Universidad Nacional de La Plata
Facultad de Informática

Especialización en Cómputo de Altas Prestaciones y Tecnología Grid

Herramientas para el Soporte de Análisis de Rendimiento

Alumno: Andrés More - amore@hal.famaf.unc.edu.ar
Director: Dr Fernando G. Tinetti - fernando@lidi.info.unlp.edu.ar

Agosto de 2013

Resumen

Este documento describe una investigación realizada como trabajo final para la Especialización en Cómputo de Altas Prestaciones dictada en la Facultad de Informática de la Universidad Nacional de La Plata. El tema de investigación consiste en métodos y herramientas para el análisis del comportamiento de aplicaciones de alto rendimiento.

Este trabajo contribuye con un resumen de la teoría de análisis de rendimiento más una descripción de las herramientas de soporte disponibles en el momento. Se propone también un proceso para analizar el rendimiento, ejemplificando su aplicación a un conjunto de núcleos de cómputo no triviales.

Luego de la introducción de terminología y bases teóricas del análisis cuantitativo de rendimiento, se detalla la experiencia de utilizar herramientas para conocer dónde se deberían localizar los esfuerzos de optimización. Este trabajo resume la experiencia que debe atravesar cualquier investigador en busca de las diferentes alternativas para el análisis de rendimiento; incluyendo la selección de herramientas de soporte y la definición de un procedimiento sistemático de optimización.

Índice general

1. Introducción **3**
- 1.1. Motivación . . . 3
- 1.2. Alcance . . . 3
- 1.3. Estado Actual . . . 4
- 1.4. Organización del Contenido . . . 4

2. Análisis de Rendimiento **5**
- 2.1. Definición . . . 5
- 2.2. Paralelismo . . . 5
  - 2.2.1. Ley de *Amdahl* . . . 5
  - 2.2.2. Ley de *Gustafson* . . . 6
  - 2.2.3. Métrica de *Karp-Flatt* . . . 7
- 2.3. Métricas . . . 8
- 2.4. Técnicas de Análisis . . . 8

3. Herramientas de Soporte **10**
- 3.1. Pruebas de Rendimiento . . . 10
  - 3.1.1. STREAM . . . 11
  - 3.1.2. Linpack . . . 11
  - 3.1.3. Intel MPI Benchmarks . . . 13
  - 3.1.4. HPC Challenge . . . 14
- 3.2. Herramientas . . . 16
- 3.3. Tiempo de Ejecución . . . 17
  - 3.3.1. Tiempo de ejecución global . . . 17
  - 3.3.2. Reloj del sistema . . . 18
  - 3.3.3. Ejemplo: `time` . . . 18
- 3.4. Perfil de Ejecución Funcional . . . 19
  - 3.4.1. Ejemplo: `gprof` . . . 19
- 3.5. Perfil de Ejecución asistido por *Hardware* . . . 20
  - 3.5.1. Ejemplo: `perf` . . . 21
- 3.6. Reporte de Vectorización . . . 22

4. Procedimiento de Análisis **23**
- 4.1. Proceso . . . 23
- 4.2. Paso a Paso . . . 24
  - 4.2.1. Pruebas de Referencia . . . 24

5. Casos de Estudio **26**
5.1. Rendimiento de Referencia 26
5.2. Multiplicación de Matrices 27
5.3. Distribución de Calor en Dos Dimensiones 29
5.4. Problema de las Ocho Reinas 32

6. Conclusiones **35**

Capítulo 1

Introducción

Este capítulo introduce este trabajo y su alcance. Luego de revisar la motivación de esta investigación y su alcance, se resume el estado actual del análisis del rendimiento y se detalla el contenido restante del informe.

1.1. Motivación

En el área de cómputo de altas prestaciones los desarrolladores son los mismos especialistas del dominio del problema a resolver. Las rutinas más demandantes de cálculo son en su mayoría científicas y su alta complejidad hace posible su correcta implementación sólo por los mismos investigadores. Estas cuestiones resultan en un tiempo reducido de análisis de resultados e impactan directamente en la productividad de los grupos de investigación y desarrollo.

Con mayor impacto que en otras áreas de la computación, el código optimizado correctamente puede ejecutarse órdenes de magnitud mejor que una implementación directa [1]. Además, se utiliza programación en paralelo para obtener una mejor utilización de la capacidad de cómputo disponible; aumentando por lo tanto la complejidad de implementación, depuración y optimización [2].

Frecuentemente el proceso de optimización termina siendo hecho de modo *ad-hoc*, sin conocimiento pleno de las herramientas disponibles y sus capacidades, y sin la utilización de información cuantitativa para dirigir los esfuerzos de optimización. Es incluso frecuente la implementación directa de algoritmos en lugar de la utilización de librerías ya disponibles, optimizadas profundamente y con correctitud comprobada.

1.2. Alcance

Este trabajo considera en particular a los sistemas de memoria distribuida corriendo sobre *software* libre GNU/Linux, inicialmente denominados sistemas Beowulf [3]. A través de las estadísticas mostradas por el Top500 [1], se puede

[1] El Top500 es una lista actualizada de super-computadoras de acuerdo al benchmark *Linpack*, disponible en `http://www.top500.org`.

determinar que son los más utilizados en el cómputo de aplicaciones de alto rendimiento en la actualidad.

1.3. Estado Actual

Actualmente existen numerosas y diversas herramientas para el análisis de rendimiento [4]. Estas funcionan a diferentes niveles de abstracción: desde contadores de eventos a nivel de *hardware*, pasando por monitores de recursos dentro del núcleo del sistema operativo, instrumentación de código, y hasta la simple utilización del tiempo de ejecución de una aplicación o la comparación contra un trabajo similar de referencia. Para poder analizar del rendimiento un desarrollador aplica las herramientas listadas en la tabla 1.1.

Tabla 1.1: Herramientas de Soporte para Optimización

Herramienta	**Descripción**
`gprof`	Muestra información de perfil de llamadas a funciones
`perf`	Muestra información de perfil de sistema
STREAM	Benchmark de jerarquía de memoria
HPL	Benchmark de capacidad de cómputo
IMB Ping Pong	Benchmark de latencia y ancho de banda de red
HPCC	Paquete de benchmarks

Las pruebas de rendimiento como STREAM, HPL, IMB Ping Pong y HPCC permiten conocer el estado del sistema y los límites de rendimiento en la práctica. Los perfiles de ejecución de aplicaciones y sistema permiten conocer como se utilizan los recursos del sistema para llevar a cabo las instrucciones de cada programa.

1.4. Organización del Contenido

El siguiente capítulo discute el análisis de rendimiento, sus principios y teoría básica. El capítulo 3 detalla las herramientas más utilizadas. El capítulo 4 aplica las herramientas revisadas a través de un proceso sistemático. El capítulo 5 ejemplifica la aplicación del proceso con algunos núcleos de computo. El capítulo 6 concluye y detalla posibles extensiones de este trabajo.

Capítulo 2

Análisis de Rendimiento

Este capítulo introduce el concepto de rendimiento y teoría básica sobre su análisis. Además ejemplifica las actividades a realizar durante el análisis.

2.1. Definición

El rendimiento se caracteriza por la cantidad de trabajo de cómputo que se logra en comparación con la cantidad de tiempo y los recursos ocupados. El rendimiento debe ser evaluado entonces de forma cuantificable, utilizando alguna métrica en particular de modo de poder comparar relativamente dos sistemas o el comportamiento de un mismo sistema bajo una configuración distinta.

2.2. Paralelismo

Una vez obtenida una implementación eficiente, la única alternativa para mejorar el rendimiento es explotar el paralelismo que ofrecen los sistemas de cómputo. Este paralelismo se puede explotar a diferentes niveles, desde instrucciones especiales que ejecutan sobre varios datos a la vez (vectorización), hasta la utilización de múltiples sistemas para distribuir el trabajo.

El cálculo de las mejoras posibles de rendimiento, cómo priorizarlas y la estimación de su límite máximo es una tarea compleja. Para ello existen algunas leyes fundamentales utilizadas durante el análisis de rendimiento.

2.2.1. Ley de *Amdahl*

La ley de *Amdahl* [5] dimensiona la mejora que puede obtenerse en un sistema de acuerdo a las mejoras logradas en sus componentes. Nos ayuda a establecer un límite máximo de mejora y a estimar cuales pueden ser los resultados de una optimización.

La mejora de un programa utilizando cómputo paralelo está limitado por el tiempo necesario para completar su fracción serial o secuencial. En la mayoría de los casos, el paralelismo sólo impacta notoriamente cuando es utilizado en un pequeño número de procesadores, o cuando se aplica a problemas altamente

escalables (*Embarrassingly Parallel Problems*). Una vez paralelizado un programa, los esfuerzos suelen ser enfocados en cómo minimizar la parte secuencial, algunas veces haciendo más trabajo redundante pero en forma paralela.

Suponiendo que una aplicación requiere de un trabajo serial más un trabajo paralelizable, la ley de *Amdahl* calcula la ganancia (S) mediante la Ecuación 2.1. Donde P es el porcentaje de trabajo hecho en paralelo, $(1 - P)$ es entonces el trabajo en serie o secuencial, y N la cantidad de unidades de cómputo a utilizar.

$$S = \frac{1}{(1 - P) + \frac{P}{N}} \tag{2.1}$$

Esta ley nos establece que incluso teniendo infinitas unidades de cómputo la ganancia está limitada. La Tabla 2.1 muestra que no importa la cantidad de unidades de procesamiento que sean utilizadas.

Tabla 2.1: Mejora Máxima

		Porcentaje de Paralelismo					
		0.1	0.3	0.5	0.8	0.9	0.95
Número de Procesadores	1	1.00	1.00	1.00	1.00	1.00	1.00
	2	1.05	1.14	1.33	1.60	1.82	1.90
	4	1.08	1.23	1.60	2.29	3.08	3.48
	8	1.10	1.28	1.78	2.91	4.71	5.93
	16	1.10	1.31	1.88	3.37	6.40	9.14
	32	1.11	1.32	1.94	3.66	7.80	12.55
	64	1.11	1.33	1.97	3.82	8.77	15.42
	128	1.11	1.33	1.98	3.91	9.34	17.41
	256	1.11	1.33	1.99	3.95	9.66	18.62
	512	1.11	1.33	2.00	3.98	9.83	19.28
	1024	1.11	1.33	2.00	3.99	9.91	19.64
	2048	1.11	1.33	2.00	3.99	9.96	19.82
	4096	1.11	1.33	2.00	4.00	9.98	19.91
	8192	1.11	1.33	2.00	4.00	9.99	19.95
	16384	1.11	1.33	2.00	4.00	9.99	19.98
	32768	1.11	1.33	2.00	4.00	10.00	19.99
	65536	1.11	1.33	2.00	4.00	10.00	19.99

Por ejemplo, en el caso de tener solo un 10 % de paralelismo en una aplicación, la mejora nunca va a superar $1{,}10x$ la original. En el caso de tener un 95 %, la mejora no puede ser mayor a $20x$.

2.2.2. Ley de *Gustafson*

Desde un punto de vista más general, la ley de *Gustafson* [6] (computada mediante la Ecuación 2.2) establece que las aplicaciones que manejan problemas repetitivos con conjuntos de datos similares pueden ser fácilmente paralelizadas. En comparación, la ley anterior no escala el tamaño o resolución de problema cuando se incrementa la potencia de cálculo, es decir asume un tamaño de problema fijo.

$$speedup(P) = P - \alpha \times (P - 1) \tag{2.2}$$

donde P es el número de unidades de cómputo y α el porcentaje de trabajo paralelizable.

Al aplicar esta ley obtenemos que un problema con datos grandes o repetitivos en cantidades grandes puede ser computado en paralelo muy eficientemente. Nos es útil para determinar el tamaño de problema a utilizar cuando los recursos de cómputo son incrementados. En el mismo tiempo de ejecución, el programa resuelve entonces problemas más grandes.

Tabla 2.2: Tamaño de Datos de Entrada

		Porcentaje de Paralelismo					
		0.1	0.25	0.5	0.75	0.9	0.95
Número de Procesadores	1	1	1	1	1	1	1
	2	1	1	2	2	2	2
	4	1	2	3	3	4	4
	8	2	3	5	6	7	8
	16	3	5	9	12	15	15
	32	4	9	17	24	29	30
	64	7	17	33	48	58	61
	128	14	33	65	96	115	122
	256	27	65	129	192	231	243
	512	52	129	257	384	461	486
	1024	103	257	513	768	922	973
	2048	206	513	1025	1536	1843	1946
	4096	411	1025	2049	3072	3687	3891
	8192	820	2049	4097	6144	7373	7782
	16384	1639	4097	8193	12288	14746	15565
	32768	3278	8193	16385	24576	29491	31130
	65536	6555	16385	32769	49152	58983	62259

Similarmente al cuadro anterior, podemos deducir de la Tabla 2.2 que en el caso de un programa con sólo 10 % de paralelismo, al incrementar los recursos 64x sólo podemos incrementar el tamaño del problema $7x$. Sin embargo nos calcula un incremento de $61x$ en el caso de tener 95 % de paralelismo.

2.2.3. Métrica de *Karp-Flatt*

Esta métrica es utilizada para medir el grado de paralelismo de una aplicación [7]. Esta métrica nos permite rápidamente estimar la mejora posible al aplicar un alto nivel de paralelismo.

Dado un cómputo paralelo con una mejora de rendimiento ψ en P procesadores, donde $P > 1$. La fracción serial *Karp-Flatt* representada con e y calculada según la Ecuación 2.3 es determinada experimentalmente, mientras menor sea e mayor se supone el nivel de paralelismo posible.

$$e = \frac{\frac{1}{\psi} - \frac{1}{p}}{1 - \frac{1}{p}} \tag{2.3}$$

Para un problema de tamaño fijo, la eficiencia típicamente disminuye cuando el número de procesadores aumenta. Se puede entonces determinar si esta disminución es debida a un paralelismo limitado, a un algoritmo no optimizado o un problema de arquitectura del sistema.

2.3. Métricas

Algunos ejemplos de medida de rendimiento son:

1. El ancho de banda y la latencia mínima de un canal de comunicación, una jerarquía de memorias o de la unidad de almacenamiento.
2. La cantidad de instrucciones, operaciones, datos o trabajo procesado por cierta unidad de tiempo.
3. El rendimiento asociado al costo del equipamiento, incluyendo mantenimiento periódico, personal dedicado y gastos propios del uso cotidiano.
4. El rendimiento por unidad de energía consumida (electricidad).

Un método de medición de rendimiento indirecto consiste en medir el uso de los recursos del sistema mientras se ejercita el mismo con un trabajo dado. Por ejemplo: el nivel de carga de trabajo en el sistema, la cantidad de operaciones realizadas por el sistema operativo o la unidad de procesamiento, la utilización de memoria o archivos temporales e incluso el ancho de banda de red utilizado durante la comunicación.

2.4. Técnicas de Análisis

El procedimiento de mejora general usualmente consiste en ciclos iterativos de medir, localizar, optimizar y comparar (Figura 2.1). Es muy importante mantener la disciplina en realizar un cambio a la vez ya que esto asegura resultados reproducibles y convergentes, sin efectos no deseados.

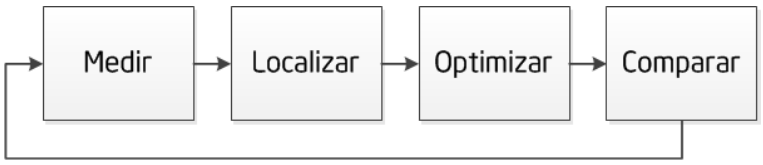

Figura 2.1: Optimización Iterativa

A la hora de tomar decisiones, éstas deben estar basadas en datos concretos, ya que en caso contrario se podría estar trabajando sin llegar a obtener un rédito adecuado.

En el caso de tener problemas de desviación en los resultados medidos, es aconsejable obtener un gran número de muestras y utilizar un valor promedio para asegurarse de evitar errores de medición tanto como sea posible. También es preferible aumentar el tamaño del problema a resolver, o la definición de los resultados para ejercitar por más tiempo y tener así un resultado más estable.

Suponiendo una distribución normal de resultados, se suelen controlar que haya menos de 3 σ de diferencia. Se busca que la mayoría de los resultados queden cerca de su promedio, como muestra la Figura 2.2.

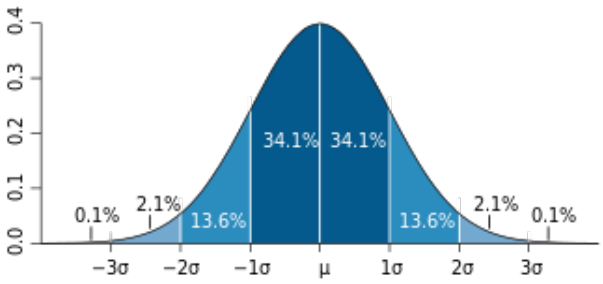

Figura 2.2: Desviación de valores en una distribución normal [Wikipedia]

Los resultados deben también ser correctamente guardados para evitar problemas de datos. Si la configuración del sistema es dinámica entonces la reproducción de resultados es no trivial. En el caso de no tener una configuración de sistema estable en el tiempo, es recomendable siempre ejecutar una versión optimizada contra una versión de referencia en un mismo sistema de cómputo.

Para comparar es recomendable utilizar la media geométrica según la Ecuación 2.4 en lugar de la aritmética [8], ya que permite dimensionar la tendencia central de un valor típico en un conjunto de números. Esto permite reducir el impacto de ruido introducido por una ejecución problemática.

$$G = \sqrt[n]{x_1 \dots x_n} \tag{2.4}$$

La raíz n-ésima de un número (para un n posiblemente muy grande), es una operación ineficiente ya que se implementa con métodos numéricos de aproximación siguiendo el método de *Newton* [9]. En cambio se suele tomar el antilogaritmo del promedio de los logaritmos de los valores siguiendo la ecuación 2.5.

$$G = 10^{(log_{10}(x_1)+\dots+log_{10}(x_n))/n} \tag{2.5}$$

Capítulo 3

Herramientas de Soporte

Este capítulo revisa las herramientas disponibles para soporte de análisis de rendimiento de aplicaciones.

3.1. Pruebas de Rendimiento

Para medir el rendimiento se utilizan pruebas de referencia (*benchmarks*); éstas pueden ser aplicaciones sintéticas construidas específicamente, o bien aplicaciones del mundo real computando un problema prefijado. Al tener valores de referencia se pueden caracterizar los sistemas de modo de predecir el rendimiento de una aplicación. Los valores a los que se llegan con un *benchmark* suelen ser más prácticos y comparables que los teóricos de acuerdo a condiciones ideales de uso de recursos. También es posible garantizar que el sistema sigue en un mismo estado con el correr del tiempo y después de cambios de configuraciones en *hardware* o *software*.

Las características deseables en un *benchmark* son portabilidad, simplicidad, estabilidad y reproducción de resultados. Esto permite que sean utilizadas para realizar mediciones cuantitativas y así realizar comparaciones de optimizaciones o entre sistemas de cómputo diferentes. También se pide que el tiempo de ejecución sea razonable y que el tamaño del problema sea ajustable para poder mantener su utilidad con el paso del tiempo y el avance de las tecnologías.

A continuación se introducen algunas de las más utilizadas para cómputo de altas prestaciones (listadas en la tabla 3.1), y posteriormente algunos detalles específicos e instancias de sus datos de salida para ser utilizados a manera de ejemplo.

Tabla 3.1: Benchmarks

Benchmark	**Componente**	**Descripción**
STREAM	Memoria	Ancho de banda sostenido
Linpack	Procesador	Operaciones de punto flotante
IMB Ping Pong	Red	Latencia/Ancho de banda de red
HPCC	Sistema	Múltiples componentes

Los *benchmarks* pueden ser utilizados para diferentes propósitos. Primero, los valores reportados son usados como referencia para contrastar rendimiento. Segundo, su desviación demuestra que algo ha cambiado en el sistema (por lo tanto su no desviación indica que el sistema sigue saludable). Por último, un *benchmark* sintético implementando el cómputo que uno quiere realizar muestra el rendimiento máximo posible a obtener en la práctica.

3.1.1. STREAM

STREAM [10] es un *benchmark* sintético que mide el ancho de banda de memoria sostenido en MB/s y el rendimiento de computación relativa de algunos vectores simples de cálculo. Se utiliza para dimensionar el ancho de banda de acceso de escritura o lectura a la jerarquía de memoria principal del sistema bajo análisis. Dentro de una misma ejecución de este *benchmark* se ejercitan diferentes operaciones en memoria, listadas en la tabla 3.2.

Tabla 3.2: Operaciones del Benchmark STREAM

Función	**Operación**	**Descripción**
copy	$\forall i\ b_i = a_i$	Copia simple
scale	$\forall i\ b_i = c \times a_i$	Multiplicación escalar
add	$\forall i\ c_i = b_i + a_i$	Suma directa
triad	$\forall i\ c_i = b_i + c \times a_i$	Suma y multiplicación escalar

La salida en pantalla muestra entonces los diferentes tiempos conseguidos y la cantidad de información transferida por unidad de tiempo. Como último paso, el programa valida también la solución computada.

```
STREAM version $Revision: 1.2 $
-------------------------------------------------------------
This system uses 8 bytes per DOUBLE PRECISION word.
-------------------------------------------------------------
Array size = 10000000, Offset = 0
Total memory required = 228.9 MB.
Each test is run 10 times, but only the *best* time is used.
-------------------------------------------------------------
Function     Rate (MB/s)   Avg time     Min time     Max time
Copy:        4764.1905       0.0337       0.0336       0.0340
Scale:       4760.2029       0.0338       0.0336       0.0340
Add:         4993.8631       0.0488       0.0481       0.0503
Triad:       5051.5778       0.0488       0.0475       0.0500
-------------------------------------------------------------
Solution Validates
```

3.1.2. Linpack

Linpack [11] es un conjunto de subrutinas *FORTRAN* que resuelven problemas de álgebra lineal como ecuaciones lineales y multiplicación de matrices. High Performance Linpack (HPL) [12] es una versión portable del *benchmark* que incluye el paquete Linpack pero modificado para sistemas de memoria distribuida.

Este *benchmark* es utilizado mundialmente para la comparación de la velocidad de las supercomputadoras en el ranking TOP500. Un gráfico del TOP500 de los últimos años (Figura 3.1) demuestra claramente la tendencia en crecimiento de rendimiento; también la relación entre el primero, el último y la suma de todos los sistemas en la lista.

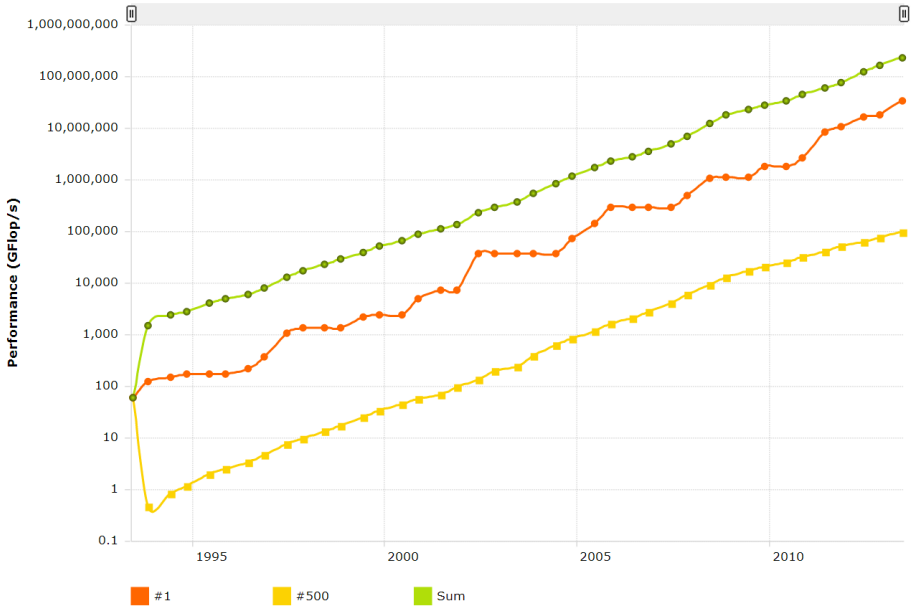

Figura 3.1: Rendimiento Agregado del Top500 [Top500])

Este *benchmark* requiere conocimiento avanzado para una correcta configuración, por ejemplo el tamaño de bloque que se va a utilizar para la distribución de trabajo debe estar directamente relacionado con el tamaño del *cache* de memoria del procesador.

La salida en pantalla resume entonces los datos de entrada y los resultados conseguidos. Como último paso el programa valida que los resultados sean correctos.

```
================================================================
HPLinpack 2.0 - High-Performance Linpack benchmark - Sep 10, 2008
Written by A. Petitet and R. Clint Whaley
================================================================
The following parameter values will be used:
N      :   28888
NB     :     168
PMAP   : Row-major process mapping
P      :       4
Q      :       4
PFACT  :   Right
NBMIN  :       4
NDIV   :       2
RFACT  :   Crout
```

```
BCAST  :  1ringM
DEPTH  :       0
SWAP   : Mix (threshold = 64)
L1     : transposed form
U      : transposed form
EQUIL  : yes
ALIGN  : 8 double precision words
----------------------------------------------------------------------
- The matrix A is randomly generated for each test.
- The relative machine precision (eps) is taken to be 1.110223e-16
- Computational tests pass if scaled residuals are less than 16.0

Column=000168 Fraction=0.005 Mflops=133122.97
...
Column=025872 Fraction=0.895 Mflops=98107.60
======================================================================
T/V                N   NB   P    Q             Time             Gflops
WR01C2R4       28888  168   4    4           165.83          9.693e+01
----------------------------------------------------------------------
||Ax-b||_oo/(eps*(||A||_oo*||x||_oo+||b||_oo)*N) = 0.0043035 .. PASSED
======================================================================
Finished      1 tests with the following results:
1 tests completed and passed residual checks,
0 tests completed and failed residual checks,
0 tests skipped because of illegal input values.
```

Existe cierta controversia de que no es una buena forma de ejercitar un sistema de cómputo distribuido ya que no implica uso significativo de la red, sólo procesamiento intensivo de aritmética de punto flotante sobre la jerarquía local de memoria.

3.1.3. Intel MPI Benchmarks

Es un conjunto de *benchmarks* cuyo objetivo es ejercitar las funciones más importantes del estándar para librerías de paso de mensajes (MPI, por sus siglas en inglés) [13]. El más conocido es el popular ping-pong, el cual ejercita la transmisión de mensajes ida y vuelta entre dos nodos de cómputo con diferentes tamaños de mensajes. [14].

Para obtener el máximo ancho de banda disponible, se ejercita la comunicación a través de mensajes con datos grandes. Para obtener la mínima latencia, se ejercita la comunicación con mensajes vacíos, es decir transmitiendo mensajes sin dato alguno.

```
# Intel (R) MPI Benchmark Suite V3.1, MPI-1 part
# Date                  : Wed Mar  3 10:45:16 2010
# Machine               : x86_64
# System                : Linux
# Release               : 2.6.16.46-0.12-smp
# Version               : #1 SMP Thu May 17 14:00:09 UTC 2007
# MPI Version           : 2.0
# MPI Thread Environment: MPI_THREAD_SINGLE
```

```
# Calling sequence was: ../IMB-MPI1 pingpong
# Minimum message length in bytes:   0
# Maximum message length in bytes:   4194304
#
# MPI_Datatype                   :   MPI_BYTE
# MPI_Datatype for reductions    :   MPI_FLOAT
# MPI_Op                         :   MPI_SUM
#
# List of Benchmarks to run: PingPong
#---------------------------------------------------
# Benchmarking PingPong
# #processes = 2
#---------------------------------------------------
#bytes     #repetitions  t[usec]     Mbytes/sec
0              1000           17.13        0.00
1              1000           17.89        0.05
2              1000           17.82        0.11
4              1000           17.95        0.21
...
1048576    40              8993.23    111.19
2097152    20              17919.20  111.61
4194304    10              35766.45  111.84
```

3.1.4. HPC Challenge

El *benchmark* HPC Challenge [15] (HPCC) está compuesto internamente por un conjunto de varios núcleos de cómputo: entre ellos STREAM, HPL, Ping Pong, Transformadas de *Fourier* y otros ejercitando la red de comunicación.

Este benchmark muestra diferentes resultados que son representativos y puestos en consideración de acuerdo al tipo de aplicación en discusión. La mejor máquina depende de la aplicación específica a ejecutar, ya que algunas aplicaciones necesitan mejor ancho de banda de memoria, mejor canal de comunicación, o simplemente la mayor capacidad de cómputo de operaciones flotantes posible.

Una analogía interesante para entender cómo el *benchmark* se relaciona con diferentes núcleos de cómputo se muestra en la Figura 3.2. Por ejemplo al tener un problema que utiliza principalmente acceso a memoria local, se puede suponer que un sistema con buenos resultados de STREAM va ser útil.

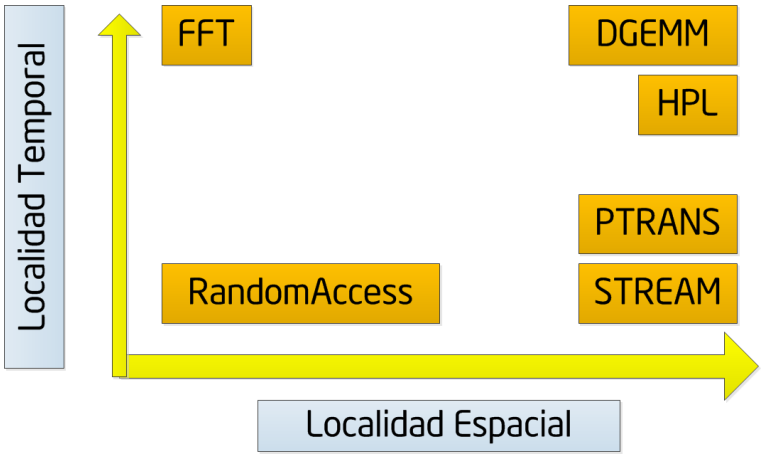

Figura 3.2: Localidad temporal versus espacial en resultados de HPCC

Para una mejor comparación de resultados de HPCC se utilizan diagramas denominados *kiviats*, un ejemplo se muestra en la Figura 3.3. Los resultados están normalizados hacia uno de los sistemas, y se puede identificar mejor rendimiento en FLOPs por poseer mejores DGEMM y HPL en comparación.

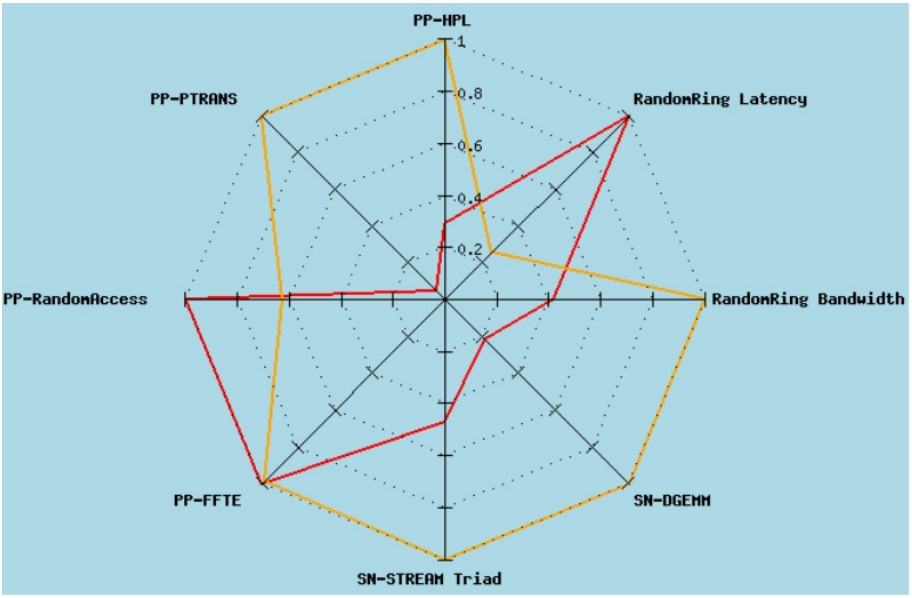

Figura 3.3: Diagrama Kiviat [Top500]

Un ejemplo de la salida que se muestra durante la ejecución se muestra a continuación.

```
This is the DARPA/DOE HPC Challenge Benchmark version 1.2.0 October 2003
Produced by Jack Dongarra and Piotr Luszczek
```

```
Innovative Computing Laboratory
University of Tennessee Knoxville and Oak Ridge National Laboratory

Begin of Summary section.

    VersionMajor=1                                  MPIRandomAccess_ErrorsFraction=0
    VersionMinor=2                                  MPIRandomAccess_ExeUpdates=536870912
    LANG=C                                          MPIRandomAccess_GUPs=0.0176327
    Success=1                                       MPIRandomAccess_TimeBound=-1
    CommWorldProcs=3                                MPIRandomAccess_Algorithm=0
    MPI_Wtick=1.000000e-06                          RandomAccess_N=33554432
    HPL_Tflops=0.0674008                            StarRandomAccess_GUPs=0.0186362
    HPL_time=26.3165                                SingleRandomAccess_GUPs=0.0184568
    HPL_eps=1.11022e-16                             STREAM_VectorSize=21332081
    HPL_N=13856                                     STREAM_Threads=8
    HPL_NB=64                                       StarSTREAM_Copy=4.34705
    HPL_nprow=1                                     StarSTREAM_Scale=3.24366
    HPL_npcol=3                                     StarSTREAM_Add=3.41196
    HPL_depth=2                                     StarSTREAM_Triad=3.46198
    HPL_nbdiv=2                                     SingleSTREAM_Copy=4.53628
    HPL_nbmin=8                                     SingleSTREAM_Scale=3.38984
    HPL_cpfact=C                                    SingleSTREAM_Add=3.59073
    HPL_crfact=R                                    SingleSTREAM_Triad=3.65083
    HPL_ctop=1                                      FFT_N=8388608
    HPL_order=R                                     StarFFT_Gflops=2.17339
    dweps=1.110223e-16                              SingleFFT_Gflops=2.26806
    sweps=5.960464e-08                              MPIFFT_N=8388608
    HPLMaxProcs=3                                   MPIFFT_Gflops=1.7043
    HPLMinProcs=3                                   MPIFFT_maxErr=1.77722e-15
    DGEMM_N=4618                                    MPIFFT_Procs=2
    StarDGEMM_Gflops=68.9053                        MaxPingPongLatency_usec=5.37932
    SingleDGEMM_Gflops=70.2692                      RandomRingLatency_usec=5.70686
    PTRANS_GBs=0.794254                             MinPingPongBandwidth_GBytes=0.675574
    PTRANS_time=0.479293                            NaturalRingBandwidth_GBytes=0.531278
    PTRANS_residual=0                               RandomRingBandwidth_GBytes=0.529161
    PTRANS_n=6928                                   MinPingPongLatency_usec=5.24521
    PTRANS_nb=64                                    AvgPingPongLatency_usec=5.30978
    PTRANS_nprow=1                                  MaxPingPongBandwidth_GBytes=0.682139
    PTRANS_npcol=3                                  AvgPingPongBandwidth_GBytes=0.678212
    MPIRandomAccess_N=134217728                     NaturalRingLatency_usec=5.79357
    MPIRandomAccess_time=30.4475                    FFTEnblk=16
    MPIRandomAccess_Check=14.0705                   FFTEnp=8
    MPIRandomAccess_Errors=0                        FFTEl2size=1048576

End of Summary section.
End of HPC Challenge tests.
```

3.2. Herramientas

Se recomienda un proceso de aplicación gradual empezando por herramientas generales de alto nivel que analizan la aplicación como un todo; terminando con herramientas de bajo nivel que proveen detalles complejos de granularidad más fina en componentes particulares del sistema. Esto permite ir analizando el rendimiento sin tener que enfrentar la dificultad de un análisis complejo y extensivo desde un principio. Una lista de las herramientas más conocidas se muestra en la Tabla 3.3.

Tabla 3.3: Aplicación Gradual de Herramientas

Característica	**Herramientas**
Capacidad del sistema	Benchmark HPCC
Medición de ejecución	`time`, `gettimeofday()`, `MPI_WTIME()`
Perfil de ejecución	profilers: `gprof, perf`
Comportamiento de la aplicación	profilers: `gprof, perf`
Comportamiento de librerías	profilers: valgrind, MPI vampir.
Comportamiento del sistema	profilers: oprofile, perf
Vectorización	compilador: gcc
Contadores en *hardware*	oprofile, PAPI, perf

A grandes rasgos el procedimiento es el siguiente:

1. Se establece una línea de comparación al ejecutar una prueba de rendimiento del sistema, *HPCC* brinda un conjunto de métricas muy completo.
2. Se utilizan herramientas para medir el tiempo de ejecución de la aplicación sobre diferentes escenarios. time permite una ejecución directa sin modificación de código, `gettimeofday()` requiere modificación de código pero puede ser utilizados con mayor libertad dentro de la aplicación. En el caso de estar utilizando la librería MPI `MPI_WTime()` y la herramienta VAMPIR[1] proveen soporte específico para análisis de rendimiento.
3. Se dimensiona el comportamiento de la aplicación mediante un perfil de ejecución y un análisis de cuello de botella utilizando `gprof`.
4. Se analiza el comportamiento del sistema ejecutando la aplicación mediante `oprofile` [2] o `perf` [3].
5. Se revisa el reporte del compilador para comprobar que se estén vectorizando los ciclos de cálculo más intensivos.
6. Se analiza el comportamiento de las unidades de cómputo utilizando soporte de *hardware* mediante herramientas como `perf`, `oprofile` y *Performance Application Programming Interface* (PAPI) [4].

3.3. Tiempo de Ejecución

Esta sección revisa como medir el tiempo de ejecución global de una aplicación, incluyendo ejemplos.

3.3.1. Tiempo de ejecución global

Para medir el tiempo de ejecución de un comando en consola se utiliza `time(1)`. Aunque rudimentaria, esta simple herramienta no necesita de instrumentación de código y se encuentra disponible en cualquier distribución *GNU/-Linux*. El intérprete de comandos tiene su propia versión embebida, sin embargo el del sistema brinda información del uso de otros recursos del sistema, usualmente localizado en `/usr/bin/time`.

[1] http://www.vampir.eu
[2] http://oprofile.sourceforge.net
[3] https://perf.wiki.kernel.org
[4] http://icl.cs.utk.edu/papi

```
$ /usr/bin/time -v ./script.sh
1
        Command being timed: "./script.sh"
        User time (seconds): 0.61
        System time (seconds): 0.00
        Percent of CPU this job got: 99%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.62
        Average shared text size (kbytes): 0
        Average unshared data size (kbytes): 0
        Average stack size (kbytes): 0
        Average total size (kbytes): 0
        Maximum resident set size (kbytes): 4560
        Average resident set size (kbytes): 0
        Major (requiring I/O) page faults: 0
        Minor (reclaiming a frame) page faults: 668
        Voluntary context switches: 6
        Involuntary context switches: 2
        Swaps: 0
        File system inputs: 0
        File system outputs: 0
        Socket messages sent: 0
        Socket messages received: 0
        Signals delivered: 0
        Page size (bytes): 4096
        Exit status: 0
```

3.3.2. Reloj del sistema

La librería de sistema permite acceder a llamadas al sistema operativo para obtener datos precisos del paso del tiempo. Las más utilizadas son `gettimeofday(3)` y `clock(3)`, aunque éste último se comporta de manera especial al utilizar multithreading ya que suma el tiempo ejecutado en cada core.

El siguiente código ejemplifica como obtener un número de segundos en una representación de punto flotante de doble precisión, permitiendo una granularidad de medición adecuada.

```
double wtime(void)
{
  double sec;
  struct timeval tv;

  gettimeofday(&tv, NULL);
  sec = tv.tv_sec + tv.tv_usec/1000000.0;
  return sec;
}
```

3.3.3. Ejemplo: `time`

El siguiente ejemplo muestra como obtener información del uso de los recursos del sistema por parte de una aplicación.

```
$ /usr/bin/time -v ./program
Command being timed: "./program"
User time (seconds): 0.00
System time (seconds): 0.00
Percent of CPU this job got: 0%
Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.00
Average shared text size (kbytes): 0
Average unshared data size (kbytes): 0
Average stack size (kbytes): 0
Average total size (kbytes): 0
Maximum resident set size (kbytes): 3808
Average resident set size (kbytes): 0
```

```
Major (requiring I/O) page faults: 0
Minor (reclaiming a frame) page faults: 289
Voluntary context switches: 1
Involuntary context switches: 1
Swaps: 0
File system inputs: 0
File system outputs: 0
Socket messages sent: 0
Socket messages received: 0
Signals delivered: 0
Page size (bytes): 4096
Exit status: 0
```

3.4. Perfil de Ejecución Funcional

Estas herramientas denominadas *profilers* extraen el perfil dinámico de una aplicación en tiempo de ejecución. Se instrumenta la aplicación con una opción específica que incluye información de uso de las diferentes partes del programa y los recursos del sistema como por ejemplo procesador y memoria.

La aplicación debe ejecutarse con un conjunto de datos prefijado. El conjunto de datos debe ser representativo y debe también ejercitar la aplicación por una cantidad de tiempo suficiente como para intensificar el uso de los recursos. Los datos del perfil de una ejecución son luego obtenidos en la forma de un archivo de datos, luego se procede a procesar los datos acumulados con un analizador respectivo.

Provee un perfil plano que consiste en una simple lista de las funciones ejecutadas ordenadas por la cantidad acumulada de tiempo utilizado. También provee el gráfico de llamadas anidadas, que muestra el tiempo utilizado por cada función en llamadas sucesivas. Las funciones recursivas son manejadas de manera especial ya que imposibilitan el armado de relaciones de dependencias.

3.4.1. Ejemplo: `gprof`

El perfil de ejecución de `gprof` muestra el tiempo individual y el tiempo acumulado en segundos de cada línea de código de la aplicación. Los binarios deben ser compilados con información extra de depuración, en el caso de `gcc`, las opciones necesarias son `-g -pg`. Si `-g` no se encuentra presente entonces no se provee el reporte detallado por línea de ejecución. Esto permite identificar donde se está consumiendo tiempo durante la ejecución. La herramienta también muestra un cuadro de las llamadas entre funciones realizadas por el programa. Esto permite visualizar el esquema de dependencias durante la ejecución.

A continuación se muestra como realizar la compilación incluyendo información de depuración específica, además de un caso concreto contra una aplicación simulando el juego de la vida [16].

```
$ gcc -g -pg program.c -o program
$ ./program
$ gprof program
...
```

```
Flat profile:
Each sample counts as 0.01 seconds.
% cumulative self self total
time seconds seconds calls us/call us/call name
37.50 0.15 0.15 48000 3.12 3.12 Life::neighbor_count(int, int)
...
```

```
Call graph
granularity: each sample hit covers 4 byte(s) for 2.50% of 0.40 seconds
index % time    self  children    called     name
       0.02 0.15 12/12 main [2]
[1] 42.5 0.02 0.15 12 Life::update(void) [1]
       0.15 0.00 48000/48000 Life::neighbor_count(int, int) [4]
--
            0.00    0.17       1/1           _start [3]
[2]     42.5    0.00    0.17       1           main [2]
            0.02    0.15      12/12          Life::update(void) [1]
            0.00    0.00      12/12          Life::print(void) [13]
            0.00    0.00      12/12          to_continue(void) [14]
            0.00    0.00       1/1           instructions(void) [16]
            0.00    0.00       1/1           Life::initialize(void) [15]
--
```

3.5. Perfil de Ejecución asistido por *Hardware*

Un *profiler* puede utilizar el *hardware* para analizar el uso de los recursos disponibles a nivel de núcleo del sistema operativo. Actúa de forma transparente a nivel global. Utiliza contadores de *hardware* del CPU y utiliza interrupciones de un temporizador cuando no logra detectar soporte específico en *hardware*. Aunque tiene un costo adicional inherente, la sobrecarga es mínima.

Para obtener un perfil de ejecución representativo, usualmente se recomienda detener toda aplicación o servicio no relevante en el sistema. La herramienta de por si no requiere acceder al código fuente de la aplicación, pero si esta disponible el código correspondiente se muestra anotado con contadores si hay símbolos de depuración en el binario.

Los registros de *hardware* implementando contadores más utilizados son los siguientes:

1. Cantidad total de ciclos de procesador
2. Cantidad total de instrucciones ejecutadas
3. Cantidad de ciclos detenidos por espera de acceso a memoria
4. Cantidad de instrucciones de punto flotante
5. Cantidad de fallos de cache de nivel uno (L1)
6. Cantidad de instrucciones de carga y descarga

En núcleos más viejos que la versión 2.6, en lugar de `oprofile` se recomienda utilizar `perf`. Al estar implementados a nivel de núcleo, éstos evitan las llamadas al sistema y tienen una sobrecarga de un orden de magnitud menor que los *profilers* a nivel de aplicación. Las herramientas propietarias suelen tener acceso a contadores más específicos e incluso programables para funciones determinadas de medición.

3.5.1. Ejemplo: `perf`

A continuación se demuestra la información provista por `perf` en sus diferentes modos de ejecución: estadísticas de contadores, perfil de sistema y por último perfil de aplicación.

```
$ perf stat -B program

Performance counter stats for 'program':

          5,099 cache-misses  0.005 M/sec (scaled from 66.58%)
        235,384 cache-references 0.246 M/sec (scaled from 66.56%)
      9,281,660 branch-misses 3.858 %     (scaled from 33.50%)
    240,609,766 branches 251.559 M/sec (scaled from 33.66%)
  1,403,561,257 instructions  0.679 IPC   (scaled from 50.23%)
  2,066,201,729 cycles 2160.227 M/sec (scaled from 66.67%)
            217 page-faults 0.000 M/sec
              3 CPU-migrations 0.000 M/sec
             83 context-switches 0.000 M/sec
     956.474238 task-clock-msecs 0.999 CPUs

     0.957617512  seconds time elapsed
```

```
$ perf record ./mm
$ perf report
# Events: 1K cycles
# Overhead Command Shared Object Symbol
28.15% main mm [.] 0xd10b45
4.45% swapper  [kernel.kallsyms] [k] mwait_idle_with_hints
4.26% swapper  [kernel.kallsyms] [k] read_hpet
...
```

```
 Percent |    Source code & Disassembly of program
         :    Disassembly of section .text:
         :    08048484 <main>:
         :    #include <string.h>
         :    #include <unistd.h>
         :    #include <sys/time.h>
         :
         :    int main(int argc, char **argv)
         :    {
0.00:    8048484:       55                      push   %ebp
0.00:    8048485:       89 e5                   mov    %esp,%ebp
...
0.00:    8048530:       eb 0b                   jmp 804853d <main+0xb9>
         :                            count++;
14.22:    8048532:       8b 44 24 2c             mov    0x2c(%esp),%eax
0.00:    8048536:       83 c0 01                add    $0x1,%eax
14.78:    8048539:       89 44 24 2c             mov    %eax,0x2c(%esp)
         :            memcpy(&tv_end, &tv_now, sizeof(tv_now));
         :            tv_end.tv_sec += strtol(argv[1], NULL, 10);
         :            while (tv_now.tv_sec < tv_end.tv_sec ||
         :                   tv_now.tv_usec < tv_end.tv_usec) {
         :                    count = 0;
         :                    while (count < 100000000UL)
14.78:    804853d:       8b 44 24 2c             mov    0x2c(%esp),%eax
56.23:    8048541:       3d ff e0 f5 05          cmp    $0x5f5e0ff,%eax
0.00:    8048546:       76 ea                   jbe 8048532 <main+0xae>
...
```

Este punto de análisis requiere conocimiento avanzado de como funciona el CPU utilizado, su acceso a memoria y los costos de las diferentes instrucciones soportadas. Una fuente de consulta debe incluir conceptos generales de arquitectura de procesadores [17] e información de los fabricantes [18].

3.6. Reporte de Vectorización

Una herramienta de bajo nivel para analizar rendimiento es el mismo compilador que debería estar vectorizando los ciclos de cómputo intensivo. Esto es muy útil para detectar si los cuellos de botella ya se encuentran optimizados o no.

Por ejemplo, GCC provee opciones específicas que deben ser provistas para mostrar el reporte.

```
$ gcc -c -O3 -ftree-vectorizer-verbose=1 ex.c
ex.c:7: note: LOOP VECTORIZED.
ex.c:3: note: vectorized 1 loops in function.
$ gcc -c -O3 -ftree-vectorizer-verbose=2 ex.c
ex.c:10: note: not vectorized: complicated access pattern.
ex.c:10: note: not vectorized: complicated access pattern.
ex.c:7: note: LOOP VECTORIZED.
ex.c:3: note: vectorized 1 loops in function.
$ gcc -c -O3 -fdump-tree-vect-details ex.c
...
```

En el caso de existir código recursivo, podemos comprobar que no suele estar soportado por los compiladores actuales.

```
$ gcc -Wall -Wextra -O3 -ftree-vectorizer-verbose=4 -g queen.c
queen.c:22: note: vectorized 0 loops in function.
queen.c:35: note: vectorized 0 loops in function.
```

Capítulo 4

Procedimiento de Análisis

Este capítulo propone un procedimiento sistemático de análisis.

4.1. Proceso

La Figura 4.1 muestra a grandes rasgos las etapas del proceso. Primero se establece una línea base de rendimiento del sistema utilizando pruebas conocidas. Luego se procede a trabajar en etapas iterativas asegurando en cada paso la estabilidad de los resultados, una correcta utilización de recursos y utilizando un perfil de ejecución. Luego de optimizar y comprobar la mejora, se vuelve a empezar el ciclo.

Figura 4.1: Procedimiento de Análisis

4.2. Paso a Paso

A continuación se muestran los pasos a realizar, junto con preguntas que guían el análisis de rendimiento de una aplicación.

4.2.1. Pruebas de Referencia

1. Ejecutar pruebas de rendimiento sobre el sistema a utilizar para poder entender sus capacidades máximas en contraste con las teóricas.

```
$ sudo apt-get install hpcc
$ mpirun -n `grep -c proc /proc/cpuinfo` ./hpcc
$ cat hpccoutf.txt
```

   a) ¿Los resultados reflejan las capacidades esperadas del sistema?
   b) ¿Los FLOPS se aproximan al rendimiento de un sistema similar?
   c) ¿El rendimiento es $CORES \times CLOCK \times FLOPS/CYCLE$?
   d) ¿La latencia y ancho de banda de la memoria es la esperada?

2. Comprobar variación de resultados para conocer la estabilidad de los mismos. La desviación estándar debe ser menor a 3 sigmas. Establecer cual es el promedio geométrico a usar como referencia para comparaciones futuras.

```
$ for i in `seq 1 32`; do /usr/bin/time -v ./app >> time.csv; done
```

   a) ¿Son los resultados estables?
   b) ¿La desviación estándar es menor que 3?
   c) ¿Cuál es el promedio geométrico para comparaciones futuras?
   d) ¿ Es necesario incrementar el problema para mejorar la desviación?
   e) ¿ Es posible reducir el tiempo sin afectar la desviación?

3. Escalar el problema para dimensionar la cantidad de trabajo según el tamaño del problema.

```
$ for size in `seq 1024 1024 10240`; do /usr/bin/time -v ./app
    $size >> size.log; done
```

   a) ¿Cuál es la relación entre el tiempo de las diferentes ejecuciones?
   b) ¿Es la relación lineal o constante?

4. Escalar cómputo para luego calcular límite de mejoras con *Amdalah* y *Gustafson*.

```
$ for threads in `grep -c proc /proc/cpuinfo | xargs seq 1`; do
    OMP_NUM_THREADS=$threads ./app >> threads.log; done
```

   a) ¿Cuál es la relación entre el tiempo de las diferentes ejecuciones?
   b) ¿Es la relación lineal o constante?
   c) ¿Qué porcentaje de la aplicación se estima paralelo?
   d) ¿Cual es la mejora máxima posible?

5. Generar el perfil de llamadas a funciones dentro de la aplicación para revisar el diseño de la misma y los posibles cuellos de botella a resolver.

```
$ gcc -g -pg app.c -o app
$ ./app
```

```
$ gprof --flat-profile --graph --annotated-source app
...
```

a) ¿Cómo esta diseñada la aplicación?
b) ¿Que dependencias en librerías externas tiene?
c) ¿Implementa algún núcleo de cómputo conocido encuadrado dentro de librerías optimizadas como BLAS?
d) ¿En que archivos, funciones y líneas se concentra la mayor cantidad de tiempo de cómputo?

6. Utilizar el profiler a nivel de sistema

```
$ prof stat ./app
$ prof record ./app
$ prof report
```

a) ¿Cómo se comporta el sistema durante la ejecución de la aplicación?
b) ¿Son las métricas de contadores de *hardware* las esperadas?
c) ¿Es la aplicación la gran concentradora de los recursos disponibles?
d) ¿Qué instrucciones de *hardware* son las utilizadas en el cuello de botella?

7. Comprobar vectorizaciones

```
$ gcc -Wall -Wextra -O3 --report-loop
```

a) ¿Hay ciclos que no pueden ser automáticamente vectorizados?

Capítulo 5

Casos de Estudio

Este capítulo aplica el procedimiento propuesto en el capítulo anterior. Varios ejemplos no triviales de aplicaciones de altas prestaciones son implementados y analizados utilizando los benchmarks y herramientas anteriormente presentadas.

La información de soporte sirve para realizar un análisis justificando potenciales optimizaciones con números y comportamiento cuantificado. Sin embargo como realizar una optimización no es trivial y depende del conocimiento de algoritmos que exploten adecuadamente la arquitectura del sistema.

5.1. Rendimiento de Referencia

El sistema utilizado es el mismo para todos los casos de prueba. A continuación se muestra su configuración de *hardware* y *software* (Tabla 5.1), además del resultado del rendimiento según HPCC con un proceso por core disponible.

Tabla 5.1: Hardware y software de cada nodo de cómputo

Component	**Description**
Placa Principal	Intel®6 Series/C200 Series Chipset
CPU	Intel®Xeon®CoreTM i7-2620M CPU @ 2.7GHz
Controlador *Ethernet*	Intel®82579V Gigabit Network
Memoria RAM	8 GB DDR3
Sistema Operativo	Ubuntu 10.02 LTS 64-bits

Una ejecución del benchmark HPCC nos muestra entonces el rendimiento a ser considerado como ideal en la práctica.

```
HPL_Tflops=0.00701244
StarDGEMM_Gflops=2.21279
SingleDGEMM_Gflops=4.45068
PTRANS_GBs=0.661972
MPIRandomAccess_GUPs=0.000839626
StarRandomAccess_GUPs=0.0184879
SingleRandomAccess_GUPs=0.0359549
```

```
StarFFT_Gflops=1.11089
SingleFFT_Gflops=1.4984
MPIFFT_Gflops=2.42216
```

5.2. Multiplicación de Matrices

La multiplicación de matrices es una operación fundamental en múltiples campos de aplicación científica como la resolución de ecuaciones lineales y la representación de grafos y espacios dimensionales. Por ello existe abundante material sobre el tema. Tomando como ejemplo una implementación utilizando OpenMP [1], se procede a seguir la metodología propuesta en la sección anterior.

Para una matriz de `2048` números de punto flotante de precisión simple, el promedio de ejecución tomó `2.036` segundos, la media geométrica `2.03567`. La desviación de resultados `0.03645`. Por lo tanto se puede concluir que es un buen conjunto de entrada para analizar rendimiento.

Al realizar ejecuciones con tamaños de entrada creciente los resultados fueron: `N=1024 2.035678236` segundos, `N=2048 18.853` segundos, `N=3072 1m19.505` segundos y `N=4096 3m42.779s` (Figura 5.1. Por defecto se ejecuta 1 *thread* por *core* disponible en el CPU.

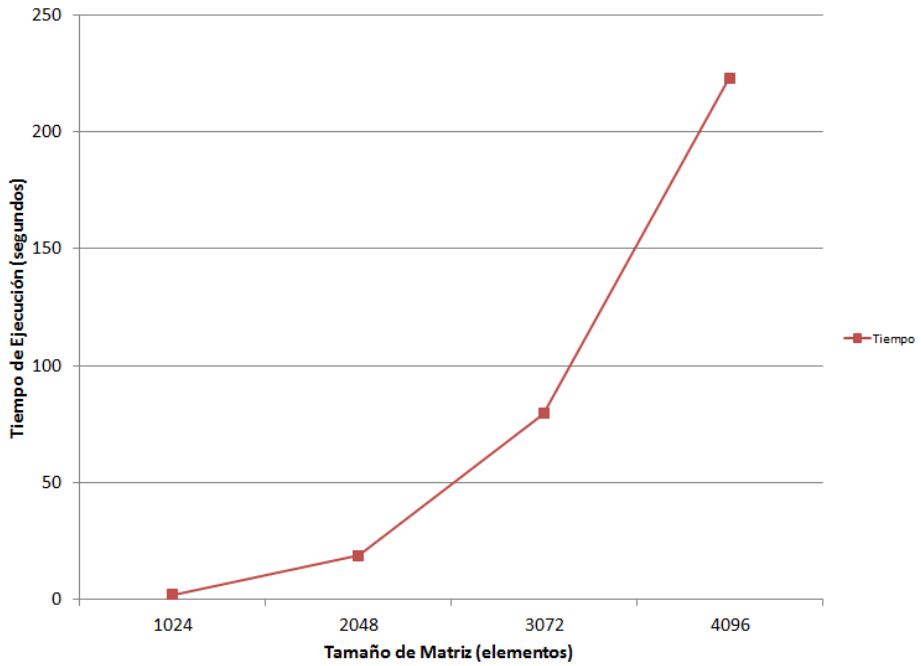

Figura 5.1: Tiempo de Ejecución con Diferentes Tamaños de Problema

Al realizar ejecuciones variando la cantidad de *threads* se obtuvieron los siguientes resultados (Figura 5.2). Calculamos entonces la proporción serial y

[1] http://blog.speedgocomputing.com/2010/08/parallelizing-matrix-multiplication.html.

paralela, 20 % y 80 % respectivamente. Los límites son entonces: *Amdalah* 5x y *Gustafson* 10x.

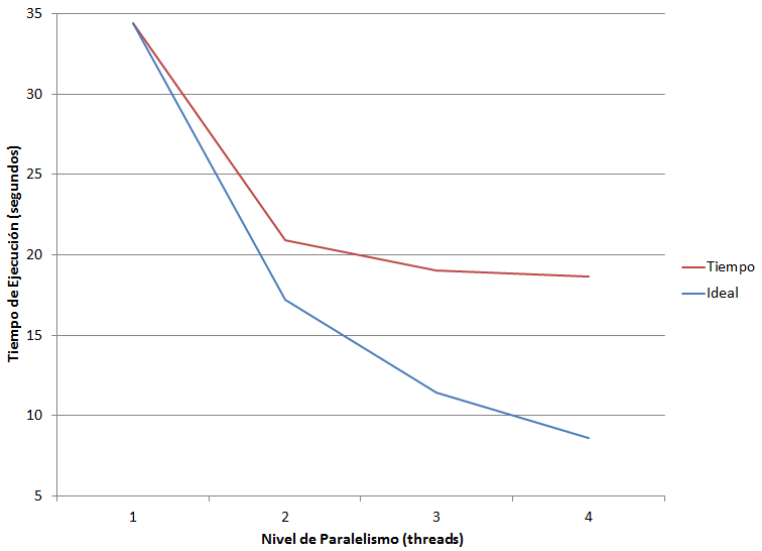

Figura 5.2: Tiempo de Ejecución con Diferente Número de Threads

Al aplicar las herramientas vistas previamente se identifica claramente que la multiplicación de los elementos de la matriz consume el mayor tiempo de cómputo.

```
91.59% 111.61 main._omp_fn.0 (matrix.c:23)
6.56% 7.99 main._omp_fn.0 (matrix.c:22)
2.59% 3.16 main._omp_fn.0 (matrix.c:23)
```

La gran mayoría del tiempo de ejecución del sistema se concentra en la aplicación.

```
97.43% matrix matrix [.] main._omp_fn.0
1.60% matrix [kernel.kallsyms] [k] __do_softirq
```

El ciclo principal realiza la mayoría de las operaciones rápidamente, pero demora en la acumulación de resultados.

```
for (i = 0; i < SIZE; ++i) {
  for (j = 0; j < SIZE; ++j) {
    for (k = 0; k < SIZE; ++k) {
      c[i][j] += a[i][k] * b[k][j];
0.45 movaps %xmm2,%xmm0
0.03 movlps (%rax),%xmm0
0.07 movhps 0x8(%rax),%xmm0
1.35 add $0x4,%rax
0.24 shufps $0x0,%xmm0,%xmm0
0.04 mulps  (%rdx),%xmm0
81.36 add $0x2000,%rdx
```

```
69278.383857 task-clock # 3.915 CPUs utilized
231 context-switches # 0.003 K/sec
4 CPU-migrations # 0.000 K/sec
12,461 page-faults # 0.180 K/sec
```

Diferentes alternativas de implementación de esta operación son demostradas en [1], la Figura 5.3 resume que utilizar una librería implementando BLAS es mejor que varios intentos de optimización.

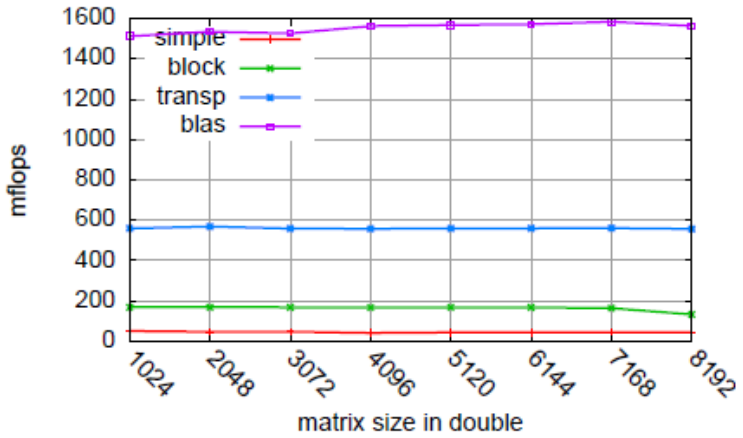

Figura 5.3: Comparación de Implementaciones

5.3. Distribución de Calor en Dos Dimensiones

Otro problema interesante es la simulación de transferencia de calor en un plano. Se utiliza una grilla donde cada celda transfiere calor a sus vecinos en una serie de ciclos finitas simulando el paso del tiempo [2]. Un gráfico de la salida del cálculo es mostrado en la Figura 5.4.

[2] http://www.rblasch.org/studies/cs580/pa5/#Source+Code-N100AC.

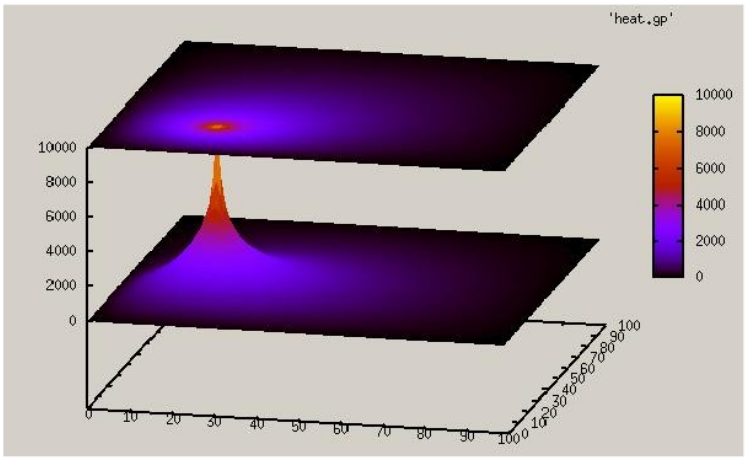

Figura 5.4: Simulación de Transferencia de Calor

Para un tamaño de matriz de `N=300` números de punto flotante de precisión simple, el promedio de ejecución tomó `21.07913` segundos, la media geométrica `21.06550` segundos. La desviación de resultados fue de `0.76358`. Por lo tanto se puede concluir que es un conjunto de datos de entrada estable.

Al realizar ejecuciones con tamaños de entrada creciente los resultados fueron: `N=300 21.0655`, `N=400 40.31`, `N=500 68.386`, `N=600 118.213`, `N=700 168.052` (Figura 5.5).

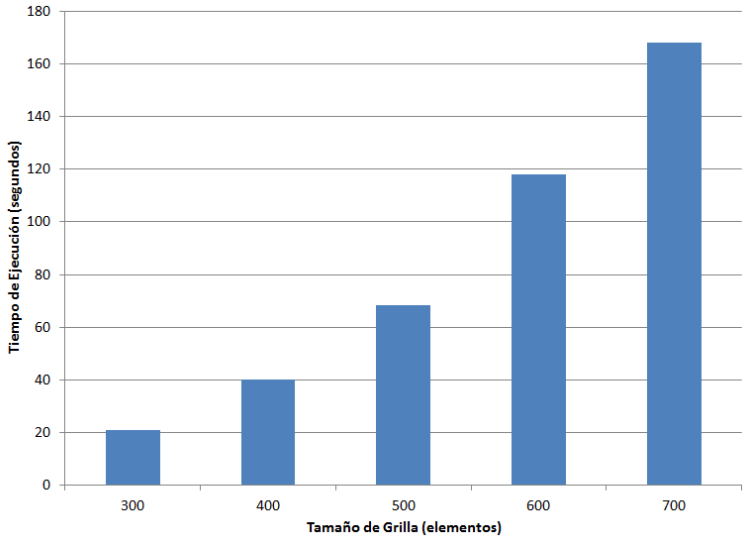

Figura 5.5: Tiempo de Ejecución con Diferente Tamaño de Grilla

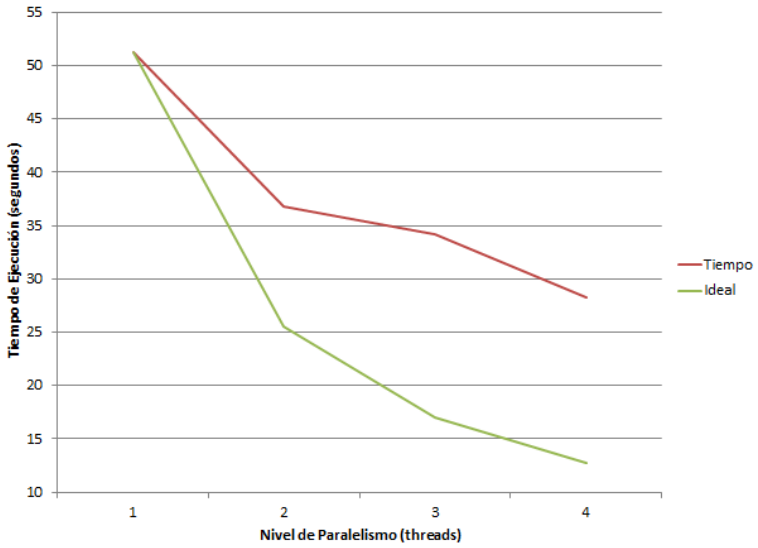

Figura 5.6: Tiempo de Ejecución con Diferente Número de Threads

Al realizar ejecuciones variando la cantidad de *threads* se obtuvieron los siguientes resultados (Figura 5.6). Calculamos entonces la proporción serial y paralela, 20 % y 80 % respectivamente. Los límites son entonces: *Amdalah* 5 x y *Gustafson* 20 x.

```
109960.157161 task-clock # 3.839 CPUs utilized
2,244 context-switches # 0.020 K/sec
6 CPU-migrations # 0.000 K/sec
1,160 page-faults # 0.011 K/sec
```

Un perfil del sistema nos localiza el computo en nuestro núcleo de computación, seguida del maneja de threads por parte de la implementación GNU de OpenMP.

```
78.38% heat heat [.] compute_one_iteration._omp_fn.0
20.61% heat libgomp.so.1.0.0 [.] 0x00000000000093be
```

Un perfil de la aplicación nos demuestra que el 35.05 % del tiempo de ejecución se concentra en la línea que realiza el promedio de las celdas vecinas. Y casi el 13.79 % donde se ajusta el promedio por la diferencia de temperatura por ciclo.

```
35.05% 28.71 compute_one_iteration._omp_fn.0 (heat.c:77)
13.79% 11.30 compute_one_iteration._omp_fn.0 (heat.c:82)
```

Un nivel más detallado nos muestra específicamente que instrucciones del procesador están en juego.

```
#pragma omp parallel for shared(solution,cur_gen,next_gen,diff_constant) private(i,j)
for (i = 1; i <= RESN; i++)
  for (j = 1; j <= RESN; j++)
```

```
    solution[next_gen][i][j] = solution[cur_gen][i][j] +
 21.79 addpd  %xmm1,%xmm0
 15.13 movlpd %xmm0,(%rdx,%rax,1)
  2.94 movhpd %xmm0,0x8(%rdx,%rax,1)
  5.66 add $0x10,%rax
```

5.4. Problema de las Ocho Reinas

El dilema trata en el problema de posicionar en un tablero de ajedrez ocho reinas sin que alguna amenace a las demás. Una reina amenaza a cualquier otra pieza en el tablero que este en la mismo diagonal, fila o columna.

El problema de encontrar todas las soluciones para un tamaño de tablero dado suele resolverse utilizando recursión [3].

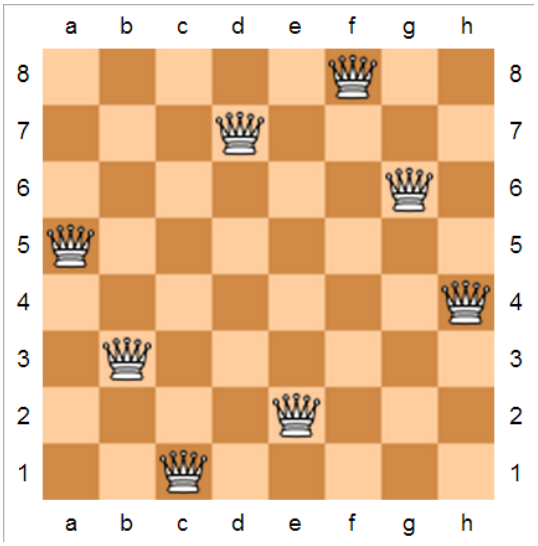

Figura 5.7: Solución simétrica del problema de las 8 reinas [Wikipedia]

Ejercitando el problema con `N=16` nos da un promedio de ejecución de `31.39727` segundos, y una media geométrica de `31.39346` segundos. Con una desviación de resultado de `0.5102`. Podemos concluir que los resultados son estables.

El escalamiento de tamaño y cantidad de unidades de cómputo refleja un comportamiento no deseado.

[3] http://www.stevenpigeon.org/blogs/hbfs/super-reines-openmp.cpp.

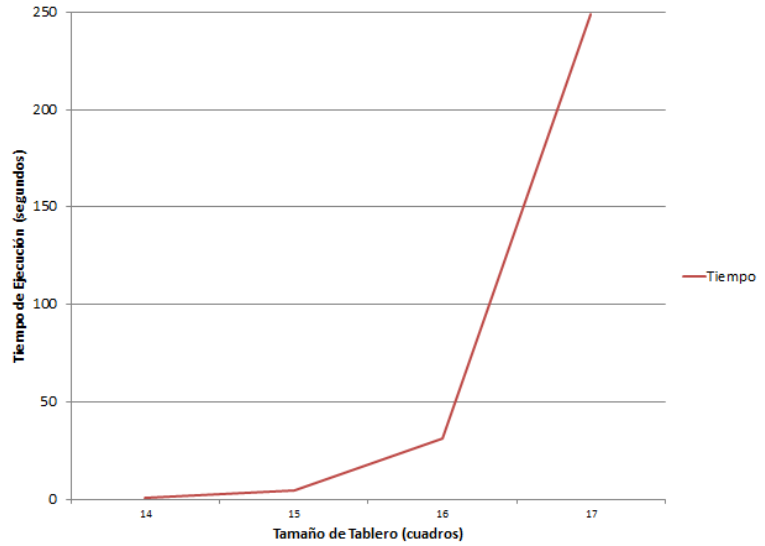

Figura 5.8: Tiempo de Ejecución con Diferente Tamaño de Tablero

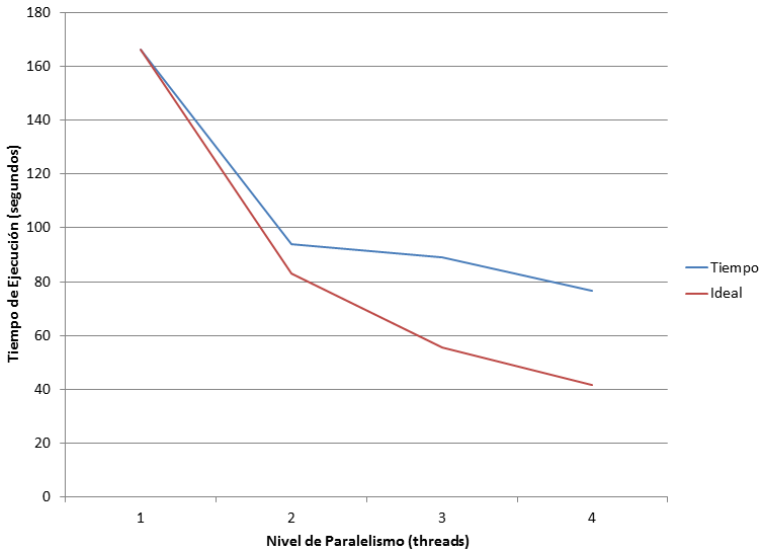

Figura 5.9: Tiempo de Ejecución con Diferente Tamaño de Tablero

```
302506.712420 task-clock                #    3.610 CPUs utilized
          545 context-switches          #    0.002 K/sec
            3 CPU-migrations            #    0.000 K/sec
          382 page-faults               #    0.001 K/sec
```

Un perfil del sistema muestra un uso casi totalitario por parte de la aplicación.

```
97.64% queens queens [.] solve(int, int, unsigned long,
```

```
  unsigned long, unsigned long)
1.56% queens [kernel.kallsyms] [k] __do_softirq
```

Un perfil de la aplicación muestra que el cuello de botella se encuentra en la determinación de amenazas por parte de otras reinas.

```
21.95 19.73 test (queens.cc:51)
12.81 11.52 test (queens.cc:51)
8.72 7.84 test (queens.cc:51)
```

Un detalle de la operación muestra el tiempo que toma cada instrucción.

```
solve(int, int, unsigned long, unsigned long, unsigned long,
  uint64_t diag135, uint64_t cols)
10.86 lea (%rdx,%r11,1),%ecx
{
  return ((cols & (1ull << j))
    + (diag135 & (1ull << (j+k)))
    + (diag45 & (1ull << (32+j-k))))==0;
0.85 mov %r14,%rdx
1.49 mov %rax,%rsi
2.95 shl %cl,%rdx
9.81 mov %r8,%rcx
1.15 and %rbp,%rsi
2.17 and %r12,%rcx
2.06 add %rsi,%rcx
4.73 mov %rdx,%rsi
1.49 and %r13,%rsi
```

Capítulo 6

Conclusiones

La optimización del rendimiento de una aplicación es algo no trivial. Es preciso realizar un análisis disciplinado del comportamiento y del uso de los recursos antes de empezar a optimizar. Las mejoras pueden ser no significativas si son realizadas en el lugar incorrecto.

Este trabajo soporta los primeros pasos de análisis de rendimiento para expertos del dominio de un problema científico utilizando computación de altas prestaciones. Se provee una metodología de uso de herramientas de soporte para principiantes, que puede ser utilizada como una lista de pasos resumidos para usuarios casuales, e incluso como referencia de consulta diaria para expertos.

Se resume también el estado del arte del análisis de rendimiento en aplicaciones de cómputo de altas prestaciones. Respecto a herramientas de soporte, se detallan diferentes opciones y se demuestra su aplicación en varios problemas simples aunque suficientemente interesantes. Este estudio propone un proceso de análisis de rendimiento gradual e iterativo que puede ser tomado como trabajo previo de una implementación práctica del mismo. Es decir incluyendo soporte automático para la aplicación de las herramientas y la generación integrada de reportes de rendimiento. La utilización de estas ideas en una aplicación del mundo real es materia pendiente, otra posibilidad es re-implementar desde cero alguna aplicación científica en colaboración con algún grupo de investigación y realizar varios ciclos de optimización para validar su utilidad.

Bibliografía

[1] Andres More. A Case Study on High Performance Matrix Multiplication. Technical report, 2008.

[2] Paul E. McKenney. *Is Parallel Programming Hard, And, If So, What Can You Do About It?* kernel.org, 2010.

[3] Thomas Sterling, Donald J. Becker, Daniel Savarese, John E. Dorband, Udaya A. Ranawake, and Charles V. Packer. Beowulf: A parallel workstation for scientific computation. In *In Proceedings of the 24th International Conference on Parallel Processing*, pages 11–14. CRC Press, 1995.

[4] Brendan Gregg. Linux Performance Analysis and Tools. Technical report, February 2013.

[5] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[6] John L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31:532–533, 1988.

[7] Alan H. Karp and Horace P. Flatt. Measuring parallel processor performance. *Commun. ACM*, 33(5):539–543, May 1990.

[8] Philip J. Fleming and John J. Wallace. How not to lie with statistics: the correct way to summarize benchmark results. *Commun. ACM*, 29(3):218–221, March 1986.

[9] Kendall Atkinson. *An Introduction to Numerical Analysis*. Wiley, 2 edition.

[10] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Technical Committee on Computer Architecture (TCCA) Newsletter*, Dec 1995.

[11] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart. *LINPACK User's Guide*. SIAM, 1979.

[12] A. Petitet, R. C. Whaley, Jack Dongarra, and A. Cleary. HPL - a portable implementation of the High-Performance linpack benchmark for Distributed-Memory computers.

[13] MPI: A Message-Passing interface standard. Technical report, University of Tennessee, May 1994.

[14] Rafael Garabato, Victor Rosales, and Andres More. Optimizing Latency in Beowulf Clusters. *CLEI Electron. J.*, 15(3), 2012.

[15] Piotr Luszczek, Jack J. Dongarra, David Koester, Rolf Rabenseifner, Bob Lucas, Jeremy Kepner, John Mccalpin, David Bailey, and Daisuke Takahashi. Introduction to the HPC Challenge Benchmark Suite. Technical report, 2005.

[16] Martin Gardner. Mathematical Games: The fantastic combinations of John Conway's new solitaire game 'life'. *Scientific American*, pages 120–123, October 1970.

[17] John L. Hennessy and David A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2002.

[18] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Number 248966-018. March 2009.