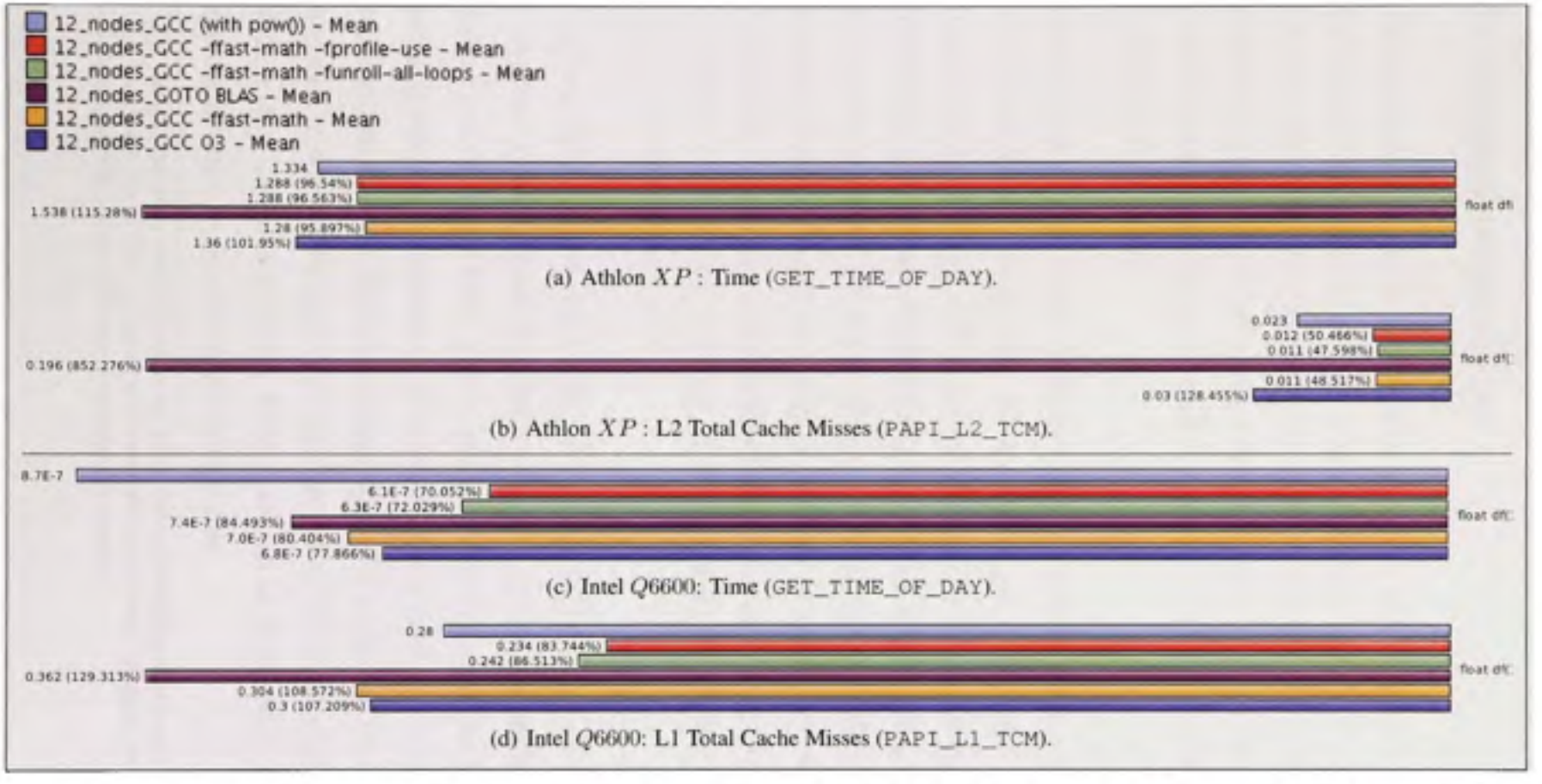
Figure 3.23 : The Level 1 BLAS libraraies (top blue bars and line) perform poorly in all cases compared to the code optimized with `-ffast-math`. This is reflected in all aspects of the computation whether it being time (a), CPU cycles (b), instructions (c) or even floating point operations ((d) and (e)). Further investigation by varying the vector size has proven this to always be the case as demonstrated in (f)

3.6.6 Comparing All Approaches

Finally, we collect the results of all approaches in Figure 3.24 , where we include the time per call results of `df()` for both the Athlon XP (Figure 3.24 (a)) and Intel $Q6600$ (Figure 3.24 (c)). This comparison in approaches and hardware brings forth many observations:

- Not worth using at all, the call to `pow()` is most detrimental on the Intel $Q6600$, where its performance is even worse than using BLAS;
- The general optimization flag, `-O3`, performs poorly on Athlon XP , even more so than using the `pow()` function, similar observations are made for the Intel $Q6600$;
- The compiler's profiling mechanism renders the best result on Intel $Q6600$, while average on Athlon XP ;
- The use of `-fast-math` is best on Athlon XP while its use alone is detrimental on Intel $Q6600$;
- On Athlon XP , the three best results are generally very close to eachother (within 1%) and are a variant of a combination of using `-ffast-math` and other more advanced compiler options not included in the general flags such as `-O3`.

Additionally, we correlate these time results with the total cache misses observed on each platform. The L2 cache misses displayed by Figure 3.24 (b) are clearly linked with the execution time seen for the Athlon XP . For the same observation to be made on the Intel $Q6600$, we have to observe the miss rate at the L1 cache. This clearly indicates that our application is mostly memory bound and that RAM to CPU bandwidth is essential for the execution performance. It also alleviates the use of data locality optimization techniques as well as any other means of taking advantage of the processor's prefetching abilities to keep the active data in local cache.



(a) Athlon XP : Time (`GET_TIME_OF_DAY`).

(b) Athlon XP : L2 Total Cache Misses (`PAPI_L2_TCM`).

(c) Intel $Q6600$: Time (`GET_TIME_OF_DAY`).

(d) Intel $Q6600$: L1 Total Cache Misses (`PAPI_L1_TCM`).

Figure 3.24 : Comparing all approaches Athlon XP (a) and Intel $Q6600$ (c). In both cases, BLAS (purple) and `pow()` (light blue) are the worst performing. A direct correlation is made between performance and L2 cache misses (b) for the Athlon XP . In the case of the Intel $Q6600$, the same clear cut correlation, requires that we go down to the L1 cache (d).

3.7 Looking at the Global Picture

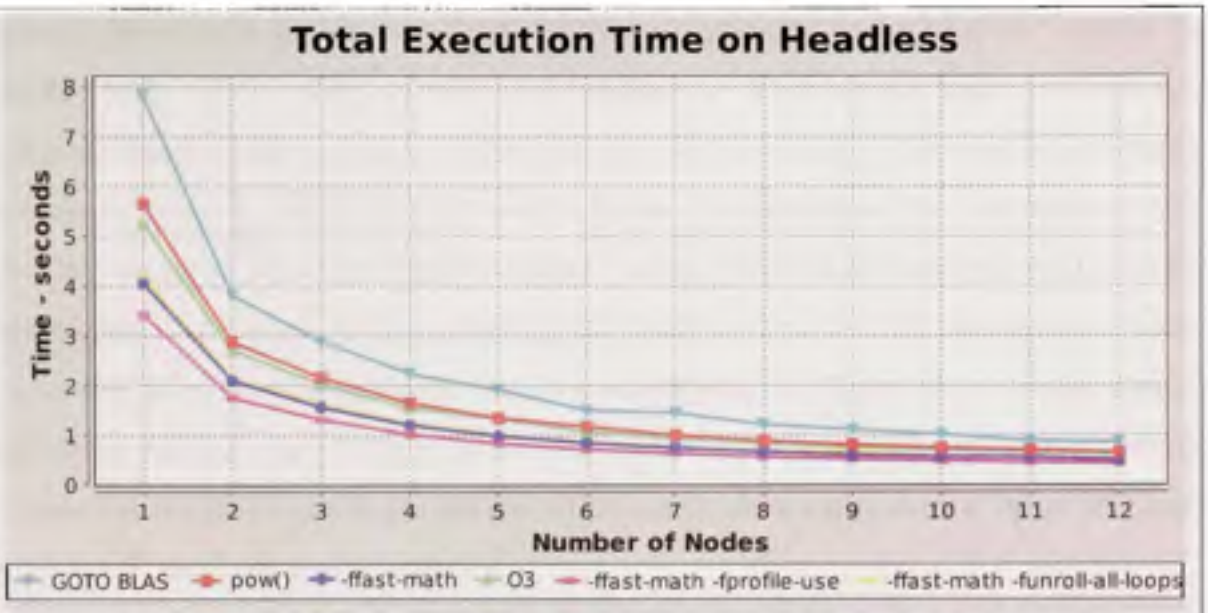
The significant impact that optimization strategies have on the cache state are bound to have repercussive effects on the program from a global point of view. It is therefore warranted that the execution of the program in its entirety be considered to ascertain its performance from a global perspective. Furthermore, even though it might be self evident, one must not forget that profiling induces significant overhead [28], especially for small computation kernels such as the two observed functions.

For this reason, it is always pertinent to compare profiled times with minimally (or ideally non-) profiled ones. In our case, we accomplish this by selecting TAU's minimal profile by including only MPI and PDT as the first is required for proper library linking and recalling that the latter for actually used inserting profile data into the source code [29]. We also perform this comparison in the parallel realm as to confirm that our proposed optimizations don't have adverse effects on the program when considering its parallel execution environment. Figure 3.25 contains the results of this time comparison executed on both the *Headless* cluster, based on Athlon XP hardware, in Figure 3.25 (a) and the H^2 cluster, based on the Intel $Q6600$ processor in Figure 3.25 (b).

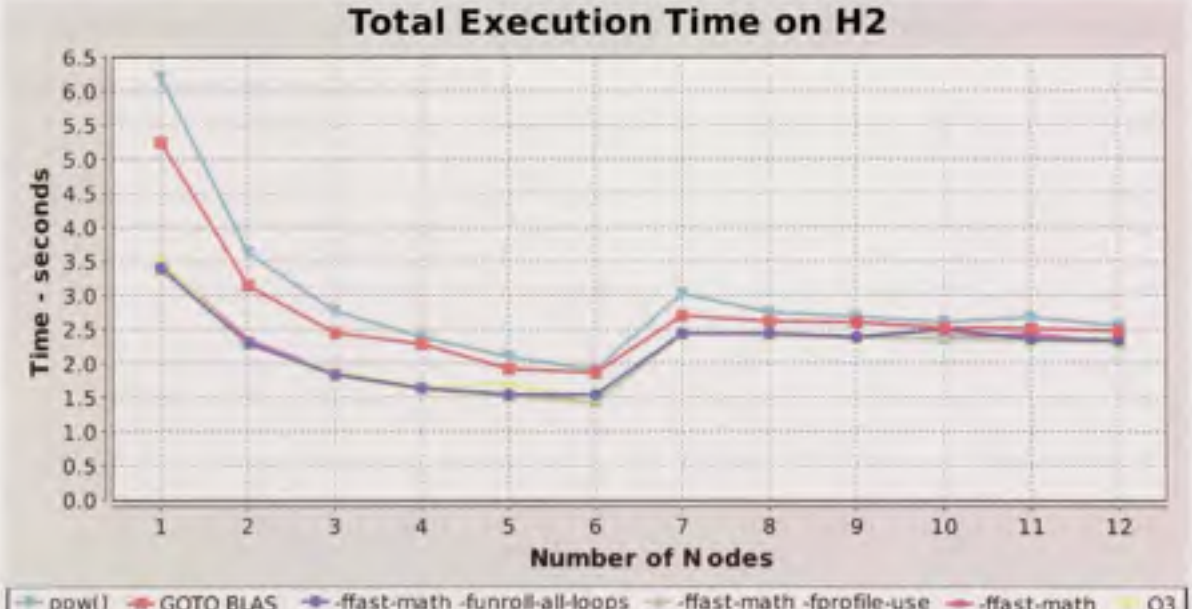
In the case of the Athlon XP architecture, the use of both `-ffast-math` and `-fprofile-use` come as the globally best approach, even though our profiling of `df()` had slated `-ffast-math` as the best. This is not too surprising since the profiling capabilities most probably optimized another area of the code, such as `centroid_def`, and that these two approaches had less than 1% differentiating them. The analysis of the Intel $Q6600$ architecture is less clear as most of the approaches overlap and no distinc advantage is given to one of them. Only a clear statement about the worst cases can be made, being that the `pow()` and BLAS approaches are to be avoided in our specific case.

28. Our profiled code rand as much as ten times slower, depending on selected counters.

29. Later versions of TAU are slated to have the ability of totally disabling the inserted profiling functions by switching to stub functions thanks to an environment variable.



(a) On the Athlon XP based cluster, there are four distinct time profiles out of six possibilities. Overlapping eachother are `O3` with `pow` as well as `-ffast-math` with `-ffast-math -funroll-all-loops`. The overall "winner" is the execution optimized with `-ffast-math -fprofile-use`



(b) On the Intel $Q6600$ based cluster, the profiles mostly overlap with the exception of the approaches using `pow()` and `GOTO BLAS`. No distinct advantage can be discerned from the other approaches. An execution jump is observed when seven nodes are reached, which we show to be attributed to OpenMPI's `MPI_Init()` in Figure 3.26 (a).

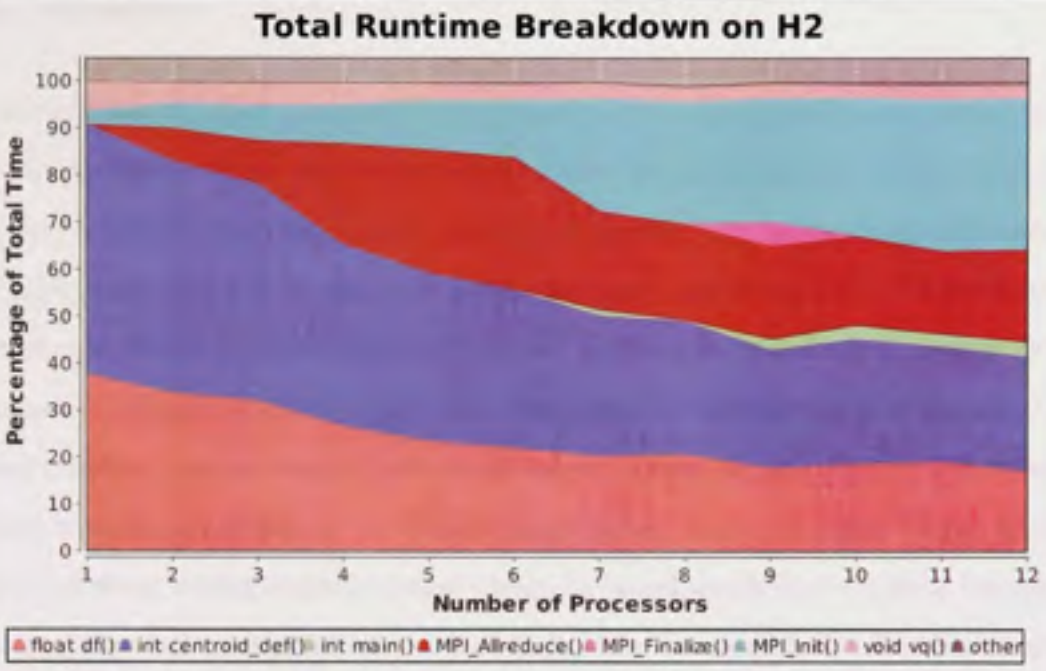
Figure 3.25 : Total execution times on both clusters. The Headless cluster (a), based on Athlon XP hardware, lends a distinct advantage to the use of `-ffast-math`. On the H^2 cluster (b), based on Intel $Q6600$ hardware, most options overlap leading to no clear "winner", barring the use of `GOTO BLAS` and `pow`.

Figure 3.26 (a) is the runtime breakdown for the best optimized option on the H^2 cluster. We note that `MPI_Init()` is responsible for the runtime jump between six and seven node execution and that the communications primitive, `MPI_Allreduce()` is growing in importance. As expected from the previous runtime results, the runtime breakdown from the *Headless* cluster is less *messy* as shown in Figure 3.26 (b). An argument could have been made that the computation is so fast on the newer Intel $Q6600$ hardware that the MPI routines were bound to take over in execution proportion. But, as seen in Figure 3.25 , the total execution time on both cluster actually place the older Athlon XP architecture ahead. This discrepancy could be explained by the fact that both clusters don't have exactly the same version of OpenMPI library ($1.2.8$ for H^2 and $1.2.9$ for *Headless*)

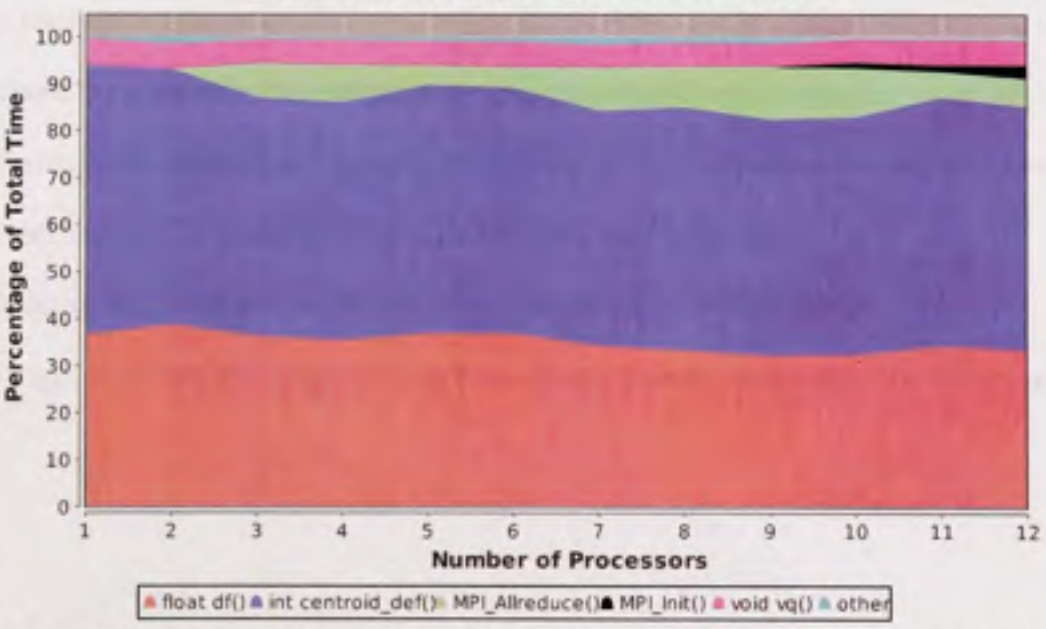
These result are interesting since they emphasize the fact that parallel models only taking into account the computation and communications can be completely off target when attempting to calculate the number of nodes to use to remain efficient [30]. They also bring forth the importance of keeping critical libraries up to date [31].

30. Recalling that efficiency is usually a $50/50$ ratio between computation and communications.

31. Note that the release notes bear no mention of performance changes made between the two aforementioned versions of OpenMPI.



(a) Runtime breakdown on H^2, `MPI_Init()` induces most of the jitter and an overhead jump after seven nodes. The communication are also growing in importance when observing`MPI_Allreduce()`.



(b) Runtime breakdown on *Headless*, most of the execution is spent in actual computation and little overhead is seen from the communications and the initialization function.

Figure 3.26 : The runrime breakdown for the best optimized options on both clusters. In (a) most of the execution time on the H^2 cluster is spent in MPI libraries. We see this is not the case in (b) for the *headless* cluster where most of the time is spent in computation.

3.8 Discussions

In this chapter, we have presented both a master-slave and a synchronous island model of the parallel K-Means. The synchronous island model was elaborated to address issues surrounding overly complex communication patterns of the original master-slave implementation and to enable computation and communications to overlap. By doing this, we have successfully replaced over fifteen communication pairs with a single collective communication. From the I/O perspective, important performance gain was obtained through the conversion of the ASCII based database into its binary format equivalent. Through profiling, the synchronous island model was optimized where six different approaches were compared. These included compiler directives, standard mathematical library calls and specialized vectorial libraries (BLAS). A correlation between performance and cache size was established for this memory bound algorithm. For our experiments, two architectures of completely different generations were compared, the Athlon XP and the Intel $Q6600$ processors.

Our final observations are that:

- There is no globally best solution or option to optimizing a program;
- Performance attainment requires profiling on a function level and on a global level;
- Profiling is to be performed for each new hardware platform;
- Process and environment initialization *must* be taken into account;
- Programs which are memory bound will always benefit from larger processor caches.

CONCLUSION AND FUTURE OUTLOOK

Our work set out to be an exploration of the profiling and optimization tools with the intent of defining the preferred hardware and software platform upon which to execute our characterized code. In Chapter 1, we have established that the typical problems encountered are memory bound and therefore would most benefit from processor with larger caches coupled with the fastest memory available. When network fabric was concerned, bigger and faster always come first but have an inherently high cost. With the advent of CMPs, virtually *communication less* parallel processing will become more and more important. However, the necessity to control execution concurrence of functions or programs accessing large sums of data will be required to ensure the processor cache is not being trashed. In the case of problems with large datasets loaded from disk, a clear advantage was set for distributed loading (local storage) of these sets after an initial propagation of the latter.

In Chapter 2, we shortly defined different approaches to profiling, which we differentiate using the terms *Black*, *White*, and *Grey Box*. We then quickly established the downfall and inappropriateness of classic profiling tools such as `gprof` when it comes to parallel HPC. An elaborate open source profiling suite, TAU was presented with its main GUI components being `paraprof`, the parallel profiling viewer, and `perfexplorer`, the performance analyzer mainly used for scalability and performance analysis. The use of support utilities such as the Program Database Toolkit (PDT), for automated *Grey Box* profiling and Performance Application Programming Interface (PAPI), for high precision and specialized measurements (such as floating point operations) were also demonstrated. Throughout the chapter, an example program and multiple synthetic setups were executed and profiled to demonstrate the suite's usage for identifying bottlenecks, with some warnings about possible misinterpretations.

Three implementations of the k-means algorithm were presented in Chapter 3. The material from both previous chapters served to surgically dissected the sequential and first parallel implementation (Master-Slave), which then served to spawn an improved implementation (the island model). Multiple performance optimization strategies were applied with special hardware

centric considerations as well as careful compiler directive selections. Optimal communications strategy, consolidating computation with data transmission, were employed to optimize the MPI aspect of the implementation. The proper use of global communicators were employed to simplify and offload the communication patterns to MPI's internal logic.

Finally, we believe that have demonstrated that parallel HPC coding requires close attention to the hardware characteristics as well as the necessity for attentive profiling of parallel code. The extensive profiling we have performed to identify the best optimization path has demonstrated that the exercise of attaining the best results in the field of HPC is an iterative process to be repeated with the each hardware platform for any given software.

Optimization Quick Reference

As we have stressed many times, optimizing execution performance is an iterative process given its dependence on code base and the environment upon which the latter is to be executed. Figure 3.27 is a deceptively simplified depiction of this iterative process where a change in any of the environmental or code elements, as we had presented them in Figure 1 , represent an entry point to the optimization process. As we have demonstrated, the application of each of these steps require a wide range of tools.

Attempting to propose a generalized solution would be futile and misleading. Nonetheless, we present in Table 3.1 a short list of the optimization techniques we have applied during the optimization process. It may be used as a quick reference when similar coding or execution paradigms are met. Obviously, this table is not meant to cover the entire realm of code optimization, there are many excellent books [54, 23] which cover this subject more appropriately.

The astute observer will note that most of these tactics have existed for well over a decade. In most cases, performance gain is obtained through consolidation of sparse data and streamlining its access, which is in essence minding data locality. We attribute this to the fact that contemporary computers are still mostly based on the *Von Neumann Architecture*, even the Chip MultiProcessorss (CMPs) may be considered a special case of this architecture.

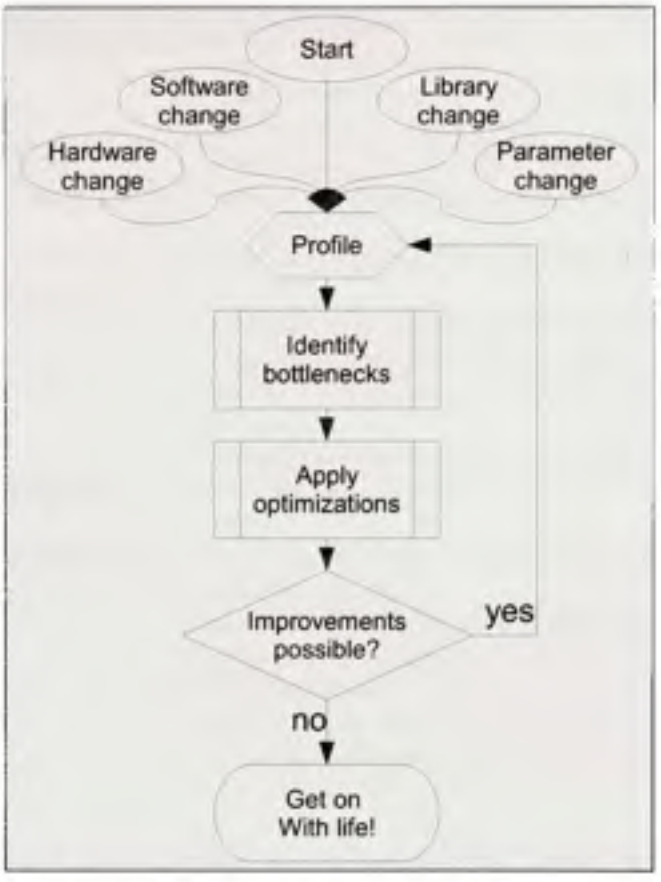


Figure 3.27 : A deceptively simple diagram depicting the iterative optimization process of a program. The multiple entry points recall that a change in any one of the elements from Figure 1 are susceptible to provoking a new optimization pass. The ultimate convergence being that there is no more possible improvements given a stabilized environment, and one can then *get on with life*.

Things To Come

The CMP, or multi-core processors, are now the *de facto* standard desktop processor with implied parallelism to harness their power. As we have demonstrated, differing architectures and cache structures offered by vendors don't make it a clear-cut choice which will provide the best performance, it's application specific. With the addition of a growing adoption of General Purpose Graphics Processing Units (GPGPUs), the parallel processing landscape is changing rapidly. The following is a condensed list of topics related to the realm of HPC not treated in this paper but with a significant growth in popularity in the last year. All of these are parallel

Category	Symptom/Cause(s)	Probable Solution Path(s)
I/O	Large files/databases	- Use binary formated files - Use local storage for frequently read data - Enhance storage performance
	Many small files	- Consolidate files into a single file
API and libraries	`MPI_INIT()` takes a long time	Switch to an MPI implementation that supports daemonization
	Most time is spent in an external API routine	Write your own implementation and *comparing results*
Communications	Many and small	- Consolidate if possible, use global communicators - Upgrade network fabric for low latency
	Many and large	- Fine tune OS specific parameters (ie: Jumbo Frames, caching parameters) - Upgrade network fabric for high bandwidth
Programming	Many calls to a small function	*Inline* the function's code
	Loop with embedded conditions	"Unswitch" the loop by creating independent loops with the conditions checked outside
	Loop applied to large datasets	Be mindful of data locality: explode or create data structures to consolidate the element(s) of interest for the loop
		- Simplify the loop's operations and exit conditions to let the compiler unroll and vectorize its execution - Don't make function calls within the loop

Table 3.1: Per bottleneck optimization recommendations. Prior profiling to identify the applicability of these approaches is primordial.

approaches which do *not* require a communication library such as OpenMPI but can very well be implemented in a hybrid context:

1) General Purpose Graphics Processing Units
   a) The Compute Unified Device Architecture (CUDA) library from NVIDIA [7, 16], is growing rapidly in importance in the realm of massively parallel computation adhering to the Single Program Multiple Data (SPMD) paradigm where the exact same sequence of instructions (execution kernel) is to be applied to a large dataset;
   b) Open Computing Language (OpenCL) [32], is a new standard describing a set of low level functions for parallel processing. It is meant to eventually supersede libraries such as CUDA to present a uniform access to multi-processing capable hardware. Although its use it not limited to GPU parallelization and includes CMPs and CELL processing

32. `http://www.khronos.org/opencl/`

units, mentioned below, most of the current work present its use in the realm of GPU processing.

2) Compiler and Coding Technologies
   a) GCC, newer versions (starting from 4.4), now support per-function optimization pragmas as well as an increasing number of optimization flags;
   b) OpenMP, although not technically a tool, its use requires slight modifications to the source code to automatically parallelized blocks of code;
   c) Low Level Virtual Machine (LLVM), a new modular compiler meant to generate faster and more efficient code.
3) Hardware
   a) CELL processors [30, 24], these multi-core platforms are growing rapidly in popularity;
   b) Intel's Quick Path Interconnect (QPI), a competitor to HT, is now starting to be available on the market, opening the doors to more multi-cpu platforms.

APPENDIX I

THE GNU C Compiler (GCC)

Unless otherwise noted, our experiments are based on GNU C Compiler version 4.3.2 (Gentoo $4.3.2 - r3\ p1.6,\ pie - 10.1.5$).

Code optimization is generally controlled using compiler directives, flags and options. Directives are defined via `#pragma` keys inserted in the source code. An example of such usage are the directives used to automate parallelization via OpenMP.

Flags are bi-valued command line *switches* that enable or disable features. Their general form is `-flag` for enabling a given `flag`, or `-noflag` for disabling this same `flag`. No all flags are performance related as some are used to enable features such as profiling [1], guided optimization [2], and even generate explanatory text files concerning decisions taken by the different heuristics engines [3] .

Options are more elaborate and accept either multiple values or a varying range of values. For example, it is possible to specify the L1 cache size of a processor via the `--param l1-cache-size=15k` parameter. Most of the options address internal variables used by GCC and can control its heuristical analisys of the source code during compilation. A demonstration of such an option follows.

GCC has over 144 flags, 77 of which are enabled by the global optimization flag `-O2` and 82 for `-O3`. As a general rule of tumbs, the third optimizaion level (`-O3`), is usualy considered to provide the best performing results while remaining *safe* [4]. Since these optimization levels are in fact a combination of individual flags, it is worthwhile to note the differences between he two levels. Recent versions of GCC make this easily possible through the command presented

1. Such as: `-fprofile-arcs -fprofile-generate`.
2. Such as:`-fbranch-probabilities -fprofile-use`
3. Notably, `-fdump-tree-vect-details -fdump-ipa-cgraph`.
4. In this context, code safeness mostly refers to the code's conformance to precision standards established by International Standards Organization (ISO) and IEEE standards

in Figure I.1, where we identify the flags disabled in the `-O2` level (thus enabled in `-O3`) thanks to the `--help=optimizers` option. The result of such a call can help guide the user as to which flags might toggled for performance comparisons. It can also come of use when attempting to idenfity which specific flags or options are included or not for a given architecture, for example, this would be accomplished by a command such as `gcc --help=target,joined`.

```
eric@fourrier ~ $ diff \
>  <(gcc -c -O2 -Q --help=optimizers ) \
>  <(gcc -c -O3 -Q --help=optimizers ) \
>  | grep disabled
<    -fgcse-after-reload                         [disabled]
<    -finline-functions                          [disabled]
<    -fpredictive-commoning                      [disabled]
<    -ftree-vectorize                            [disabled]
<    -funswitch-loops                            [disabled]
```

Figure I.1: disabled in the `-O2` level but enabled in `-O3`. As specified in the manpage for GCC, the `-O2` optimization level leaves out options that can grow the code size. This is to be considered if excessive instruction cache misses are found during the profiling of the application.

1 Help GCC Help You: Choosing the Right Flags

Unlike Fortran, which was destined for mathematical computation from its inception, the C[5] language was intended to be used as system programming language. This means that the assumptions made for Fortran do not apply to C.

For example, in C it is not uncommon to have two seemingly distinct variables point to the same location. This is known as variable or pointer aliasing and has a significant impact on the compiler's ability to implement optimizations which are data dependent. This is one of the many examples where the user can *tell* the compiler about the absence of such aliasing therefore permitting higher levels of optimization. This would be accomplished by enabling

5. And most other languages.

the `-fstrict-aliasing` flag. Note that this flag is actually enabled by default for most optimization levels. We present it as a meere example. As a matter of fact, the selection of optimization flags, givent their count and non-trivial implications, has become quite complex. As stated ealyer, general opimization flags, `-O2`, `-O3`, contain opions that make no assumptions about the code and ensure that there is no alteration of the expected output.

We will use our a priori knowledge of the source code and inspect `df()` and `centroid_def()` in Figure I.2 and note the following characteristics also considering knwon variables such such as the size of the T vector, and K, the centroid count.

Concerning `df()`:

- The heart of the loop is composed of three floating point operations;
- The loop is called T times, which we know to be 47 in our case;

Concerning `centroid_def()`:

- It calls `df()` K times;
- It gets called N/ω times[6] itself;
- The inner loop is dependant upon the return of the `df()` function call.

In both cases, we are dealing with simple mathematical kernels applied in a loop upon many elements of a given vector. Furthermore, the k-means computation is an iterative process where the K cenroids are re-computed at each iteration. The cummulative error or bias only apply to a single iteration. We also posses the knowledge that our mathematical evaluations are not using nor are they sensitive to boundary conditions, such as Not a Number (NaN) and Infinity (Inf), and we needn't distinguish between positive and negative zero values as every numerical values in the database are between 1 and 10^-9. Such a situation therefore implies optimizations that are proper to small loops and simple mathematical operations. These considerations permit the use of `-O3` in conjunction with `-ffast-math`, which are general optimization flags made up of a selection of other individual flags. The `-ffast-math` flag implements techniques known to have repercussions on the mathematical precision and also ignores many exceptional

6. The number of samples treated by the local worker.

Source code for df()

```
1  float df(const float *v1, const float *v2)
2  {
3          float sum=0.0;
4          int i;
5          for(i=0; i<T; i++)
6                  sum+=(v1[i]-v2[i])*(v1[i]-v2[i]);
7  }
```

(a) The `df()`, distance function, Euclidean computation function from the k-means implementation.

Source code for centroid_def()

```
1  inline int centroid_def(unsigned int idx, float *d)
2  {
3    register int i,centroid=-1;
4    float dist, mdist=999999999.;
5
6    for(i=0;i<K;i++) {
7      dist=df(&centroids[i*T], &samples[idx*T]);
8      if (dist < mdist) {
9        mdist=dist;
10       centroid=i;
11     }
12   }
13
14   *d=mdist;
15   return centroid;
16 }
```

(b) The `centroid_def()` centroid definition function. It calls the `df()` function K times and gets called X/ω times.

Figure I.2: The `df()`, distance function, Euclidean computation function from the k-means implementation.

conditions pertaining to boundary values. Still, to ensure the validity of the end results, the computetd centroids of each optimization technique is compared to the ones obtained by running a non-optimized, baseline version of the code. In all cases, the total summed distortion beetween each component was found to be null.

The following sections present our findings and results supporting theuse of such optimization flags in our context.

2 Let GCC Help You: Using Profiles

One of the last avenues we explore is the capability that most compilers possess of adapting optimization strategies with *a priori* knowledge of the code's behaviour thanks to specially generated profiles. This approach obviously implies that the code be compiled with specific flags to enable the profiling (`-fprofile-arcs` and `-fprofile-generate`) and then that it be recompiled with the explicit mention that the generated profiles be used (`-fprofile-use`). The intended outcome of this approach is that the compiler should generate code that uti-

lizes case-specific optimizations, prooven to be the best with the collected knowledge. This approach essentially provide measured values to the internal cost model heuristics of the compiler and also enables specific optimizations which depend on the availability of such profile. This is notably the case of the `-fbranch-probabilities` flag which is most significant in the area of control structures prevalent in loops.

Obviously, two phases are implied where the first one is composed of a trial execution and the second one consists in compiling with the generated data. We illustrate this in Figure I.3, a section of our project's `Makefile`, where a call to `make mpi` automatically compiles the application, a profiling version, runs it once with typical parameters, and then re-compiles it with the generated profile.

Makefile with profile based optimizations

```
1  GCCFLAGS_O3 = -Wall -Winline -march=native -O3 -save-temps
2  GCCFLAGS = $(GCCFLAGS_O3) \
3  -mfpmath=sse -msse -mthreads \
4  -ffast-math
5
6  GCC_PROFILE = $(GCCFLAGS) -fprofile-arcs -fprofile-generate
7  GCC_POST_PROFILE = $(GCCFLAGS) -fbranch-probabilities -fprofile-use -Wcoverage-mismatch
8
9  mpi:
10   mpicc $(GCCFLAGS) $(SRCS) -o $(PROGOUT)
11   mpicc $(GCC_PROFILE) $(SRCS) -o $(PROGOUT)_gcc-prof
12   ortexun -np 12 -hostfile ~/hosts ./$(PROGOUT)_gcc-prof /data/eric/featq_col.dat 10 342910
13   mpicc $(GCC_POST_PROFILE) $(SRCS) -o $(PROGOUT)_gcc-profiled
```

Figure I.3: Part of our `Makefile` used to generate and use GCC's profilie guided optimizations on Athlon XP hardware. The application is built calling `make mpi`, which will automatically generate the application, a profiling version, run a single execution and the compile a profile-guided version from the results of the previous run.

As we applied this approach, we have noticed that the best resutls are obtained if the profile phase is compiled with the same optimization flags as the final code using the profile. In other words, don't expect the profiler to automagically enable `-ffast-math` and don't simply enable it *after* the application was profiled. In essence, the approach should be used transparantly with all other compilation option and optimization techniques discussed earlyer.

APPENDIX II

COLLECTION OF COMMANDS

This section contains the extended version of logs and traces for commands and their output referred to throughout the document.

1 Identification of GCC Option Differences

The following sequence of commands are used to identify the inclusion of specific directives within global optimization flags. The basic technique is described in GCC's manual page and we present here our usage to obtain the data pertaining to 3.6 when attempting to identify probable paths to further optimizing code execution thanks to specific performance-centric options.

2 Taxonomy of the k-means Algorithm

Sequential k-means profiling

```
$ export TAU_MAKEFILE=~/TAU/TAU/x86_64/lib/Makefile.tau-callpath-pdt
$ tau_cc.sh -optCompile="-Wall -march=native -mfpmath=sse -O3 -fkeep-inline-functions -funsafe-loop-optimizations \
-freorder-blocks-and-partition -fno-math-errno -ffinite-math-only -fno-trapping-math -fno-signaling-nans \
-fwhole-program -combine -ffast-math -ftree-vectorizer-verbose=4 -fdump-tree-vect-details" vq.c -o vq_SIMD
$ ./vq_SIMD ./featq_col.dat $((6787955/100))
Limiting sample load to 67879 samples.
NSR = 67879
Take it easy, I am classifying...
TWO LAST AVERAGE DISTORTIONS: AD1=0.000000 AD2=0.463098 Dif=0.463098
TWO LAST AVERAGE DISTORTIONS: AD1=0.463098 AD2=0.369992 Dif=0.093105
TWO LAST AVERAGE DISTORTIONS: AD1=0.369992 AD2=0.352347 Dif=0.017645
TWO LAST AVERAGE DISTORTIONS: AD1=0.352347 AD2=0.343929 Dif=0.008418
TWO LAST AVERAGE DISTORTIONS: AD1=0.343929 AD2=0.339822 Dif=0.004107
TWO LAST AVERAGE DISTORTIONS: AD1=0.339822 AD2=0.337425 Dif=0.002398
TWO LAST AVERAGE DISTORTIONS: AD1=0.337425 AD2=0.335747 Dif=0.001678
TWO LAST AVERAGE DISTORTIONS: AD1=0.335747 AD2=0.334442 Dif=0.001305
TWO LAST AVERAGE DISTORTIONS: AD1=0.334442 AD2=0.333580 Dif=0.000862

Tempo total(s): 1333.028
```

Figure II.1: Profiling and execution of the sequential k-means algorithm using TAU. The program is then started by specifying the reference database and the number of samples to load from the database.

APPENDIX III

MACHINE DESCRIPTIONS

1 The *Thinkbig* Cluster

1.1 General Description

This Beowulf style cluster is composed of sixteen machines with two different processor specifications and interconnected using 100 BaseT Fast Ethernet. The topology consists of a logically flat networks with two switches bridging interconnected as described by Figure III.1. This classifies it as a slightly heterogeneous cluster with a fully connected topology. Communication paths between the nodes are direct while communications with the head node is split at the IP level between two links thanks to subnet separation. The network mask is set to a typicl class C of 255.255.255.0 with one broadcast domain from the point of view of the nodes. The serve has its NICs configured with a subclass of 255.255.255.128, where the first NIC is assigned the lower part of the address range and the second NIC the upper section.

1.2 Node Specifications

The two node types are described in table Table III.1 where the most significant hardware differences are outlined. The local disks vary in size between 20, 40 and 80 Gigs and performance caracteristics as illustrated by Figure III.2. This data was collected using the averages results for 30 runs of the Zoned Constant Angular Velocity (ZCAV) utility [1]. It is well illustrated that, in most cases [2], HDD transfer rate diminishes significantly as the data is located on higher order blocks.

All nodes are booted via Pre eXecution Environment (PXE) and share the user's `$HOME` folder via NFS with a local disk for scratch space.

1. Included with the *bonnie++* HDD performance suite
2. With the notable exception of one of the 80 Gigabyte HDDs which seems to be defective given its low and irregular performance.

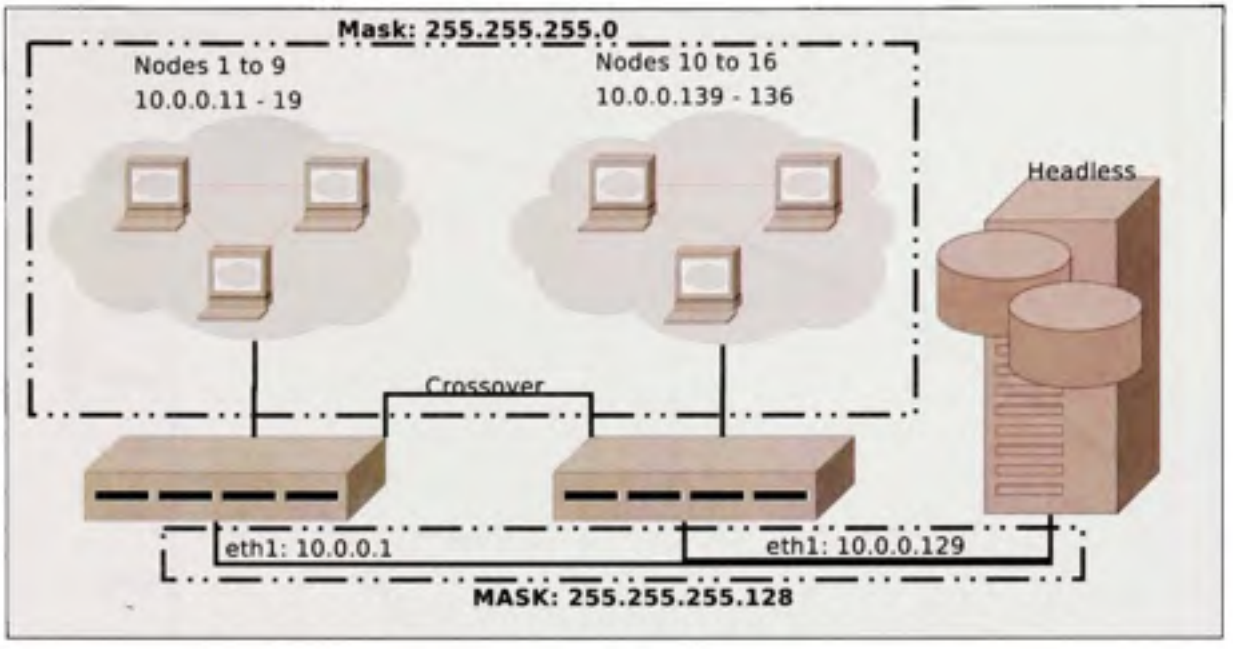


Figure III.1: Thinkbig Beowulf cluster topology

Parameters		Machine Profiles	
		A	B
Processor	Model Name	AMD Athlon(TM) XP 2500+	AMD Athlon(TM) XP 2600+
	Cache Size (KB)	512	256
	CPU MHz	1833.18	2083.158
	BogoMIPS	3669.17	4169.51

Table III.1: *Thinkbig* Node Specifications

1.3 Operating System

The cluster's OS is Gentoo based with important software versions described in table III.2.

Software	Version
GCC	Gentoo 4.3.1-r1 p1.1
ICC	Version 10.1 Build 20080602
OpenMPI	1.2.7 r19401
Linux Kernel	linux-2.6.17-gentoo-r4
PAPI	3.5.0

Table III.2: *Thinkbig* Software Specifications

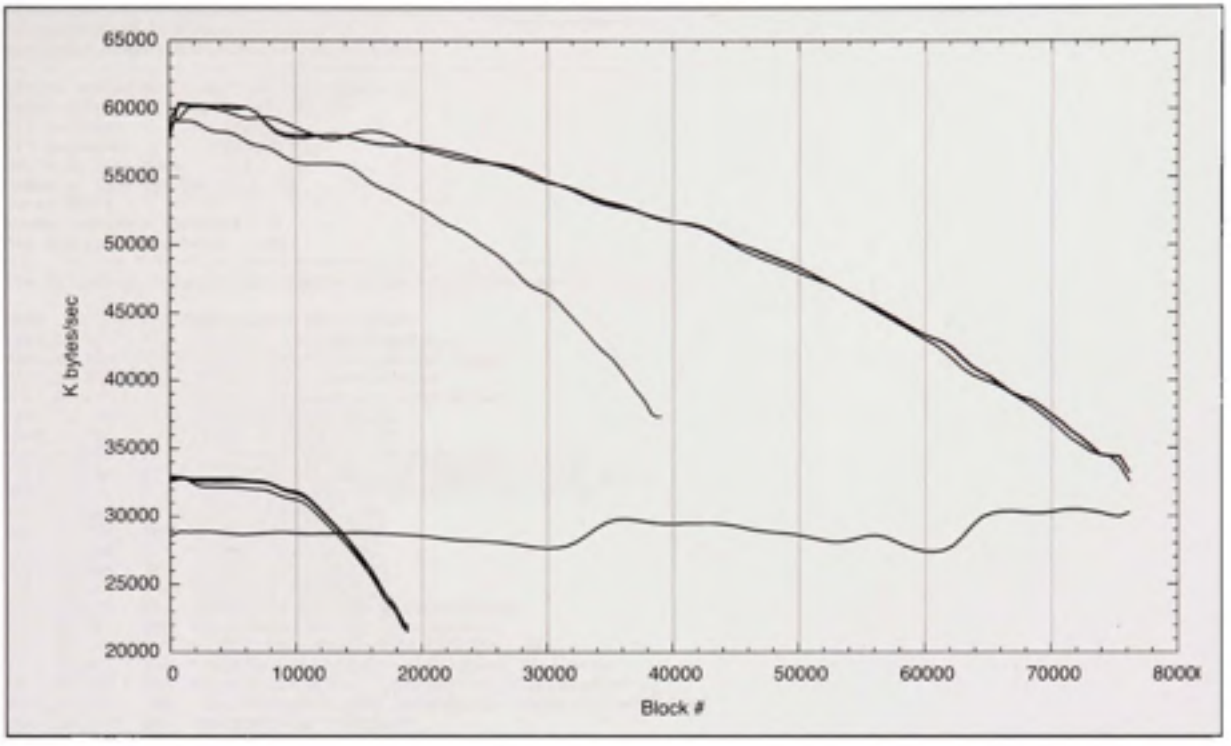


Figure III.2: The HDD's Zoned Constant Angular Velocity graph for 16 nodes of the *Thinkbig* cluster. These performance profiles illustrate well the heterogenety of the HDDs performance. The 40 and 80 G Byte HDDs start off with the same performance whereas the 20 G byte models are more than twice as slow.

1.4 Performance Application Programming Interface

The node's kernel was patched to support PAPI. Figure III.3 lists the available events.

papi_avail -a Output

```
eric@thinkbig1 ~ $ papi_avail -a
Available events and hardware information.
--------------------------------------------------------------------------------
Vendor string and code   : AuthenticAMD (2)
Model string and code    : AMD K7 (9)
CPU Revision             : 0.000000
CPU Megahertz            : 2083.157959
CPU's in this Node       : 1
Nodes in this System     : 1
Total CPU's              : 1
Number Hardware Counters : 4
Max Multiplex Counters   : 32
--------------------------------------------------------------------------------
The following correspond to fields in the PAPI_event_info_t structure.

Name        Derived  Description (Mgr. Note)
PAPI_L1_DCM   Yes    Level 1 data cache misses
PAPI_L1_ICM   No     Level 1 instruction cache misses
PAPI_L2_DCM   No     Level 2 data cache misses
PAPI_L2_ICM   No     Level 2 instruction cache misses
PAPI_L1_TCM   Yes    Level 1 cache misses
PAPI_L2_TCM   Yes    Level 2 cache misses
PAPI_TLB_DM   No     Data translation lookaside buffer misses
PAPI_TLB_IM   No     Instruction translation lookaside buffer misses
PAPI_TLB_TL   Yes    Total translation lookaside buffer misses
PAPI_L1_LDM   No     Level 1 load misses
PAPI_L1_STM   No     Level 1 store misses
PAPI_L2_LDM   No     Level 2 load misses
PAPI_L2_STM   No     Level 2 store misses
PAPI_HW_INT   No     Hardware interrupts
PAPI_BR_UCN   No     Unconditional branch instructions
PAPI_BR_CN    No     Conditional branch instructions
PAPI_BR_TKN   No     Conditional branch instructions taken
PAPI_BR_NTK   Yes    Conditional branch instructions not taken
PAPI_BR_MSP   No     Conditional branch instructions mispredicted
PAPI_BR_PRC   Yes    Conditional branch instructions correctly predicted
PAPI_TOT_INS  No     Instructions completed
PAPI_BR_INS   No     Branch instructions
PAPI_RES_STL  No     Cycles stalled on any resource
PAPI_TOT_CYC  No     Total cycles
PAPI_L1_DCH   Yes    Level 1 data cache hits
PAPI_L2_DCH   No     Level 2 data cache hits
PAPI_L1_DCA   No     Level 1 data cache accesses
PAPI_L2_DCA   Yes    Level 2 data cache accesses
PAPI_L2_DCR   No     Level 2 data cache reads
PAPI_L2_DCW   No     Level 2 data cache writes
PAPI_L1_ICA   No     Level 1 instruction cache accesses
PAPI_L2_ICA   No     Level 2 instruction cache accesses
PAPI_L1_ICR   No     Level 1 instruction cache reads
PAPI_L1_TCA   Yes    Level 1 total cache accesses
--------------------------------------------------------------------------------
avail.c                  PASSED
```

Figure III.3: Output listing of all PAPI events as per `papi_avail -a` for the Athlon XP processors.

2 The H^2 Cluster

2.1 General Description

This Beowulf style cluster is composed of nine machines each possessing a single Intel Intel *Q*6600 Quad Core processor and interconnected using Gigabyte Ethernet. The topology consists of a flat networks with a single Dell Powerconnect 2745 switch left in unmanaged mode.

This classifies it as a homogeneous cluster with a fully connected topology. Communication paths between the nodes and the master are direct.

2.2 Node Specifications

All nodes are identically built with an Intel $Q6600$ processor, 4GB of RAM and a local Serial Advanced Technology Attachment (SATA) HDD of 500GB. A more detailed description is presented in Table III.3, note that the processor cache size is shared amongst all four cores while other specifications are for each independent core.

Parameters		Machine Profile
Processor	Model Name	Intel(R) Core(TM)2 Quad CPU Q6600
	Cache Size (KB)	4096
	CPU MHz	2400
	BogoMIPS	4800
HDD	Brand	Western Digital
	Model Name	WD5000AAKS-0
	Cache Size (MB)	16
	Capacity (GB)	500
Motherboard	Brand	ASUSTeK Computer INC.
	Model Name	P5N7A-VM
	Revision	Rev 1.xx
RAM	Installed (GB)	4
	Speed (MHz)	800
	Count	2

Table III.3: H^2 Node Specifications.

2.3 Operating System

The cluster's OS is Gentoo based with important software versions described in table III.4.

2.4 Performance Application Programming Interface

The node's kernel was patched to support PAPI. Figure III.4 lists the available events.

Software	Version
GCC	Gentoo 4.3.3-r2 p1.1, pie-10.1.5
ICC	10.1 Build 20080801
OpenMPI	1.2.7 r19401
Linux Kernel	2.6.25-gentoo-r7
PAPI	3.6.2

Table III.4: H^2 Software Specifications

papi_avail -a Output

```
eric@node01 ~ $ papi_avail -a
Available events and hardware information.
--------------------------------------------------------------------------------
Vendor string and code   : GenuineIntel (1)
Model string and code    : Intel Core 2 (18)
CPU Revision             : 11.000000
CPU Megahertz            : 2399.969971
CPU Clock Megahertz      : 2399
CPU's in this Node       : 4
Nodes in this System     : 1
Total CPU's              : 4
Number Hardware Counters : 5
Max Multiplex Counters   : 32
--------------------------------------------------------------------------------
The following correspond to fields in the PAPI_event_info_t structure.

    Name         Code    Deriv Description (Note)
PAPI_L1_DCM  0x80000000  No   Level 1 data cache misses
PAPI_L1_ICM  0x80000001  No   Level 1 instruction cache misses
PAPI_L2_DCM  0x80000002  Yes  Level 2 data cache misses
PAPI_L2_ICM  0x80000003  No   Level 2 instruction cache misses
PAPI_L1_TCM  0x80000006  No   Level 1 cache misses
PAPI_L2_TCM  0x80000007  No   Level 2 cache misses
PAPI_CA_SHR  0x8000000a  No   Requests for exclusive access to shared cache line
PAPI_CA_CLN  0x8000000b  No   Requests for exclusive access to clean cache line
PAPI_CA_ITV  0x8000000d  No   Requests for cache line intervention
PAPI_TLB_DM  0x80000014  No   Data translation lookaside buffer misses
PAPI_TLB_IM  0x80000015  No   Instruction translation lookaside buffer misses
PAPI_L1_LDM  0x80000017  No   Level 1 load misses
PAPI_L1_STM  0x80000018  No   Level 1 store misses
PAPI_L2_LDM  0x80000019  Yes  Level 2 load misses
PAPI_L2_STM  0x8000001a  No   Level 2 store misses
PAPI_HW_INT  0x80000029  No   Hardware interrupts
PAPI_BR_CN   0x8000002b  No   Conditional branch instructions
PAPI_BR_TKN  0x8000002c  No   Conditional branch instructions taken
PAPI_BR_NTK  0x8000002d  No   Conditional branch instructions not taken
PAPI_BR_MSP  0x8000002e  No   Conditional branch instructions mispredicted
PAPI_BR_PRC  0x8000002f  Yes  Conditional branch instructions correctly predicted
PAPI_TOT_IIS 0x80000031  No   Instructions issued
PAPI_TOT_INS 0x80000032  No   Instructions completed
PAPI_FP_INS  0x80000034  No   Floating point instructions
PAPI_BR_INS  0x80000037  No   Branch instructions
PAPI_VEC_INS 0x80000038  No   Vector/SIMD instructions
PAPI_RES_STL 0x80000039  No   Cycles stalled on any resource
PAPI_TOT_CYC 0x8000003b  No   Total cycles
PAPI_L1_DCH  0x8000003e  Yes  Level 1 data cache hits
PAPI_L1_DCA  0x80000040  No   Level 1 data cache accesses
PAPI_L2_DCA  0x80000041  Yes  Level 2 data cache accesses
PAPI_L2_DCR  0x80000044  No   Level 2 data cache reads
PAPI_L2_DCW  0x80000047  No   Level 2 data cache writes
PAPI_L1_ICH  0x80000049  Yes  Level 1 instruction cache hits
PAPI_L2_ICH  0x8000004a  Yes  Level 2 instruction cache hits
PAPI_L1_ICA  0x8000004c  No   Level 1 instruction cache accesses
PAPI_L2_ICA  0x8000004d  No   Level 2 instruction cache accesses
PAPI_L2_TCH  0x80000056  Yes  Level 2 total cache hits
PAPI_L1_TCA  0x80000058  Yes  Level 1 total cache accesses
PAPI_L2_TCA  0x80000059  No   Level 2 total cache accesses
PAPI_L2_TCR  0x8000005c  Yes  Level 2 total cache reads
PAPI_L2_TCW  0x8000005f  No   Level 2 total cache writes
PAPI_FML_INS 0x80000061  No   Floating point multiply instructions
PAPI_FDV_INS 0x80000063  No   Floating point divide instructions
PAPI_FP_OPS  0x80000066  No   Floating point operations
--------------------------------------------------------------------------------
Of 45 available events, 10 are derived.

avail.c                  PASSED
```

Figure III.4: Output listing of all PAPI events as per `papi_avail -a` for the Intel $Q6600$ processor.

3 The Multiprocessor Servers

The two SMP machines used for our experimentations both possessed 32GBytes of RAM and 8 Dual Core AMD Opteron processors. Table III.5 lists their key hardware caratersitics. Software caracteristics are listed in Table Table III.6, cells containing '-' mean the software wasn't installed on the specific machine. Note that PAPI was not installed on these systems either.

Parameters		Machine Profiles	
		SunFire x4600	Tyan VX50
Processor	Model Name	Processor 885	Processor 875
	Cache Size (KB)	1024	1024
	CPU MHz	2600	2200
	BogoMIPS	5226	4420

Table III.5: SMP **machine hardware specifications**

Software	**Version**
GCC	Gentoo 4.3.1-r1 p1.1
ICC	Version 10.1 Build 20080602
OpenMPI	1.2.7 r19401
Linux Kernel	linux-2.6.17-gentoo-r4
PAPI	3.5.0

Table III.6: SMP **machine software specifications**

APPENDIX IV

SOURCE CODE

This section contains the printout of the principal source code used in our experimentation. When reasonable, the code was left untouched. When applicable, blocks of commeted test code were removed for clarity.

1 The Island Master-Slave Implementation

The following is the original implementation of the Master-Slave k-means.

```
1  /*&=============================================================================&*
2   *&                                                                             &*
3   *& Module Name:  vector quantisation based on k-means algorithm                &*
4   *&               (Parallel Algorithm)                                          &*
5   *& This is a C++ program with MPI library                                      &*
6   *& Authors:      Alceu Britto / Albert Hung-Ren Ko                             &*
7   *&                                                                             &*
8   *&                                                                             &*
9   *&                                                                             &*
10  *& To compile with the Makefile: make                                          &*
11  *& To set up the topology of kernels: lamboot -v lamconf.lam                   &*
12  *& To run: mpirun -v -np #(number of kernels) pvq filename                     &*
13  *& To erase the set topology of kernels: wipe                                  &*
14  *& It will generate the file: centroids                                        &*
15  *&=============================================================================&*/
16
17  #include <stdio.h>
18  #include <ctype.h>
19  #include <string.h>
20  #include <stdlib.h>
21  #include <math.h>
22  #include <iostream>
23  #include <fstream>
24  #include <iomanip>
25  #include <cassert>
26  #include <sstream>
27  #include <mpi.h>
28  using namespace std;
29
30  #include <ctime>
31
32  #define THRESHOLD 0.001  /* threshold used to stop iterations */
33  #define T   34       /* size of the feature vector */
34  #define NC  256      /* number of centroids */
35
36  int NSR;              /* number of samples */
37  int SKIP;             /* NSR divided by NC */
38  int NS;    /* maximum number of samples */
39
```

```
40  /* struct used to keep a feature vector and its centroid */
41  typedef struct
42  {
43      float feat[T];
44      int centroid;
45  } sample;
46
47  /* struct used to keep a centroid and the number of samples in it*/
48  typedef struct
49  {
50      float feat[T];
51      float number;
52  } centroid;
53
54
55  sample *samples;          /* keep all training samples    */
56  centroid centroids[NC];   /* keep all centroids */
57  float c_sum[NC][T];       /* sum of all samples of a class, it is used to update the centroids */
58  int mynode, totalnodes;
59  int slaves =1;
60  int master = 0;
61  int tag=1;
62  int sum, startval, endval, accum;
63  int i,j,k;
64  MPI_Status status;
65  int loadCount = 0;
66
67  /* distance function - Euclidian Distance */
68  float df ( float *v1, float *v2 )
69  {
70      int i;
71      float dist, sum;
72
73      sum=0.;
74      for ( i=0; i<T; i++ )
75          sum=sum+ ( v1[i]-v2[i] ) * ( v1[i]-v2[i] );
76
77      dist= ( float ) sqrt ( ( float ) sum );
78      return dist;
79  }
80
81  /* load samples */
82  int load_samples ( char *filename )
83  {
84      FILE *fp;
85      int i,j;
86      int Obs;
87      int reg_size = sizeof ( float ) * T;
88      ifstream inStream ( filename );
89      loadCount = 0;
90      string line;
91      int lineCount = 0;
92
93      fp=fopen ( filename, "r" );
94
95      if ( !fp )
96      {
97          printf ( "can't open thou file: %s\n", filename );
98          return ( 0 );
99      }
```

```
100     fseek ( fp, 0, SEEK_END );
101
102     NS = ( int ) ftell ( fp ) / reg_size;
103
104     if ( totalnodes > 1 )
105     {
106         samples= ( sample * ) malloc ( ( NS/ ( totalnodes-1 ) ) *sizeof ( sample ) );
107     }
108
109     else
110     {
111         samples= ( sample * ) malloc ( ( NS ) *sizeof ( sample ) );
112     }
113
114     //fseek(fp, 0, SEEK_SET);
115     if ( !samples ) return -1;
116
117     fclose ( fp );
118
119     /* load samples */
120
121     while ( !inStream.eof() && lineCount < NS )
122     {
123         getline ( inStream, line );
124         istringstream istr ( line );
125         if ( ( fmod ( ( lineCount+1 ), ( totalnodes -1 ) ) == ( mynode -1 ) ) && ( mynode != master ) )
126         {
127             for ( i = 0; i< T; i++ )
128             {
129                 istr >> samples[loadCount].feat[i];
130                 samples[loadCount].centroid=-1;
131             }
132             loadCount++;
133         }
134         lineCount ++;
135     }
136
137     if ( mynode  == ( 1 + fmod ( ( lineCount ), ( totalnodes - 1 ) ) ) )
138     {
139         loadCount --;
140     }
141     // 'cause the last kernel will load the end line of the file
142
143     NS = ( lineCount -1 ); //minus one because there is one empty line at the end of the file fp
144
145     printf ( "Final_NS_=_%d\n", NS );
146     //cout << " mynode " << mynode << " loadCount " << loadCount << endl;
147     return ( lineCount -1 );  /* return i-1 when binary mode */
148 }
149
150 /* centroid initialization - it selects the first set of centroids */
151 void centroid_init()
152 {
153     int i,j,k,x;
154
155     for ( i = 0; i < NC; i++ )
156     {
157         x = 0;
158         if ( mynode == fmod ( ( i ), ( totalnodes -1 ) ) +1 )
159         {
```

```
160             for ( j=0;j<T;j++ )
161             {
162                 centroids[i].feat[j]= samples[x].feat[j];
163                 //cout << " " << samples[x].feat[j];
164                 centroids[i].number=0;
165             }
166             //cout << endl << " mynode= " << mynode << " x= " << x << endl;
167             x++;
168             for ( k=0; k < totalnodes; k++ )
169             {
170                 MPI_Send ( &centroids[i], ( T+1 ), MPI_FLOAT, k, tag+9, MPI_COMM_WORLD );
171             }
172         }
173         MPI_Recv ( &centroids[i], ( T+1 ), MPI_FLOAT, ( int ) ( fmod ( ( i ), ( totalnodes -1 ) ) +1 ), tag+9, MPI_COMM_WORLD
                , &status );
174     }
175 }
176
177 /* classification of a sample taking into account each centroid */
178 int centroid_def ( int pos, float *d )
179 {
180     int i,index;
181     float mdist, dist;
182
183     mdist=999999999.;
184     for ( i=0;i<NC;i++ )
185     {
186         dist=df ( centroids[i].feat,samples[pos].feat );
187         if ( dist < mdist ) {mdist=dist; index=i;};
188     }
189
190     *d=mdist;
191     return index;
192 }
193
194 /* it calculates new centroids */
195 void mean_vector()
196 {
197     int i,j,c;
198     float master_c_number[NC];
199
200     for ( c=0;c<NC;c++ )
201         if ( centroids[c].number != 0 )
202             for ( j=0;j<T;j++ )
203                 centroids[c].feat[j] = c_sum[c][j]/centroids[c].number;
204
205     for ( i = 1; i < totalnodes; i++ )
206         MPI_Send ( &centroids , ( NC* ( T+1 ) ), MPI_FLOAT, i, tag, MPI_COMM_WORLD );
207 }
208
209
210 /* it calculates the mean distortion */
211 float average_distortion ( float *x )
212 {
213     int i;
214     float ad;
215     ad=0;
216     for ( i=0;i<NC;i++ )
217         ad=ad+x[i];
218
```

```
219     ad=ad/NSR;
220     return ad;
221 }
222
223 void sq()
224 {
225     int iteration;
226     int centr,j,i, k, c;
227     float distortion, distortion_ant, distc[NC], dist;
228     distortion=0;
229     iteration=1;
230
231     if ( mynode == master )
232         printf ( "Take_it_easy._I_am_classifying...\n" );
233
234     do
235     {
236         /* Initialization   */
237         for ( i=0;i<NC;i++ )
238         {
239             centroids[i].number=0; distc[i]=0.;
240             for ( j=0; j<T;j++ ) c_sum[i][j]=0.;
241         };
242
243
244         if ( mynode != master )
245         {
246             float send_c_number[NC];
247             j = 0;
248             while ( j < loadCount )
249             {
250                 centr=centroid_def ( j, &dist );
251                 samples[j].centroid=centr;
252                 distc[centr]=distc[centr]+dist;
253                 centroids[centr].number = ( centroids[centr].number ) + 1;
254                 for ( i=0;i<T;i++ )
255                     c_sum[centr][i]+=samples[j].feat[i];
256
257                 j++;
258             }
259
260             //parallelize this parts
261             for ( i = 0; i < NC; i++ )
262                 send_c_number[i] = centroids[i].number;
263
264             MPI_Send ( &c_sum, ( NC*T ), MPI_FLOAT, master, tag+1, MPI_COMM_WORLD );
265             MPI_Send ( &distc, NC, MPI_FLOAT, master, tag+3, MPI_COMM_WORLD );
266             MPI_Send ( &send_c_number, NC, MPI_FLOAT, master, tag+2, MPI_COMM_WORLD );
267
268             //receive from the broadcast
269             MPI_Recv ( &centroids, ( NC* ( T+1 ) ), MPI_FLOAT, master, tag, MPI_COMM_WORLD, &status );
270             MPI_Recv ( &distortion_ant, 1, MPI_FLOAT, master, tag+5, MPI_COMM_WORLD, &status );
271             MPI_Recv ( &distortion, 1, MPI_FLOAT, master, tag+6, MPI_COMM_WORLD, &status );
272         }
273
274         if ( mynode == master )
275         {
276
277             float slave_c_sum[NC][T];
278             float slave_distc[NC];
```

```
279             float master_centroids_number[NC];
280
281             for ( j = 1; j < totalnodes; j++ )
282             {
283                 MPI_Recv ( &slave_c_sum , ( NC*T ), MPI_FLOAT, j, tag+1, MPI_COMM_WORLD, &status );
284                 MPI_Recv ( &slave_distc , NC, MPI_FLOAT, j, tag+3, MPI_COMM_WORLD, &status );
285                 MPI_Recv ( &master_centroids_number , NC, MPI_FLOAT, j, tag+2, MPI_COMM_WORLD, &status );
286                 for ( i = 0; i < NC; i++ )
287                 {
288                     for ( k = 0; k < T; k++ )
289                         c_sum[i][k] = c_sum[i][k] + slave_c_sum[i][k];
290
291                     distc[i] =  distc[i]+ slave_distc[i];
292                     centroids[i].number = centroids[i].number + master_centroids_number[i];
293                 }
294
295             }
296
297             mean_vector();
298
299             distortion_ant=distortion;
300             distortion=average_distortion ( distc );
301
302             for ( i = 1; i < totalnodes; i++ )
303             {
304                 MPI_Send ( &distortion_ant , 1, MPI_FLOAT, i, tag+5, MPI_COMM_WORLD );
305                 MPI_Send ( &distortion , 1, MPI_FLOAT, i, tag+6, MPI_COMM_WORLD );
306             }
307         }
308
309         iteration++;
310     }
311     while ( fabs ( ( float ) ( distortion_ant - distortion ) ) > THRESHOLD );
312
313     cout << "iteration " << iteration << endl;
314 }
315
316 /* show centroids */
317 void show_centroids()
318 {
319     int i,j;
320
321     printf ( "Centroids\n" );
322     for ( i=0;i<NC;i++ )
323     {
324         for ( j=0;j<T;j++ )
325             if ( i == 0 || i== ( NC-1 ) ) printf ( "%2.2f ", centroids[i].feat[j] );
326
327         if ( i == 0 || i== ( NC-1 ) ) printf ( "\n" );
328     }
329 }
330
331 /* show samples */
332 void show_samples()
333 {
334     int i, j;
335     for ( i=0;i<NSR;i++ )
336     {
337         for ( j=0;j<T;j++ ) printf ( "%f ", samples[i].feat[j] );
338         printf ( " c=%d\n", samples[i].centroid );
```

```
339     }
340 }
341
342 /* save centroids */
343 void save_centroids()
344 {
345     int i,j;
346     FILE *fp;
347
348     fp=fopen ( "centroids", "wb" );
349     /*printf("Saving centroids \n");*/
350     for ( i=0;i<NC;i++ )
351     {
352         for ( j=0;j<T;j++ )
353             fprintf ( fp, "%f,", centroids[i].feat[j] );
354         fprintf ( fp, "\n" );
355     }
356     fclose ( fp );
357 }
358
359
360 /* main */
361 main ( int argc, char *argv[] )
362 {
363     char *fnamein;
364
365     time_t tempo1, tempo2, tempo3;
366     float tempo;
367
368     fnamein=argv[1];
369
370     MPI_Init ( &argc, &argv );
371     MPI_Comm_size ( MPI_COMM_WORLD, &totalnodes );
372     MPI_Comm_rank ( MPI_COMM_WORLD, &mynode );
373
374     /* load samples */
375
376     NSR=load_samples ( fnamein );
377
378     time ( &tempo1 );
379
380     if ( mynode == master )
381     {
382         if ( NSR== -1 ) {printf ( "error in loading sample file\n" ); exit ( 1 );}
383         printf ( "NSR = %d\n", NSR );
384         time ( &tempo2 );
385         tempo = difftime ( tempo2, tempo1 );
386         cout << endl << "Loading time(s): " << tempo << endl;
387     }
388
389     centroid_init();
390
391     sum = 0;
392
393     if ( mynode != master )
394     {
395         startval = ( NSR* ( mynode-1 ) / ( totalnodes-1 ) ) +1;
396         endval = NSR* ( mynode ) / ( totalnodes-1 );
397     }
398
```

```
399     /* vector quantisation */
400     vq();
401
402     if ( mynode == master )
403     {
404         //show_centroids();
405         save_centroids();
406         time ( &tempo3 );
407         tempo = difftime ( tempo3, tempo2 );
408         printf ( "\nExecution_time(s):_%.3f\n", tempo );
409     }
410     MPI_Finalize();
411     return 0;
412 }
```

2 The Island k-means Implementation

The following is the implementation of the Island k-means we have implemented and used througout the document and experiments. It is an evolution of the orignial code presented above.

```
1  /*&=============================================================================&*
2   *&                                                                             &*
3   *& Module Name:   vector quantisation based on k-means algorithm               &*
4   *& Author:        Alceu Bissoto                                                &*
5   *&                Eric Thibodeau (12-2007)                                     &*
6   *& Revisions:     (2007) ET: optimized using blas/ipp/mkl libraries            &*
7   *&                (2008) ET: MPI re-implementation                             &*
8   *&          (28-11-2008) ET: No need to send distc as a vector, we only need &*
9   *&                           the summed distortion? distc[K] becomes distc    &*
10  *&                           Communications are now fused into a single call &*
11  *&=============================================================================&*/
12
13  #include <mpi.h>
14  #include <stdio.h>
15  #include <ctype.h>
16  #include <string.h>
17  #include <stdlib.h>
18  #include <math.h>
19  #include <sys/time.h>
20  //#ifdef USE_BLAS
21      #include "cblas.h"
22  //#endif
23
24  // Don't change this unless you re-adjust the loops manually
25  #define UNROLL_LEVEL 4
26  #define THRESHOLD 0.001
27  #define T   47        /* size of the feature vector */
28
29  #define DEBUG
30
31  long NS_total=-1;   // number of samples (total), derived from DB_size or argv[3]
32  long NS;            // number of samples (local), is NS_total/totalnodes
33  long K;             // K in K-Means, this is argv[2]
34
```

```
35  typedef float sample;   /* Although this might seem convoluted, we can just change this line
36                             to double and all computations now use doubles instead of float */
37  sample *samples;    /* Ptr to table of all training samples */
38  int V_sz = sizeof(sample) * T;  //Vector Size
39  float *centroids;   /* Ptr to table of centroids */
40  float *c_cnt;       /* keeps the count of samples per centroid */
41  float *c_sum;       /* sum of all samples of a class, it is used to update the centroids */
42  int   c_sum_size;   /* Used to supersize c_sum to also contain c_sum + c_cnt + distc*/
43
44  // MPI vars:
45  //MPI_Status status;
46  int mynode;
47  int totalnodes;
48
49
50  /* distance function -
51   * Euclidean Distance is used, which also means it is assumed that the "T" elements in the
52   * multi-dimention vectors are orthogonal, meaning that the information they carry about the
53   * data does not overlap. In a perfectly orthogonal system, if one of the variables of the T
54   * dimention is varied, all other values aren't affected. This is seldom the case in practice
55   * though we try to get as close as possible. It's defintion:
56   *
57   * distance = sqrt((Vect1 - Vect2)^2) <<-- essentially Pythagorean Theorem on a dimention > 2
58   *
59   * */
60  #ifdef USE_BLAS
61  // blas temp vectors:
62  float Vdist[T];
63
64  inline float df(const float *v1, const float *v2){
65      cblas_scopy(T,v1,1,Vdist,1);
66      cblas_saxpy(T,-1.0,v2,1,Vdist,1);
67      return cblas_snrm2(T,Vdist,1);
68  }
69
70  #else
71
72  #ifndef UNROLL
73  //float df(sample *v1, sample *v2)
74  inline float df(const float  *v1, const float *v2)
75  {
76      float sum=0.0;
77      int i;
78
79      // The use of pow(i,2) gives faster code but has little or no
80      // impact when -O3/-O2 is used
81      // We replace sum=sum+(v1[i]-v2[i])*(v1[i]-v2[i]);
82      // with sum+=pow((v1[i]-v2[i]),2);
83      for(i=0; i<T; i++)
84  #ifndef POW
85          sum+=(v1[i]-v2[i])*(v1[i]-v2[i]);
86  #else
87          sum+=pow((v1[i]-v2[i]),2);
88  #endif //POW
89
90      return sqrtf(sum);
91  }
92  #else //UNROLL
93  inline float df(const float  *v1, const float *v2)
94  {
```

```
95      float sum =0.0;
96      float sum1=0.0;
97      float sum2=0.0;
98      float sum3=0.0;
99      int i=0;
100
101     if (T>UNROLL_LEVEL){
102         for(; i<(T-UNROLL_LEVEL); i+=UNROLL_LEVEL){
103             sum +=(v1[i  ]-v2[i  ])*(v1[i  ]-v2[i  ]);
104             sum1+=(v1[i+1]-v2[i+1])*(v1[i+1]-v2[i+1]);
105             sum2+=(v1[i+2]-v2[i+2])*(v1[i+2]-v2[i+2]);
106             sum3+=(v1[i+3]-v2[i+3])*(v1[i+3]-v2[i+3]);
107         }
108     }
109     if (T%UNROLL_LEVEL)
110         for(; i<T; i++)
111             sum+=(v1[i]-v2[i])*(v1[i]-v2[i]);
112
113     sum+=sum1;
114     sum+=sum2;
115     sum+=sum3;
116     return sqrtf(sum);
117 }
118 #endif //UNROLL
119
120 #endif //USE_BLAS
121
122 /* load samples
123 *
124 *   The current loading of samples is technically optimal but not ideal for a
125 *   parallel implementation since the last node might end up with potentially
126 *   no computation. This is less than desireable.
127 **/
128 int load_samples(char *filename)
129 {
130     FILE *fp;
131     int loaded;
132     int n;
133     int n_w=0;  // worker's n count
134     int n_w_adj=0;  // count after adjusting with n % w
135
136     fp=fopen(filename, "r");
137
138     fseek(fp, 0, SEEK_END);
139     n = ftell(fp) / V_sz;
140     fseek(fp, 0, SEEK_SET);
141     if ( (NS_total < 0) || (NS_total > n) )
142         NS_total = n;
143 #ifdef DEBUG
144     if (mynode == 0)
145         printf("Total number of samples = %d\n",NS_total);
146 #endif //DEBUG
147
148     // Normal chunck size
149     n_w = n_w_adj = NS_total/totalnodes;
150     n_w_adj--; // off-by-one: C indices start at 0, otherwise we end up with overlap
151     // The overflow is assigned to the last node (not good if NS_total/totalnodes !>>> totalnodes) :
152     if (mynode == totalnodes-1)
153         n_w_adj = n_w + NS_total % totalnodes;
154
```

```
155     // Checkin logic:
156     //printf("Node %d, Start/End: %d/%d\n", mynode, mynode*n_w, mynode*n_w+n_w_adj);
157     samples=(sample *)malloc(n_w_adj*V_sz);
158     if (!samples) {
159         printf("Memory allocation error while loading DB.\n");
160         goto load_error;
161     }
162
163     /* load samples */
164     // seeking to the chunck this proces will start loading at; and load...
165     fseek(fp, mynode*n_w*V_sz , SEEK_SET);
166     loaded=fread(samples, V_sz, n_w_adj, fp);
167     if (loaded != n_w_adj) {
168         printf("An error occured while loading the DB: ");
169         printf("Loaded %d and expected %d",loaded,n_w_adj);
170         goto mem_error;
171     }
172
173     return loaded;
174
175 load_error:
176     free(samples);
177 mem_error:
178     fclose(fp);
179     return -1;
180 }
181
182 /* centroid initialization - it selects the first set of centroids
183 *
184 *   The original parallel code would search through locals loaded samples
185 *   and selec samples as initial centroids using a module operator * node#.
186 *   We will use the actuall DB and keep the same initialisation as withr the
187 *   sequential code reading the samples from the DB
188 **/
189 void centroid_init(const char *fname)
190 {
191     FILE *fp;
192     int i;
193     int x;
194     // To get the same init as the sequential version:
195     // * V_sz because we're dealing with file pointers (bytes):
196     int skip=(NS_total-1)/K*V_sz;
197
198     fp=fopen(fname, "r");
199
200     for(i=0, x=0; i<K; i++, x+=skip) {
201         fseek(fp, x, SEEK_SET);
202         fread(&centroids[T*i], sizeof(sample), T, fp);
203     //  memcpy((void *) &centroids[T*i], (void *) &samples[T*x], T * sizeof(float));
204     }
205     fclose(fp);
206 }
207
208 /*
209 *   This version of the init is meant to be used if the DB is not stored locally.
210 *   The advantage is that only 1 process does the slow IO and the broadcasts the loaded
211 *   data using an optimized (we hope, MPI implementation dependant) broadcast to all nodes.
212 *   Cost model:
213 *       C_sz = "sizeof(centroids)" = T*K*sizeof(float)
214 *       T_load = BW_io/C_sz + T_broadcast_K
```

```
215  *       T_broadcast_K = T_comm_init + BW_net/C_sz + C_sz/MTU*T_IGU
216  *       T_comma_init=73.1028 (usec) (average latency from hpcc-1.0.0
217  *       MTU=1500 (ethernet)
218  *       T_IGU=15ms (empirical, mpptest)
219  **/
220  void centroid_init_net(const char *fname)
221  {
222      if (mynode == 0)
223          centroid_init(fname);
224      MPI_Bcast(centroids, T*K, MPI_FLOAT, 0, MPI_COMM_WORLD);
225  }
226
227  /* classification of a sample taking into account each centroid */
228  inline int centroid_def(unsigned int ids, float *d)
229  {
230      register int i,centroid=-1;
231      float mdist, dist;
232
233      mdist=999999999.;
234      //mdist=pow(2,32); // we start off very far...
235
236      for(i=0;i<K;i++) {
237          dist=df(&centroids[i*T], &samples[ids*T]);
238          if (dist < mdist) {
239              mdist=dist;
240              centroid=i;
241          }
242      }
243
244      *d=mdist;
245      return centroid;
246  }
247
248  /* it calculates new centroids */
249  void mean_vector()
250  {
251      int i,c, offset;
252
253      for(c=0;c<K;c++) {
254          if (c_cnt[c] != 0){
255              offset=c*T;
256              i=0;
257  #ifdef UNROLL2
258              if (T>UNROLL_LEVEL){
259                  for(; i<(T-UNROLL_LEVEL); i+=UNROLL_LEVEL){
260                      centroids[offset+i  ] = c_sum[offset+i  ]/c_cnt[c];
261                      centroids[offset+i+1] = c_sum[offset+i+1]/c_cnt[c];
262                      centroids[offset+i+2] = c_sum[offset+i+2]/c_cnt[c];
263                      centroids[offset+i+3] = c_sum[offset+i+3]/c_cnt[c];
264                  }
265              }
266              if (T%UNROLL_LEVEL) // Compiler eliminates this if T and UNROLL_LEVEL are static
267  #endif
268                  for(; i<T; i++)
269                      centroids[offset+i  ] = c_sum[offset+i  ]/c_cnt[c];
270          }
271      }
272  }
273
274  /* it calculates the mean distortion */
```

```
275 inline float average_distortion(sample *x)
276 {
277     int i;
278     float ad;
279     ad=0;
280     for (i=0;i<K;i++)
281         ad+=x[i];
282
283     return ad/NS_total;
284 }
285
286 void vq()
287 {
288     int iteration=1;
289     int centr,j,i;
290     float distortion=0;
291     float distortion_ant;
292     //float distc[K];
293     float distc;
294     float dist;
295     int sPos; // used to compute the correct "to next sample" offset
296     int cPos; // used to compute the correct "to next centroid" offset
297     // Custom data type for combined communications:
298
299 /*  if (mynode == 0 )
300         printf("Take it easy, I am classifying...\n");
301 */
302     distortion=0;
303     iteration=1;
304
305     do {
306
307         /* Initialization   */
308         distc=0.;
309         for(i=0;i<K;i++) {
310             //distc[i]=0.;
311             c_cnt[i]=0;
312             for(j=0; j<T;j++)
313                 c_sum[i*T+j]=0.;
314         }
315 //      Using memset is actually longer than the above!
316 /*      memset(distc, 0, sizeof(sample)*K  );
317         memset(c_sum, 0, sizeof(sample)*K*T);
318         memset(c_cnt, 0, sizeof(sample)*K  );*/
319
320         j=0;
321         // The core, we pass the entire DB here:
322         for (j=0; j<NS; j++) {
323
324             centr=centroid_def(j, &dist);
325             //distc[centr]+=dist;
326             distc+=dist;
327             c_cnt[centr]++;
328             // we do the multiplication, out of the loop:
329             // Type of thing a compiler should optimize.
330             sPos=j*T;
331             cPos=centr*T;
332             i=0;
333     #ifdef UNROLL2
334             if (T>UNROLL_LEVEL){
```

```
335                 for(; i<(T-UNROLL_LEVEL); i+=UNROLL_LEVEL){
336                     c_sum[cPos+i  ]+=samples[sPos+i  ];
337                     c_sum[cPos+i+1]+=samples[sPos+i+1];
338                     c_sum[cPos+i+2]+=samples[sPos+i+2];
339                     c_sum[cPos+i+3]+=samples[sPos+i+3];
340                 }
341             }
342             if (T%UNROLL_LEVEL) // Compiler eliminates this if T and UNROLL_LEVEL are static
343     #endif
344                 for(; i<T; i++)
345                     c_sum[cPos+i]+=samples[sPos+i];
346         }
347
348         // Put distc into end of c_sum 'mega-vector' (offset from malloc)
349         c_sum[c_sum_size-1] = distc;
350         MPI_Allreduce(MPI_IN_PLACE, c_sum, c_sum_size,  MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
351         distc = c_sum[c_sum_size-1];
352         /*  After this point, all vars are in the same state as if they had been computed
353             by a single process
354         */
355         mean_vector();
356
357         distortion_ant=distortion;
358         //distortion=average_distortion(distc);
359         distortion=distc/NS_total;
360
361         iteration++;
362 #ifdef DEBUG
363         if (mynode == 0)
364             printf("TWO_LAST_AVERAGE_DISTORTIONS: AD1=%f, AD2=%f, Dif=%f\n", \
365             distortion_ant, distortion, fabs((double) (distortion_ant - distortion)));
366 #endif
367     } while (fabs((double)(distortion_ant - distortion)) > THRESHOLD);
368 }
369
370 /* show centroids */
371 void show_centroids()
372 {
373   int i,j;
374
375   printf("Centroids:\n");
376   for(i=0;i<K;i++) {
377     for(j=0;j<T;j++)
378       if(i == 0 || i==(K-1)) printf("%2.2f ", centroids[i*T+j]);
379
380     if(i == 0 || i==(K-1)) printf("\n");
381   }
382 }
383
384 /* show samples */
385 void show_samples()
386 {
387     int i, j;
388     for (i=0;i<NS;i++) {
389         for (j=0;j<T;j++)
390             printf("%f ", samples[NS*T+i]);
391         printf(" c=N/A\n");
392     }
393 }
394
```

```
395 /* save centroids */
396 void save_centroids(const char* outname)
397 {
398     int i,j;
399     FILE *fp;
400
401     fp=fopen(outname, "wb");
402     /*printf("Saving centroids \n");*/
403     for(i=0;i<K;i++) {
404         for(j=0;j<T;j++)
405             fprintf(fp, "%f,", centroids[i*T+j]);
406         fprintf(fp, "\n");
407     }
408     fclose(fp);
409 }
410
411
412 /* main */
413 int main(int argc, char *argv[])
414 {
415     char   *fnamein;
416     struct timeval tempo1, tempo2;
417     struct timezone tzp;
418     double tempo;
419
420     MPI_Init(&argc, &argv);
421     MPI_Comm_size(MPI_COMM_WORLD, &totalnodes);
422     MPI_Comm_rank(MPI_COMM_WORLD, &mynode);
423
424     fnamein=argv[1];
425     K = atoi(argv[2]);
426     if (argc > 3){
427         NS_total=atoi(argv[3]);
428         if (mynode == 0)
429             printf("Limiting sample load to %d samples.\n",NS_total);
430     }
431
432     gettimeofday(&tempo1,&tzp);
433
434     // we merge c_sum, c_cnt and dist into a single vector to simplify communication consolidations
435     c_sum_size  = (K + T + K + 1 );
436     c_sum       = (float *) malloc(c_sum_size * sizeof(sample));
437     c_cnt       = &c_sum[K*T]; // beyond last element of c_sum is start of c_cnt
438     //c_sum     = malloc(K * sizeof(sample)*T );
439     //c_cnt     = malloc(K * sizeof(sample)   );
440     centroids   = (float *) malloc(K * sizeof(sample)*T );
441
442 #ifdef DEBUG
443     if(mynode==0)
444         printf("K:%d\tT:%d\nc_sum_size:%d\n(c_cnt-c_sum):%d\n",K,T,c_sum_size,(c_cnt - c_sum));
445 #endif
446
447     if (!c_sum || !centroids || !c_cnt ) {
448          printf("malloc failure on c_sum || centroids || c_cnt!\n");
449          exit (1);
450     }
451
452     /* load samples */
453
454     NS=load_samples(fnamein);
```

```
455     if (NS==-1) {
456         printf("\nNode_%d:_ERROR_=_loading_sample_file\n",mynode);
457         MPI_Finalize();
458         exit(1);
459     }
460 #ifdef DEBUG
461     printf("NS_=_%lu\n", NS);
462 #endif
463 
464     /* centroids initialization */
465     centroid_init(fnamein);
466     //centroid_init_net(fnamein);
467 
468     /* vector quantisation */
469     vq();
470 
471     // show_centroids();
472 
473 #ifdef DEBUG
474     gettimeofday(&tempo2,&tzp);
475     tempo = (double)(tempo2.tv_sec - tempo1.tv_sec) + (((double)(tempo2.tv_usec-tempo1.tv_usec))/1000000);
476 
477     printf("Total_time_for_node_%d_(s):_%.3f\n", mynode, tempo);
478 #endif
479     if (mynode == 0)
480         save_centroids("centroids");
481 
482     MPI_Finalize();
483     return 0;
484 }
```

BIBLIOGRAPHY

[1] Allan, Benjamin A. et Robert Armstrong et al.: *A Component Architecture for High-Performance Scientific Computing*. International Journal of High Performance Computing Applications, 20(2) :163–202, 2006, ISSN 1094-3420.

[2] Bell, R., A.D. Malony et S. Shende: *ParaProf : A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis*. LECTURE NOTES IN COMPUTER SCIENCE, pages 17–26, 2003.

[3] Browne, S., J. Dongarra, N. Garner, G. Ho et P. Mucci: *A Portable Programming Interface for Performance Evaluation on Modern Processors*. International Journal of High Performance Computing Applications, 14(3) :189–204, 2000. http://hpc.sagepub.com/cgi/content/abstract/14/3/189.

[4] Bruck, J., Ching Tien Ho, S. Kipnis, E. Upfal et D. Weathersby: *Efficient algorithms for all-to-all communications in multiport message-passing systems*. Parallel and Distributed Systems, IEEE Transactions on, 8(11) :1143–1156, Nov 1997, ISSN 1045-9219.

[5] Castain, R.H., T.S. Woodall, D.J. Daniel, J.M. Squyres, B. Barrett et G.E. Fagg: *The Open Run-Time Environment (OpenRTE) : A transparent multicluster environment for high-performance computing*. Tome 24, pages 153–157, 2008. http://www.sciencedirect.com/science/article/B6V06-4NH7DWP-1/2/25a63048395769febd13ce3f2a9545b8.

[6] Chang, J., Ming Huang, J. Shoemaker, J. Benoit, Szu Liang Chen, Wei Chen, Siufu Chiu, R. Ganesan, G. Leong, V. Lukka, S. Rusu et D. Srivastava: *The 65-nm 16-MB Shared On-Die L3 Cache for the Dual-Core Intel Xeon Processor 7100 Series*. Solid-State Circuits, IEEE Journal of, 42(4) :846–852, April 2007, ISSN 0018-9200.

[7] Che, Shuai, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer et Kevin Skadron: *A performance study of general-purpose applications on graphics processors using CUDA*. J. Parallel Distrib. Comput., 68(10) :1370–1380, 2008, ISSN 0743-7315.

[8] Desnoyers, M. et M. Dagenais: *Low disturbance embedded system tracing with linux trace toolkit next generation*. rapport technique, Ecole Polytechnique de Montréal, 2006. http://ltt.polymtl.ca/files/papers/celf2006-desnoyers.pdf.

[9] Dongarra, J., K. London, S. Moore, P. Mucci et D Terpstra: *Using PAPI for Hardware Performance Monitoring on Linux Systems*. Dans *Conference on Linux Clusters : The HPC Revolution*, page 11. National Center for Supercomputing Applications (NCSA), University of Illinois, June 2001.

[10] Dongarra, J., A.D. Malony, S. Moore, P. Mucci et S. Shende: *Performance Instrumentation and Measurement for Terascale Systems*. LECTURE NOTES IN COMPUTER SCIENCE, pages 53–62, 2003.

[11] Dongarra, Jack: *The Impact of Multicore on Math Software and Exploiting Single Precision Computing to Obtain Double Precision Results*. Parallel Processing, 2006. ICPP 2006. International Conference on, page 19, Aug. 2006, ISSN 0190-3918.

[12] Dongarra, Jack, Kevin London, Shirley Moore, Philip Mucci, Daniel Terpstra, Haihang You et Min Zhou: *Experiences and Lessons Learned with a Portable Interface to Hardware Performance Counters*. Parallel and Distributed Processing Symposium, International, 0 :6, 2003, ISSN 1530-2075.

[13] Forman, George et Bin Zhang: *Linear speed-up for a parallel non-approximate recasting of center-based clustering algorithms, including K-Means, K-Harmonic means, and EM*. Rapport technique 93, HP Laboratories, 2000. Mean square error (MSE) ; K-harmonic means (KHM) ; Expectation-maximization (EM) ; Multidimensional data clustering ; Center-based clustering.

[14] Foster, Ian: *Designing and Building Parallel Programs*. Addison-Wesley Publishing Co., février 1995, ISBN 0201575949.

[15] Franchetti, F., S. Kral, J. Lorenz et C.W. Ueberhuber: *Efficient Utilization of SIMD Extensions*. Proceedings of the IEEE, 93(2) :409–425, Feb. 2005, ISSN 0018-9219.

[16] Garcia, V., E. Debreuve et M. Barlaud: *Fast k nearest neighbor search using GPU*. Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on, pages 1–6, June 2008.

[17] Garg, P.: *Investigating coverage-reliability relationship and sensitivity of reliability to errors in the operational profile*. Dans *Software Testing, Reliability and Quality Assurance*, pages 21–35, Dec 1994.

[18] Gleixner, T. et D. Niehaus: *Hrtimers and Beyond : Transforming the Linux Time Subsystems*. Dans *Proceedings of the Ottawa Linux Symposium*, tome 1, page 16, juillet 2006.

[19] GOEDEKER, Adolfy HOISIE Stephan: *Performance Optimization of Numerically Intensive Code*. Siam, 2001, ISBN 0-89871-484-2.

[20] Goto, Kazushige et Robert A. van de Geijn: *Anatomy of high-performance matrix multiplication*. ACM Trans. Math. Softw., 34(3) :1–25, 2008, ISSN 0098-3500.

[21] Graham, Susan L., Peter B. Kessler et Marshall K. Mckusick: *Gprof : A call graph execution profiler*. Dans *SIGPLAN '82 : Proceedings of the 1982 SIGPLAN symposium*

on Compiler construction, pages 120–126, New York, NY, USA, 1982. ACM, ISBN 0-89791-074-5.

[22] Gropp, W. et E. Lusk: *Reproducible measurements of MPI performance characteristics.* DuroPVM/MPI'99, septembre 1999.

[23] Hennessy, John L. et David A. Patterson: *Computer Architecture : A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Denise E. M. Penrose (The Morgan Kaufmann Series in Computer Architecture and Design), fourth edition édition, May 2007, ISBN 13 : 978-0-12-370490-0 10 : 0-12-370490-1. http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20\&path=ASIN/1558605967.

[24] Hofstee, Peter et Michael Day: *Hardware and software architectures for the CELL processor*. Hardware/Software Codesign and System Synthesis, 2005. CODES+ISSS '05. Third IEEE/ACM/IFIP International Conference on, pages 1–1, Sept. 2005.

[25] Huang, JC et T. Leng: *Generalized loop-unrolling : a method for program speedup*. Dans *1999 IEEE Symposium on Application-Specific Systems and Software Engineering and Technology, 1999. ASSET'99. Proceedings*, pages 244–248, 1999.

[26] Huck, Kevin A. et Allen D. Malony: *PerfExplorer : A Performance Data Mining Framework For Large-Scale Parallel Computing*. Dans *SC '05 : Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 41, Washington, DC, USA, 2005. IEEE Computer Society, ISBN 1-59593-061-2.

[27] Huck, Kevin A., Allen D. Malony, Robert Bell et Alan Morris: *Design and Implementation of a Parallel Performance Data Management Framework*. Dans *ICPP '05 : Proceedings of the 2005 International Conference on Parallel Processing*, pages 473–482, 2005. http://csdl2.computer.org/persagen/DLAbsToc.jsp?resourcePath=/dl/proceedings/\&toc=comp/proceedings/icpp/2005/2380/00/2380toc.xml\&DOI=10.1109/ICPP.2005.29.

[28] Hughes, P. et B. Conway: *The AMD Opteron Northbridge Architecture*. IEEE Micro, 27(2) :10–21, March-April 2007, ISSN 0272-1732.

[29] K. Huck, A. Malony S. Shende et A. Morris.: *TAUg : Runtime Global Performance Data Access using MPI*. Dans Springer (rédacteur) : *EuroPVM/MPI Conference*, numéro LNCS 4192, pages 313–321, September 2006.

[30] Kahle, J.: *The Cell Processor Architecture*. Microarchitecture, 2005. MICRO-38. Proceedings. 38th Annual IEEE/ACM International Symposium on, pages 3–3, 12-16 Nov. 2005.

[31] Kaspersky, Kris: *Code Optimizatoin : Effective Memory Usage*. A-LIST, 295 East Swedesford Rd., 2003, ISBN 1-931769-24-9.

[32] Keltcher, Chetana N., Kevin J. McGrath, Ardsher Ahmed et Pat Conway: *The AMD Opteron Processor for Multiprocessor Servers*. IEEE Micro, 23(2) :66–76, 2003, ISSN 0272-1732.

[33] Kufrin, R.: *PerfSuite : An Accessible, Open Source Performance Analysis Environment for Linux*. Dans *Presented at The 6th International Conference on Linux Clusters : The HPC Revolution*, tome 151, page 05, 2005.

[34] Lindlan, K.A. ; Cuny J. ; Malony A.D. ; Shende S. ; Mohr B. ; Rivenburgh R. ; Rasmussen C.: *A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates*. Supercomputing, ACM/IEEE 2000 Conference, pages 49–49, Nov 2000, ISSN 1063-9535.

[35] Luszczek, P., J.J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. McCalpin, D. Bailey et D. Takahashi: *Introduction to the HPC Challenge Benchmark Suite*, avril 2005.
`http://repositories.cdlib.org/lbnl/LBNL-57493`.

[36] Moore, M., R. Wisniewski, R. Yaghmour, K. Zanussi et T. Dagenais: *Efficient and Accurate Tracing of Events in Linux Clusters*. rapport technique, Ecole Polytechnique de Montreal, May 11-14 2003.

[37] Moore, S., D. Cronk, F. Wolf, A. Purkayastha, P. Teller, R. Araiza, M.G. Aguilera et J. Nava: *Performance Profiling and Analysis of DoD Applications Using PAPI and TAU*. Users Group Conference, (10.1109/DODUGC.2005.50) :394–399, 2005, ISSN 10.1109/DODUGC.2005.50.

[38] Mucci, P.J., S. Browne, C. Deane et G. Ho: *PAPI : A Portable Interface to Hardware Performance Counters*. Dans *Proc. Dept. of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[39] Mueller, S.M. ; Jacobi C. ; Oh H. J. ; Tran K.D. ; Cottier S.R. ; Michael B.W. ; Nishikawa H. ; Totsuka Y. ; Namatame T. ; Yano N. ; Machida T. ; Dhong S.H.: *The vector floating-point unit in a synergistic processor element of a CELL processor*. Computer Arithmetic, 2005. ARITH-17 2005. 17th IEEE Symposium on, pages 59–67, 27-29 June 2005, ISSN 1063-6889.

[40] Munson, J.C.: *A software blackbox recorder*. Dans *Aerospace Applications Conference, 1996. Proceedings., 1996 IEEE*, tome 4, pages 309–320 vol.4, Feb 1996.

[41] Nataraj, Aroon, Alan Morris, Allen D. Malony, Matthew Sottile et Pete Beckman: *The Ghost in the Machine : Observing the Effects of Kernel Operation on Parallel*

Application Performance. Dans *SC '07 : Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–12, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-764-3.

[42] Ridge, D., D. Becker, P. Merkey et T. Sterling: *Beowulf : harnessing the power of parallelism in a pile-of-PCs*. Aerospace Conference, 1997. Proceedings., IEEE, 2 :79–91 vol.2, Feb 1997.

[43] S. Britto Jr. Alceu de, Paulo S. L. de Souza, Robert Sabourin, Simone R. S. de Souza et Dîbio L. Bogres: *A Low-Cost Parallel K-Means Algorithm Using Cluster Computing*. rapport technique, École de Technologie Supérieure, août 2003.

[44] S. Britto Jr. Alceu de, Robert Sabourin, Bortolozzi F. et Suen C.Y: *Recognition of Handwritten Numeral Strings Using a Two-Stage Hmm-Based Method*. septembre 2003.

[45] Shahbahrami, A., B. Juurlink et S. Vassiliadis: *Performance Impact of Misaligned Accesses in SIMD Extensions*. Dans *Proc. 17th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC2006), Veldhoven, The Netherlands, November*, pages 23–24, 2006.

[46] Shende, Sameer S. et Allen D. Malony: *The Tau Parallel Performance System*. International Journal of High Performance Computing Applications, 20(2) :287–311, 2006.
`http://hpc.sagepub.com/cgi/content/abstract/20/2/287`.

[47] Slogsnat, David, Alexander Giese et Ulrich Brüning: *A versatile, low latency HyperTransport core*. Dans *FPGA '07 : Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*, pages 45–52, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-600-4.

[48] Sottile, Matthew Joseph: *A measurement and simulation methodology for parallel computing performance studies*. Thèse de doctorat, University of Oregon, Albuquerque, NM, USA, 2006. ISBN 978-0-542-73568-4. Adviser-David A. Bader.

[49] Spear, W., A. Malony, A. Morris et S. Shende: *Integrating TAU with Eclipse : A Performance Analysis System in an Integrated Development Environment*. LECTURE NOTES IN COMPUTER SCIENCE, 4208 :230, 2006.

[50] Stallings, William: *Computer Organization and Architecture : designing for performance*. Alan Apt (Prentice-Hall Inc.), fifth edition édition, 2000. ISBN 0-13-081294-3.

[51] Stultz, J., N. Aravamudan et D. Hart: *We Are Not Getting Any Younger : A New Approach to Time and Timers*. Dans *Linux Symposium*, pages 219–232, 2005.

[52] Tam, S., S. Rusu, J. Chang, S. Vora, B. Cherkauer et D. Ayers: *A 65nm 95W Dual-Core Multi-Threaded XeonÂ® Processor with L3 Cache*. Solid-State Circuits Conference, 2006. ASSCC 2006. IEEE Asian, pages 15–18, Nov. 2006.

[53] Turner, D. et X. Chen: *Protocol-dependent message-passing performance on linux clusters*. Dans *Proceedings of the IEEE International Conference on Cluster Computing.*, pages 187–194, septembre 2002.

[54] Wadleigh, Kevin R. et Isom L. Crawford: *Software Optimization for High Performance Computing*. Helwett-Packard Professinal Books. Prentice Hall PTR, 2000, ISBN 0-13-017008-9.

[55] Weaver, V.M. et S.A. McKee: *Can hardware performance counters be trusted*. Dans *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 141–150, 2008.

[56] Worringen, J.: *Pipelining and overlapping for MPI collective operations*. Dans *Local Computer Networks, 2003. LCN '03. Proceedings. 28th Annual IEEE International Conference on*, pages 548–557, octobre 2003.

[57] Zhang, Bin, Meichun Hsu et George Forman: *Accurate Recasting of Parameter Estimation Algorithms Using Sufficient Statistics for Efficient Parallel Speed-Up Demonstrated for Center-Based Data Clustering Algorithms*. rapport technique, HP Laboratories Palo Alto, juillet 2000.

[58] Zhang, Yufang, Zhongyang Xiong, Jiali Mao et Ling Ou: *The Study of Parallel K-Means Algorithm*. Intelligent Control and Automation, 2006. WCICA 2006. The Sixth World Congress on, 2 :5868–5871, 2006.