Universidad Nacional de La Plata
Facultad de Informática

Tesis presentada para obtener el grado de
Magister en Cómputo de Altas Prestaciones

Infraestructura para el Análisis de Rendimiento

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

Octubre de 2014

Resumen

En el área del cómputo de altas prestaciones las aplicaciones son construidas por especialistas del dominio del problema, no siempre expertos en optimización de rendimiento. Se identifica entonces la necesidad de una infrastructura automática para soportar el análisis de rendimiento.

Este trabajo revisa la construcción de una infraestructura que simplifica el análisis de rendimiento; incluyendo casos de estudio como etapa de validación. El objetivo es facilitar la tarea evitando la tediosa recolección de datos relevantes de modo manual, permitiendo más tiempo de experimentación y análisis.

En particular, este trabajo contribuye con un generador automático de reporte de rendimiento para aplicaciones de cómputo de altas prestaciones utilizando tecnología *OpenMP* sobre sistemas *GNU/Linux*.

Índice general

Capítulo 1

Introducción

Este capítulo introduce el tema bajo estudio, definiendo los objetivos principales, las contribuciones logradas durante la investigación, y detallando la estructura del resto del documento.

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 gran 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. Objetivos

La propuesta principal consiste en el desarrollo de una infraestructura de soporte para el análisis de aplicaciones de cómputo de altas prestaciones (*HPC*, por las siglas en inglés de *High Performance Computing*). Este trabajo se realiza como extensión al trabajo final *Herramientas para el Soporte de Análisis de Rendimiento* de la Especialización en Cómputo de Altas Prestaciones y Tecnología Grid.

La infraestructura desarrollada implementa un procedimiento de análisis de rendimiento ejecutando pruebas de referencia, herramientas de perfil de rendimiento y graficación de resultados. La infraestructura genera como etapa final un informe detallado que soporta la tarea de optimización con información cuantitativa. El reporte final incluye datos estadísticos de la aplicación y el sistema donde se ejecuta, además de gráficos de desviación de resultados, escalamiento de problema y cómputo, e identificación de cuellos de botella.

1.3. Contribuciones

La siguiente lista enumera las diferentes publicaciones realizadas durante el cursado del magister y el desarrollo de la tesis.

1. Estudio de Multiplicación de Matrices. Reporte Técnico. Realizado como parte del curso *Programación en Clusters* dictado por el Dr *Fernando Tinetti* [1].
2. Articulo *Optimizing Latency in Beowulf Clusters*. HPC Latam 2012 [3].
3. Comparación de Implementaciones de una Operación BLAS. Reporte Técnico. Realizado como parte del curso *Programación GPU de Propósito General* dictado por la Dra *Margarita Amor*.
4. Sección *Intel Cluster Ready* e *Intel Cluster Checker* en el libro *Programming Intel Xeon Phi*. Intel Press. 2013 [4].
5. Reseña del Libro *Intel Xeon Phi Coprocessor High Performance Programming* - JCS&T Vol 13 N 2 Octubre 2013 [1].
6. Articulo *Lessons learned from Contrasting BLAS Kernel Implementations* - XIII Workshop Procesamiento Distribuido y Paralelo (WPDP), 2013 [5].

1.4. Metodología

En base al problema y a los objetivos establecidos previamente, la metodología adoptada es la siguiente:

1. Analizar el estado del arte del análisis de rendimiento y las herramientas utlizadas para ello.
2. Formular el problema específico a resolver.
3. Dado una propuesta de procedimiento sistemático de análisis de rendimiento, automatizarlo y analizar potenciales mejoras de utilidad.
4. Identificar que tipos de gráficos y tablas pueden resumir la información obtenida de modo de facilitar su utilización.
5. Aplicar la infraestructura sobre núcleos de cómputo conocidos para poder focalizar los esfuerzos en la mejora del reporte.
6. Documentación de la experiencia

1.5. Estructura

El resto del documento se estructura de la siguiente forma:

[1] JCS&T Vol. 13 No. 2

- Capitulo 2: revisa el estado del arte de los temas incluidos.
- Capitulo 3: describe el problema a resolver.
- Capitulo 4: muestra la propuesta de solución.
- Capitulo 5: aplica la solución a casos de estudio.
- Capítulo 6: concluye reflejando los objetivos y proponiendo trabajo futuro.

Capítulo 2

Estado del Arte

Este capítulo revisa el estado del arte de los temas relevantes a este trabajo.

2.1. Análisis de Rendimiento

Este capítulo introduce el concepto de rendimiento y teoría básica sobre su análisis. Además revisa las herramientas disponibles.

2.1.1. 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.

2.1.2. 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.

Ley de *Amdahl*

La ley de *Amdahl* [6] 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 existe un límite en la práctica.

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

En el caso de conocer los tiempos de ejecución para distinto número de procesadores, la porción serial/paralelo puede ser aproximada.

Ley de *Gustafson*

Desde un punto de vista más general, la ley de *Gustafson* [7] (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 $7x$. En el otro extremo, nos estima un incremento de $61x$ en el caso de tener 95 % de paralelismo.

Métrica de *Karp-Flatt*

Esta métrica es utilizada para medir el grado de paralelismo de una aplicación [8]. Su valor nos permite rápidamente dimensionar 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.

2.1.3. 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.

2.1.4. 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.

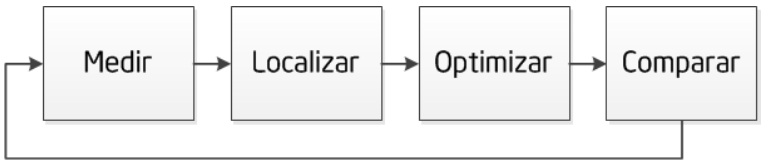

Figura 2.1: Optimización Iterativa

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 suele 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.

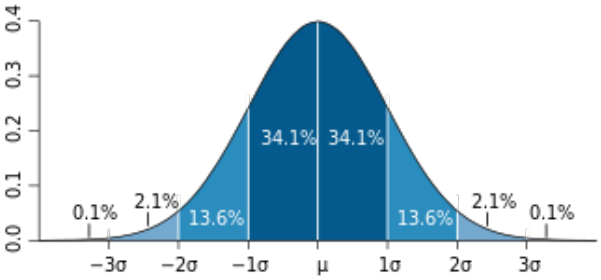

Figura 2.2: Desviación de valores en una distribución normal [Wikipedia]

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 [9], 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* [10]. En cambio se suele tomar el anti-logaritmo del promedio de los logaritmos de los valores siguiendo la ecuación 2.5.

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

2.2. Herramientas

Actualmente existen numerosas y diversas herramientas para el análisis de rendimiento [11]. 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 2.3.

Tabla 2.3: Herramientas de Soporte para Optimización

Ejemplo	**Descripción**
Software `gprof`	Muestra información de perfil de llamadas a funciones
Hardware `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.

En este trabajo se ha limitado el alcance a herramientas de código abierto.

2.2.1. Pruebas de Rendimiento

Para medir el rendimiento se utilizan pruebas de referencia (en inglés, *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 2.4), y posteriormente algunos detalles específicos e instancias de sus datos de salida para ser utilizados a manera de ejemplo.

Tabla 2.4: 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.

STREAM

STREAM [12] 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 2.5.

Tabla 2.5: 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
```

Linpack

Linpack [13] es un conjunto de subrutinas *FORTRAN* que resuelven problemas de álgebra lineal como ecuaciones lineales y multiplicación de matrices. High Performance Linpack (HPL) [14] 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 2.3) 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.

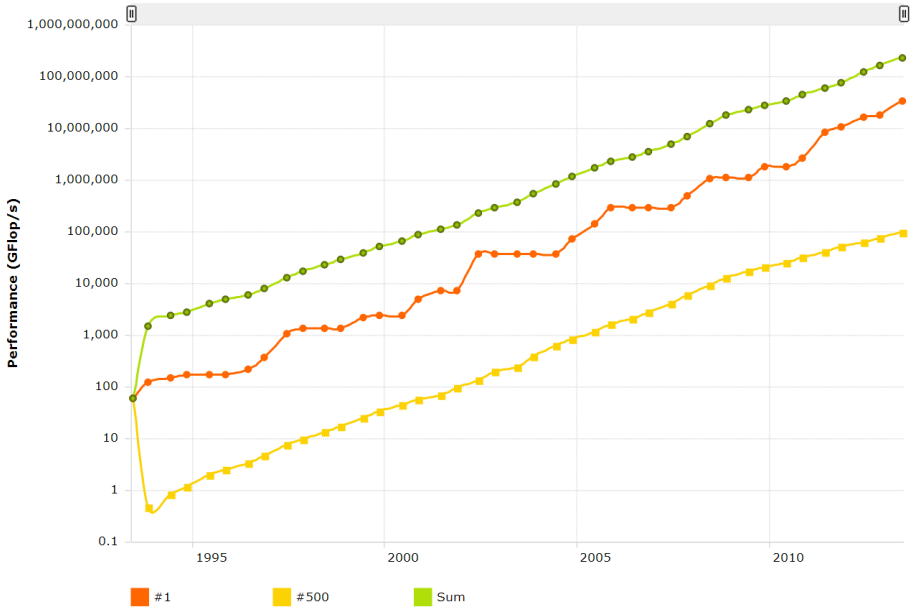

Figura 2.3: Rendimiento Agregado del Top500 [Top500])

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.

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) [15]. 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. [3].

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.

```
# 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
```

HPC Challenge

El *benchmark* HPC Challenge [16] (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 2.4. 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.

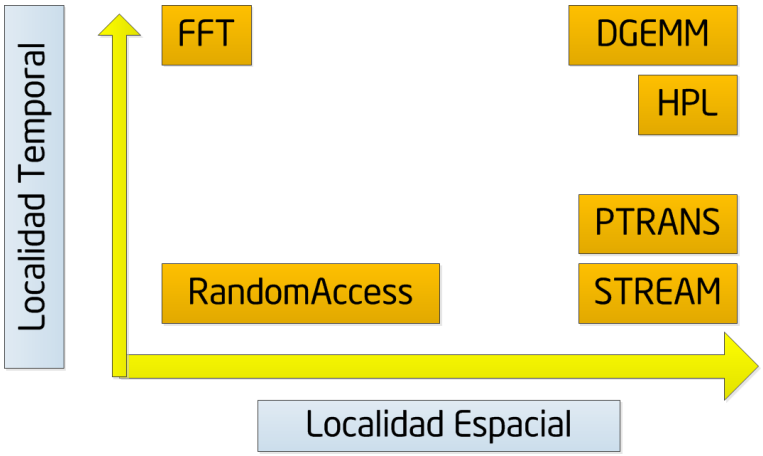

Figura 2.4: Localidad temporal versus espacial en resultados de HPCC

Para una mejor comparación de resultados de HPCC se utilizan diagramas denominados *kiviats*, un ejemplo se muestra en la Figura 2.5. 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.

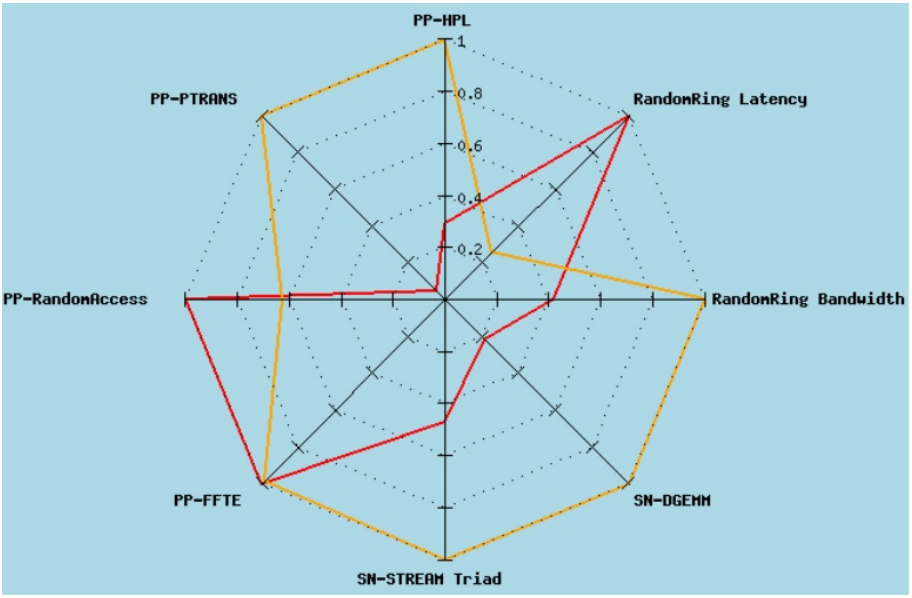

Figura 2.5: Diagrama Kiviat [Top500]

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

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

2.2.2. Utilización de las 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 partes específicas del código. 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 2.6.

Tabla 2.6: 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].

2.2.3. Tiempo de Ejecución

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

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`. Un ejemplo se demuestra en el listado 1.

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

Listado 1: Ejecución del Programa

```
$ /usr/bin/time -v ./program
1
        Command being timed: "./program"
        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
```

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 código en el listado 2 ejemplifica como obtener un número de segundos en una representación de punto flotante de doble precisión, permitiendo una granularidad de medición adecuada.

Listado 2: Tiempo de Ejecución

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

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

2.2.4. 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.

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 en el listado 3 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 [17].

Listado 3: Compilación con Información de Depuración

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

En el listado 4 se muestra la información de las funciones del programa ordenadas por mayor impacto en el tiempo de ejecución.

Listado 4: Perfil de Rendimiento

```
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)
...
```

En el listado 5 se muestra la información del gráfico de llamadas del programa.

Listado 5: Gráficos de Llamadas

```
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]
--
```

2.2.5. 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.

Ejemplo: `perf`

A continuación en el listado 6 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.

Listado 6: Estadísticas de Contadores

```
$ 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
```

En el listado 7 se muestra la salida del perfil de rendimiento. Notar que se incluye incluso la información del comportamiento del núcleo del sistema.

Listado 7: Perfil de Rendimiento

```
$ 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
...
```

En el listado 8 se muestra la salida del perfil de código anotado con las instrucciones del ensamblador respectivas.

Listado 8: Código Anotado

```
 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 [18] e información de los fabricantes [19].

2.2.6. 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. En el listado 9 se muestra la información incluída.

Listado 9: Información de Vectorización

```
$ 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.
```

```
8  ex.c:3: note: vectorized 1 loops in function.
9  $ gcc -c -O3 -fdump-tree-vect-details ex.c
10 ...
```

En el caso de existir código recursivo, podemos comprobar que no suele estar soportado por los compiladores actuales. La información de vectorización sobre un código que posee recursividad se muestra en el listado 10

Listado 10: Vectorización de Código Recursivo

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

Capítulo 3

Descripción del Problema

Este capítulo introduce el problema a resolver.

3.1. Análisis de Rendimiento

Este trabajo entonces trata de simplificar la tarea de análisis de rendimiento.

Aunque es una tarea difícil de sistematizar y que muchas veces depende fuertemente de la aplicación bajo análisis, se propone un procedimiento general para ciertas aplicaciones. El análisis de rendimiento requiere la utilización de diversas herramientas, las cuales implican un tiempo no trivial de aprendizaje de su correcta aplicación.

El problema concreto sobre el que se trabaja es poseer automatización para ahorrar esfuerzo y minimizar el nivel de conocimiento requerido durante el desarrollo de aplicaciones de cómputo de altas prestaciones.

3.2. Infrastructura de Soporte

Los siguientes requerimientos son los necesarios para una infrastructura:

3.2.1. Reusabilidad

La infastructura debe ser aplicable a un gran rango de programas, no requiriendo su modificación. Su instalación solo debe depender de la existencia previa de las mismas herramientas que un usuario podría ejecutar para obtener información relacionada al rendimiento de un programa. La infrastructura debe ser eficiente, no debe requerir ejecutar pruebas largas y tediosas cuando la información puede ser reusada.

3.2.2. Configurabilidad

La aplicación de las herramientas debe ser configurable en su totalidad mediante un archivo de configuración, evitando tareas manuales y repetitivas donde usualmente se comenten errores que causan pérdidas de tiempo valiosas. Los

parámetros a configurar deberían incluir entre otros: la forma de compilar y ejecutar un programa, el rango de valores de entrada, el número de repeticiones de las diferentes ejecuciones del programa.

3.2.3. Portabilidad

La infrastructura debe ser implementada con un lenguaje portable, de ser posible basado en scripts de código abierto de forma que puedan ser revisado fácilmente por los usuarios para incluir nuevas fuentes de información o cambiar la forma en que las herramientas base son utilizadas.

3.2.4. Extensibilidad

La infrastructura debe poseer un diseño de fácil extensión, la incorporación de nuevas herramientas, gráficos o secciones dentro de un reporte debe ser una tarea trivial asumiendo que ya se conoce la forma de obtener la información requerida para ello.

3.2.5. Simplicidad

La infrastructura debe reutilizar las mismas herramientas disponibles en el sistema, de tal forma el usuario puede continuar el analisis de forma directa. También debe generar archivos de soporte con la información pura de los comandos ejecutados y su salida sin depurar. Debe ser posible completar un reporte entre un día de trabajo y otro sin la interacción con un usuario.

Capítulo 4

Propuesta de Solución

Este capítulo muestra la propuesta de solución, incluyendo el diseño de la misma a diferentes niveles, un ejemplo de su reporte de salida y realizando casos de aplicación con aplicaciones del mundo real.

4.1. Procedimiento

La Figura 4.1 muestra a grandes rasgos las etapas del proceso a automatizar.

Figura 4.1: Procedimiento de Análisis

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.

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. La infrastructura solo implementa los pasos especificos de ejecución de herramientas para la recolección de información.

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.

Listado 11: Instalación de HPCC

```
$ 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.
3. Establecer cual es el promedio geométrico a usar como referencia para comparaciones futuras.

Listado 12: Estabilidad de Resultados

```
$ for i in `seq 1 32`; do /usr/bin/time -v ./program >> 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?

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

Listado 13: Escalamiento de Problema

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

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

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

Listado 14: Escalamiento de Cómputo

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

6. 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.

Listado 15: Generación de Perfil de Rendimiento

```
$ gcc -g -pg program.c -o program
$ ./program
...
$ gprof --flat-profile --graph --annotated-source app
...
```

a) ¿Cómo está 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?

7. Utilizar el profiler a nivel de sistema

Listado 16: Generación de Perfil de Sistema

```
$ prof stat ./program
$ prof record ./program
$ 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?

8. Comprobar vectorizaciones

Listado 17: Información de Vectorización

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

a) ¿Hay ciclos que no pueden ser automáticamente vectorizados?
b) ¿Pueden los ciclos no optimizados ser modificados?

4.3. Infraestructura

El procedimiento anterior se implementó como una infrastructura automática de generacion de reportes de rendimiento denominada `hotspot` [1].

La automatización se comporta del mismo modo que un usuario realizando un procedimiento sistemático de analisis de rendimiento. *Latex* se utiliza para la generación del reporte final. Es decir que ejecuta utilizades del sistema como `gcc`, `make`, `prof`, `gprof`, `pidstat` para obtener la información relevante.

[1] El proyecto `hotspot` está disponible en https://github.com/moreandres/hotspot

Las limitaciones *a-priori* que posee la infrastructura son en materia de portabilidad y aplicación. Por el lado de la portabilidad, solo se soportan sistemas GNU/Linux recientes, siendo necesarias el conjunto de utilidades de soporte especificadas anteriormente. Por el lado de la aplicación, solo se soportan programan utilizando tecnología OpenMP multi-hilo.

4.4. Teoría de Operación

Esta sección detalla el funcionamiento de la infraestructura relacionando los componentes.

4.4.1. Configuración

La herramienta de línea de comando toma como parámetro principal el binario de la aplicación a analizar. Asume por defecto que la aplicación esta construida utilizando *Makefile* de la forma usual, ya que para algunos pasos de análisis se necesita recompilar la aplicación con información extra de depuración. En el caso de que no sea así, el archivo de configuración permite la definición de tareas como configuración, compilacion y ejecución del programa.

El listado 18 muestra la pantalla de ayuda de la infrastructura.

Listado 18: Ayuda de `hotspot`

```
$ hotspot --help
usage: hotspot [-h] [-v] [--config CONFIG] [--debug]

Generate performance report for OpenMP programs.

optional arguments:
  -h, --help            show this help message and exit
  -v, --version         show program's version number and exit
  --config CONFIG, -c CONFIG
                        path to configuration
  --debug, -d           enable verbose logging

Check https://github.com/moreandres/hotspot for details.
$ hotspot
```

El archivo de configuración requerido se muestra en el listado 19. Un parámetro necesario es el rango de tamaños de problema a utilizar durante las ejecuciones, definidos como una secuencia de formato compatible con `seq(1)`.

Listado 19: Configuración de `hotspot`

```
# hotspot configuration file

[hotspot]
# python format method is used to pass parameters

# range is a seq-like definition for problem size
range=1024,2048,256

# cflags are the compiler flags to use when building
cflags=-O3 -Wall -Wextra

# build is the command used to build the program
build=CFLAGS='{0}' make

# clean is the cleanup command to execute
```

```
16 clean=make clean
17
18 # run is the program execution command
19 run=OMP_NUM_THREADS={0} N={1} ./{2}
20
21 # count is the number of runs to check workload stabilization
22 count=16
```

4.4.2. Arquitectura

A grandes rasgos el usuario utiliza la infrastructura para analizar un programa. La infrastructura ejercita el programa multiples veces hasta obtener la información necesaria para generar un reporte.

Esto se detalla en la Figura 4.2.

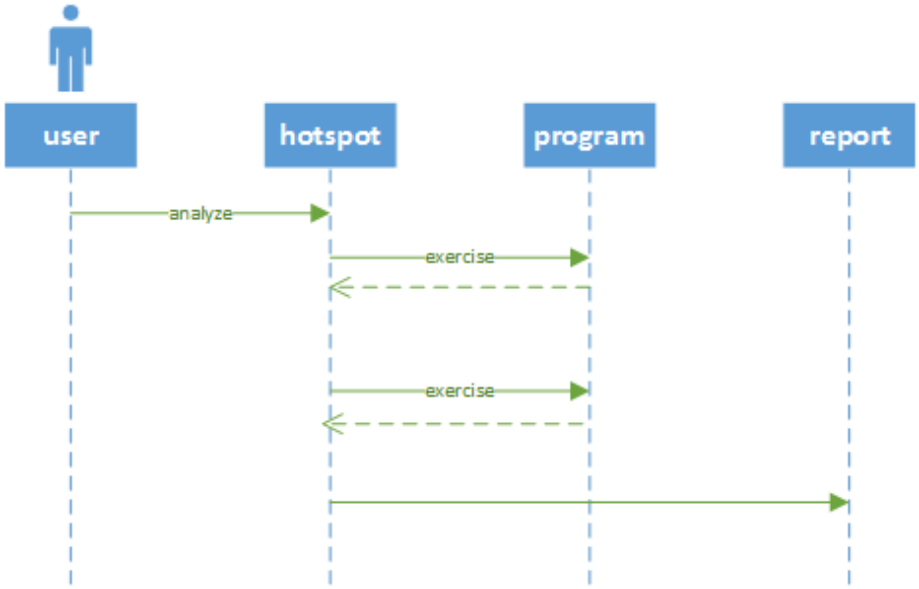

Figura 4.2: Diagrama de Secuencia

4.4.3. Funcionamiento Interno

Se utiliza un directorio escondido dentro de la carpeta que contiene la aplicación y se guardan en sub-directorios por fecha y hora las diferentes ejecuciones. Esta información es luego usada para comparación histórica de resultados.

Inicialmente se ejecuta la aplicación múltiples veces para validar que los resultados poseen una desviación saludable. Se resume esta información con un histograma y se hace una aproximación a una distribución de resultados normales como comparación. Se toma como referencia la media geométrica de los resultados. La primer ejecución se descarta.

Se ejercita la aplicación dentro del rango de tamaño de problema, por cada punto en el rango se ejecuta múltiples veces para promediar. Se grafican los

resultados en un gráfico, posiblemente con escala logarítmica. Se sobrepone una curva ideal suponiendo que el doble de tamaño va a necesitar el doble de tiempo de cómputo. Se sobrepone también la información más vieja disponible como mecanismo de validación de las optimizaciones.

Se detecta cuantas unidades de procesamiento hay en el sistema. Se utiliza una y luego se itera hasta utilizar todas, se ejecuta múltiples veces la aplicación y se promedia el resultado. Se sobrepone una curva ideal suponiendo *speedup* ideal. Se sobrepone también la información más vieja disponible como mecanismo de validación de las optimizaciones.

Utilizando la información anterior, se calcula porcentaje de ejecución en serie y en paralelo. Con esta información se calcula limites según leyes de *Amdalah* y *Gustafson* para los procesadores disponibles y para un número grande como para visualizar que va a pasar con infinitos procesadores.

Se recompila la aplicación con información extra de depuración y se obtiene información del perfil de ejecución. Se detallan las funciones, las líneas de código y el *assembler* implementado por las mismas.

4.5. Diseño de Alto Nivel

El diseño de la infrastructura refleja la interacción manual con el sistema. Se depende de la presencia de herramientas de línea de comando, del programa a analizar en particular, y de LaTeX. Esto se resume en la Figura 4.3

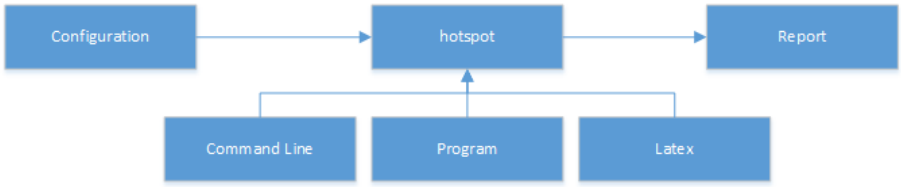

Figura 4.3: Diseño de Alto Nivel

1. **Configuración**: el componente lee información de configuración y la deja disponible para los demás componentes.
2. **Reporte**: el componente guarda valores de variables que luego utiliza para generar el reporte final.
3. **Infrastructura**: este componente utiliza información de configuración para generar un reporte.

4.6. Diseño de Bajo Nivel

La infrastructura se implementa mediante una jerarquía de clases simple, fácil de extender de ser necesario. Las clases se muestran en la Figura 4.4.

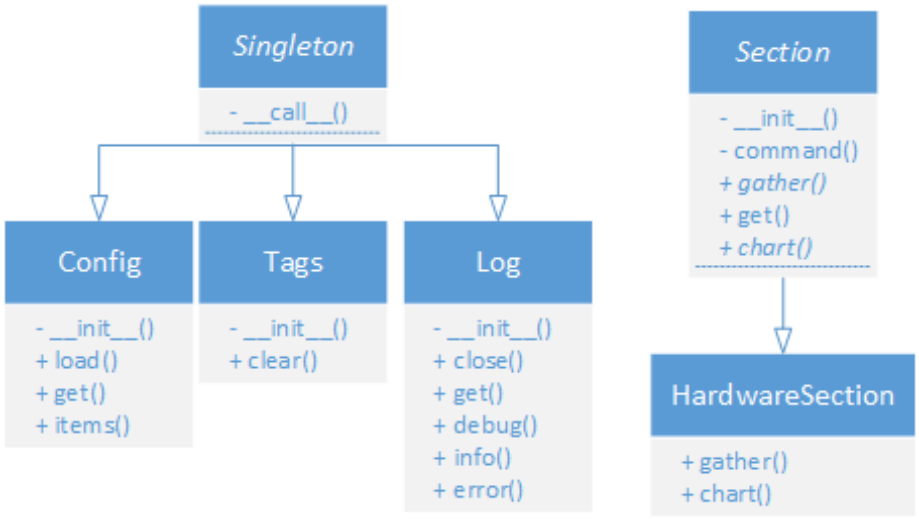

Figura 4.4: Diseño de Bajo Nivel

Aunque solo se incluye *HardwareSection* como ejemplo, existe un objeto sección por cada sección dentro del reporte.

1. *Singleton*: Patrón de Instancia Única. Es utilizado por otras clases para garantizar que solo existe una instancia única.
2. *Tags*: Almacenamiento de Palabras Clave. Es utilizado para guardar palabras clave que son utilizadas para generar el reporte.
3. *Log*: Generador de Mensajes. Es utilizado para estructurar los mensajes durante la ejecución.
4. *Config*: Administración de la Configuracion. Es utilizado para leer y consultar la configuración de cada programación.
5. *Section*: Sección del Reporte. Es utilizado como base de otras secciones.
6. *HardwareSection*: Descripción del *Hardware*. Es utilizado para obtener información del *hardware* disponible.
7. *ProgramSection*: Detalles sobre el Programa. Es utilizado para obtener información del programa a analizar.
8. *SoftwareSection*: Descripción del *Software*, Es utilizado para obtener información del *software* disponible.
9. *SanitySection*: Chequeo Base. Es utilizado para comprobaciones básicas del programa.
10. *BenchmarkSection*: Pruebas de Rendimiento, Es utilizada para obtener información de rendimiento.
11. *WorkloadSection*: Caso de Prueba. Es utilizada para obtener información del caso de prueba.
12. *ScalingSection*: Escalamiento de Problema. Es utilizada para entender como escala el tamaño del problema.
13. *ThreadsSection*: Hilos. Es utilizada para entender como escalan los hilos.
14. *OptimizationSection*: Optimización. Es utilizada para entender los diferentes niveles de optimización del compilador.
15. *ProfileSection*: Sección sobre el Perfil de Rendimiento. Permite entender

en que partes del programa se invierte el tiempo de ejecución.
16. *ResourcesSection*: Utilización de Recursos. Permite entender como se utilizan los recursos del sistema.
17. *AnnotatedSection*: Código Anotado. Incluye código ensamblador.
18. *VectorizationSection*: Vectorización de Ciclos. Es utilizado para entender cuales ciclos fueron vectorizados o no.
19. *CountersSection*, Información de Contadores de *Hardware*. Es utilizado para resumir la información de los contadores de *hardware*.
20. *ConfigSection*, Configuración. Es utilizado para revisar la configuración del análisis.

4.6.1. Implementación

La implementación se desarrollo sobre una plataforma *GNU/Linux*, utilizando la distribución `Ubuntu 14.04.1 LTS`. El lenguaje utilizado Python 2.7.6, publicado en *Python Software Foundation Package Index* como hotspot versión 0.3. El repositorio de código se encuetra en https://github.com/moreandres/hotspot. La licencia del código fuente es GPLv2. Se reutilizan librerías como `matplotlib.pyplot` [20] y `numpy` [21] para gráficar las estadísticas obtenidas durante las pruebas de rendimiento.

4.7. Reporte Generado

A continuación se explica el contenido de cada sección en particular. En el apéndice se muestran tres ejemplos utilizando núcleos de cómputo conocidos.

4.7.1. Consideraciones Generales

Las consideraciones generales tenidas en cuenta en todas las secciones fueron las siguientes:

1. Formato portable similar a un artículo de investigación.
2. Hipervínculos a archivos con la información pura.
3. Inclusión de una explicación breve del objetivo de la sección y los gráficos incluídos.
4. Inclusión de líneas de tendencía y comportamiento ideal en gráficos.
5. Referencias a material de consulta.
6. Utilización del inglés.

4.7.2. Resumen

El resumen incluído en la primer página introduce el reporte junto con información sobre la infrastructura utilizada y la ubicación de la información de ejecución.

4.7.3. Contenido

Contiene un índice de secciones en formato de dos columnas con hipervinculos para facilitar la búsqueda rápida de la información.

4.7.4. Programa

Se incluyen detalles sobre el programa bajo análisis.

4.7.5. Capacidad del Sistema

Se incluyen detalles sobre la configuración de *hardware* y *software* del sistema. Además se incluye información de referencia obtenida de la prueba de rendimiento *HPCC*.

4.7.6. Carga de Trabajo

Se incluye información sobre el caso de prueba ejecutado. El tamaño del mismo en memoria y la composición de sus estructuras.

Se incluye un resumen estadistico de los tiempos de ejecución luego de repetir el caso de pruebas varias veces. Un histograma muestra la distribución de los resultados.

Se incluye un gráfico demostrando los tiempos de ejecución del caso de prueba bajo diferentes niveles de optimización en el compilador.

4.7.7. Escalabilidad

Se incluyen gráficos resumiendo la escalabilidad del programa al aumento el tamaño del problema y también por separado la cantidad de unidades de cómputo.

Se estima la proporción paralelo/lineal del programa, y junto con ellas se calculan mejoras teóricas máximas bajo las leyes de *Amdalah* y *Gustafson*.

4.7.8. Perfil de Ejecución

Se incluye información sobre las funciones y líneas de código más usadas.

Se incluyen gráficos sobre el uso de los recursos del sistema durante una ejecución del programa. Incluyendo utilización de CPU, memoria y lectura/escritura de disco.

Se listan cuellos de botella, junto con el código de ensamblador respectivo.

4.7.9. Bajo Nivel

Se incluye el reporte de vectorización emitido por el compilador.

Se incluye el reporte de contadores de *hardware* relacionados con el rendimiento.

4.7.10. Referencias

Se incluye una lista de publicaciones relacionadas, las cuales son referenciadas en las secciones anteriores.

Capítulo 5

Casos de Aplicación

Este capítulo contiene casos de estudio mostrando los resultados de aplicar la infraestructura desarrollada a aplicaciones de cómputo de alto rendimiento conocidas.

5.1. Código de Prueba

Esta sección incluye los resultados obtenidos al aplicar la infrastructura a núcleos de cómputo de prueba.

5.1.1. Matrix

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]

El reporte generado en bruto puede consultarse en el apéndice A.

5.1.2. Heat2d

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].

El reporte generado en bruto puede consultarse en el apéndice A.

5.1.3. Ocho Reinas

El programa resuelve 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 misma diagonal, fila o columna.

[1] http://blog.speedgocomputing.com/2010/08/parallelizing-matrix-multiplication.html.
[2] http://www.rblasch.org/studies/cs580/pa5/#Source+Code-N100AC.

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

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

Capítulo 6

Conclusiones y Trabajo Futuro

Este capítulo concluye revisando los objetivos propuestos y posibles líneas de investigación como continuación.

6.1. Conclusiones

Se desarrollo como esta planeado una infrastructura de soporte que permite a un desarrollador especialista en el dominio de un problema obtener rapidamente información cuantitativa del comportamiento de un programa OpenMP, incluyendo utilización del recurso y cuellos de botella.

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.

6.2. Trabajo Futuro

Las posibilidades de extensión de esta infrastructura son muchas, cada sección del reporte final puede incluir información más detallada o utilizar otras herramientas. En particular la sección de contadores de *hardware* solo contiene una lista de contadores generalizada para cualquier arquitectura; si se supone una arquitectura dada se puede proveer información más especifica y por lo tanto más útil.

Otra posibilidad consiste en incorporar soporte para programas basados sobre la libreria de comunicacion MPI. Una vez que los programas son optimizados para ejecutarse eficientemente sobre una arquitectura multicore, el siguiente paso consiste en utilizar varios sistemas multicore al unísono.

Las herramientas y procedimientos ya integrados solo utilizan el programa ya compilado para obtener información. Existen tecnicas de analisis de rendimiento o de calidad de software que necesitan acceso al codigo fuente. Analisis estatico de código. Ejecucion simbólica. cppcheck. clint. klocwork. coverity.

Por último, aunque la generación de un documento portable en formate PDF es simple y útil, se puede pensar en utilizar una tecnología que permita la generacion de reportes dinámicos que permitan a los expertos extraer información. Una implementación en HTML5 utilizando tablas filtrables y gráficos configurables puede ser interesante y causar gran impacto.

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] Rafael Garabato, Andrés More, and Victor Rosales. Optimizing latency in beowulf clusters. *CLEI Electronic Journal*, 15(3):3–3, 2012.

[4] James Jeffers and James Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

[5] Andres More. Lessons learned from contrasting a blas kernel implementations. In *XVIII Congreso Argentino de Ciencias de la Computacion*, 2013.

[6] 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.

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

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

[9] 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.

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

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

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

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

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

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

[16] 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.

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

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

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

[20] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[21] Stefan van der Walt, S. Chris Colbert, and Gael Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science and Engg.*, 13(2):22–30, March 2011.

Apéndice A

Reporte de Ejemplo

A.1. Multiplicación de Matrices

`matrix` Performance Report

20140913-174714

Abstract

This performance report is intended to support performance analysis and optimization activities. It includes details on program behavior, system configuration and capabilities, including support data from well-known performance analysis tools.

This report was generated using `hotspot` version `0.1`. Homepage `http://www.github.com/moreandres/hotspot`. Full execution log can be found at `~/.hotspot/matrix/20140913-174714/hotspot.log`.

Contents

1 Program

This section provides details about the program being analyzed.

1. Program: `matrix`.
   Program is the name of the program.
2. Timestamp: `20140913-174714`.
   Timestamp is a unique identifier used to store information on disk.
3. Parameters Range: `[1024, 1280, 1536, 1792]`.
   Parameters range is the problem size set used to scale the program.

2 System Capabilities

This section provides details about the system being used for the analysis.

2.1 System Configuration

This subsection provides details about the system configuration.

The hardware in the system is summarized using a hardware lister utility. It reports exact memory configuration, firmware version, mainboard configuration, CPU version and speed, cache configuration, bus speed and others.

The hardware configuration can be used to contrast the system capabilities to well-known benchmarks results on similar systems.

```
memory      7985MiB System memory
processor   Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz
bridge      440FX - 82441FX PMC [Natoma]
bridge      82371SB PIIX3 ISA [Natoma/Triton II]
storage     82371AB/EB/MB PIIX4 IDE
network     82540EM Gigabit Ethernet Controller
bridge      82371AB/EB/MB PIIX4 ACPI
storage     82801HM/HEM (ICH8M/ICH8M-E) SATA Controller [AHCI mode]
```

The software in the system is summarized using the GNU/Linux platform string.

```
Linux-3.13.0-32-generic-x86_64-with-Ubuntu-14.04-trusty
```

The software toolchain is built upon the following components.

1. Host: `ubuntu`

2. Distribution: `Ubuntu, 14.04, trusty`.
   This codename provides LSB (Linux Standard Base) and distribution-specific information.
3. Compiler: `gcc (Ubuntu 4.8.2-19ubuntu1) 4.8.2`.
   Version number of the compiler program.
4. C Library: `GNU C Library (Ubuntu EGLIBC 2.19-0ubuntu6.3) stable release version 2.19`.
   Version number of the C library.

The software configuration can be used to contrast the system capabilities to well-known benchmark results on similar systems.

2.2 System Performance Baseline

This subsection provides details about the system capabilities.

A set of performance results is included as a reference to contrast systems and to verify hardware capabilities using well-known synthetic benchmarks.

The HPC Challenge benchmark [1] consists of different tests:

1. HPL: the Linpack TPP benchmark which measures the floating point rate of execution for solving a linear system of equations.
2. DGEMM: measures the floating point rate of execution of double precision real matrix-matrix multiplication.
3. PTRANS (parallel matrix transpose): exercises the communications where pairs of processors communicate with each other simultaneously.
4. RandomAccess: measures the rate of integer random updates of memory (GUPS).
5. STREAM: a simple synthetic benchmark program that measures sustainable memory bandwidth (in GB/s).
6. FFT: measures the floating point rate of execution of double precision complex one-dimensional Discrete Fourier Transform (DFT).

Table 1: Benchmarks

Benchmark	Value	Unit
hpl	0.00346363 TFlops	tflops
dgemm	0.916768 GFlops	mflops
ptrans	0.754392 GBs	MB/s
random	0.0253596 GUPs	MB/s
stream	4.15247 MBs	MB/s
fft	1.14658 GFlops	MB/s

Most programs will have a dominant compute kernel that can be approximated by the ones above, the results helps to understand the available capacity.

3 Workload

This section provides details about the workload behavior.

3.1 Workload Footprint

The workload footprint impacts on memory hierarchy usage.

```
matrix: ELF 64-bit LSB  executable, x86-64, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux
```

Binaries should be stripped to better fit inside cache.

```
struct _IO_FILE {
int                        _flags;              /*     0     4 */

/* XXX 4 bytes hole, try to pack */

char *                     _IO_read_ptr;        /*     8     8 */
char *                     _IO_read_end;        /*    16     8 */
char *                     _IO_read_base;       /*    24     8 */
char *                     _IO_write_base;      /*    32     8 */
char *                     _IO_write_ptr;       /*    40     8 */
char *                     _IO_write_end;       /*    48     8 */
char *                     _IO_buf_base;        /*    56     8 */
/* --- cacheline 1 boundary (64 bytes) --- */
char *                     _IO_buf_end;         /*    64     8 */
char *                     _IO_save_base;       /*    72     8 */
char *                     _IO_backup_base;     /*    80     8 */
```

```
char *                    _IO_save_end;         /*    88     8 */
struct _IO_marker *       _markers;             /*    96     8 */
struct _IO_FILE *         _chain;               /*   104     8 */
int                       _fileno;              /*   112     4 */
int                       _flags2;              /*   116     4 */
__off_t                   _old_offset;          /*   120     8 */
/* --- cacheline 2 boundary (128 bytes) --- */
short unsigned int        _cur_column;          /*   128     2 */
signed char               _vtable_offset;       /*   130     1 */
char                      _shortbuf[1];         /*   131     1 */

/* XXX 4 bytes hole, try to pack */

_IO_lock_t *              _lock;                /*   136     8 */
__off64_t                 _offset;              /*   144     8 */
void *                    __pad1;               /*   152     8 */
void *                    __pad2;               /*   160     8 */
void *                    __pad3;               /*   168     8 */
void *                    __pad4;               /*   176     8 */
size_t                    __pad5;               /*   184     8 */
/* --- cacheline 3 boundary (192 bytes) --- */
int                       _mode;                /*   192     4 */
char                      _unused2[20];         /*   196    20 */

/* size: 216, cachelines: 4, members: 29 */
/* sum members: 208, holes: 2, sum holes: 8 */
/* last cacheline: 24 bytes */
};
struct _IO_marker {
struct _IO_marker *       _next;                /*     0     8 */
struct _IO_FILE *         _sbuf;                /*     8     8 */
int                       _pos;                 /*    16     4 */

/* size: 24, cachelines: 1, members: 3 */
/* padding: 4 */
/* last cacheline: 24 bytes */
};
```

Showing layout of data structures. Reorganizing such data to remove alignment holes. Improve CPU cache utilization.

More information `sevendwarves`https://www.kernel.org/doc/ols/2007/ols2007v2-pages-35-44.pdf

3.2 Workload Stability

This subsection provides details about workload stability.

1. Execution time:
   (a) problem size range: `1024 - 2048`
   (b) geomean: `5.54788` seconds
   (c) average: `5.55018` seconds
   (d) stddev: `0.15965`
   (e) min: `5.32458` seconds
   (f) max: `5.76582` seconds
   (g) repetitions: `8` times

   The histogram plots the elapsed times and shows how they fit in a normal distribution sample.

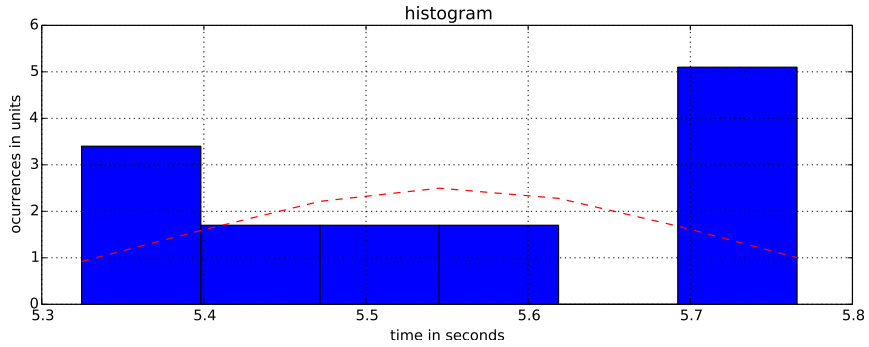

Figure 1: Results Distribution

The workload should run for at least one minute to fully utilize system resources. The execution time of the workload should be stable and the standard deviation less than 3 units.

3.3 Workload Optimization

This section shows how the program reacts to different optimization levels.

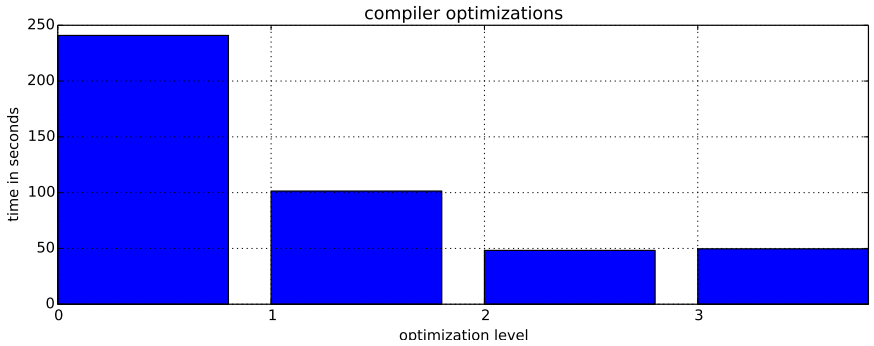

Figure 2: Optimization Levels

4 Scalability

This section provides details about the scaling behavior of the program.

4.1 Problem Size Scalability

A chart with the execution time when scaling the problem size.

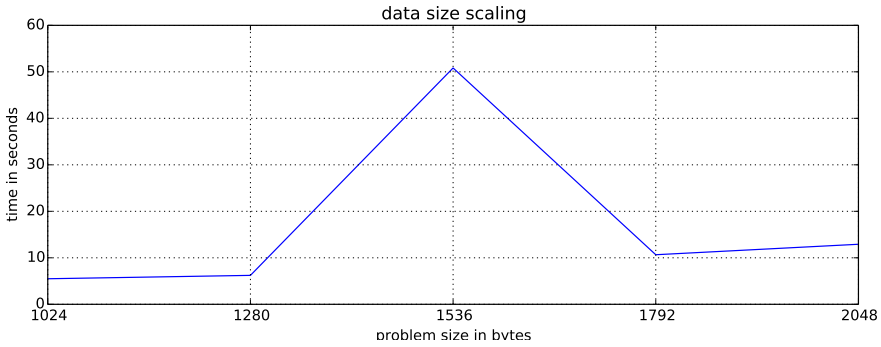

Figure 3: Problem size times

The chart will show how computing time increases when increasing problem size. There should be no valleys or bumps if processing properly balanced across computational units.

4.2 Computing Scalability

A chart with the execution time when scaling computation units.

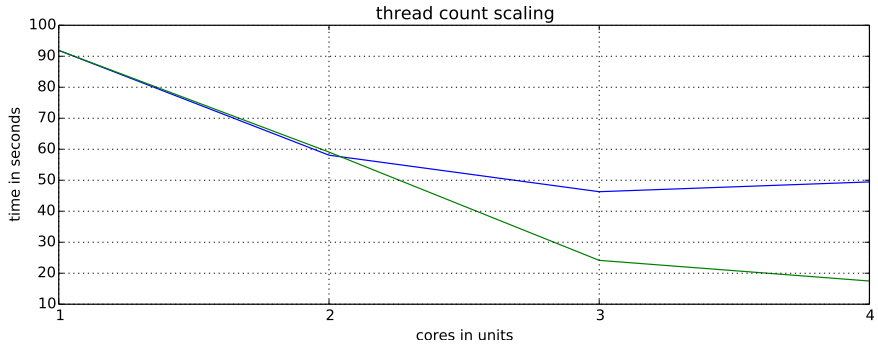

Figure 4: Thread count times

The chart will show how computing time decreases when increasing processing units. An ideal scaling line is provided for comparison.

The parallel and serial fractions of the program can be estimated using the information above.

1. Parallel Fraction: `0.73575`.
   Portion of the program doing parallel work.
2. Serial: `0.26425`.
   Portion of the program doing serial work.

Optimization limits can be estimated using scaling laws.

1. Amdalah Law for 1024 procs: `3.78433 times`.
   Optimizations are limited up to this point when scaling problem size. [**?**]
2. Gustafson Law for 1024 procs: `753.67484 times`.
   Optimizations are limited up to this point when not scaling problem size. [3]

5 Profile

This section provides details about the execution profile of the program and the system.

5.1 Program Profiling

This subsection provides details about the program execution profile.

5.1.1 Flat Profile

The flat profile shows how much time your program spent in each function, and how many times that function was called.

```
Flat profile:
Each sample counts as 0.01 seconds.
     cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 94.65    152.60   152.60                             main._omp_fn.0 (matrix.c:28 @ 400b48)
  5.66    161.72     9.12                             main._omp_fn.0 (matrix.c:28 @ 400b5b)
```

The table shows where to focus optimization efforts to maximize impact.

5.2 System Profiling

This subsection provide details about the system execution profile.

5.2.1 System Resources Usage

The following charts shows the state of system resources during the execution of the program.

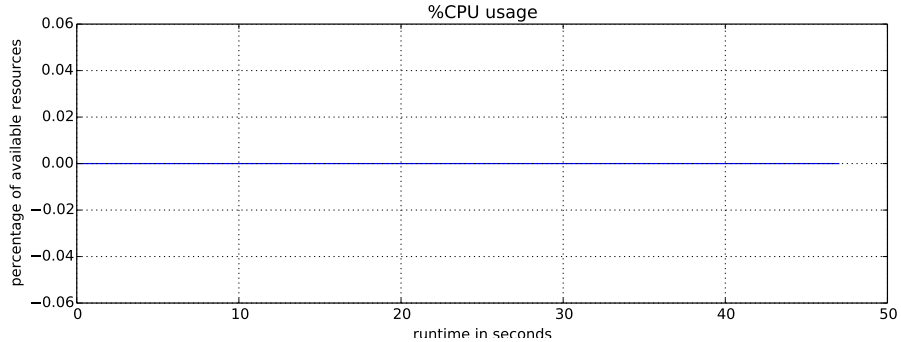

Figure 5: CPU Usage

Note that this chart is likely to show as upper limit a multiple of 100% in case a multicore system is being used.

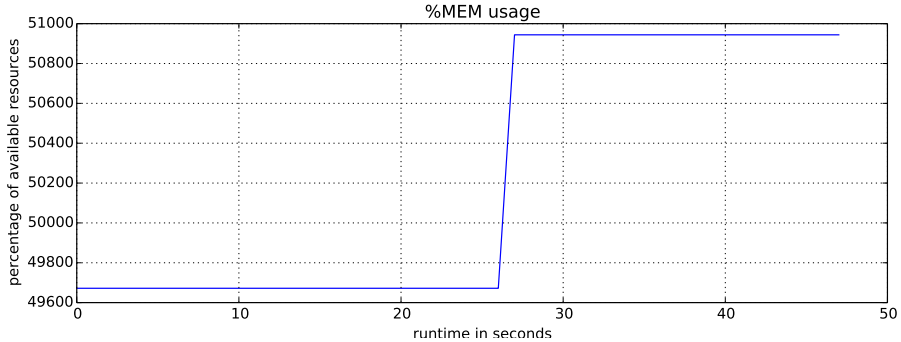

Figure 6: Memory Usage

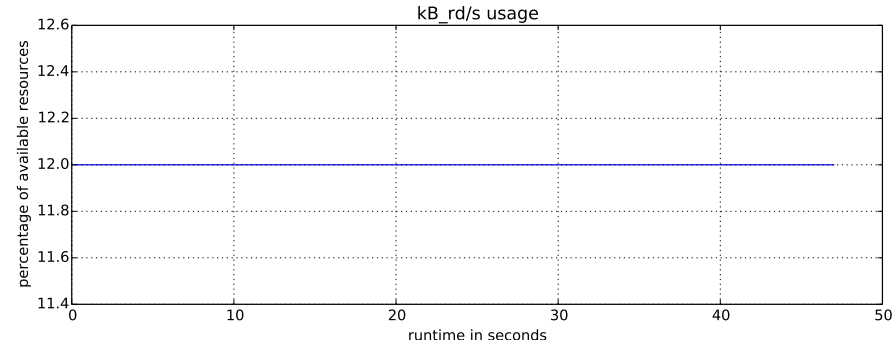

Figure 7: Reads from Disk

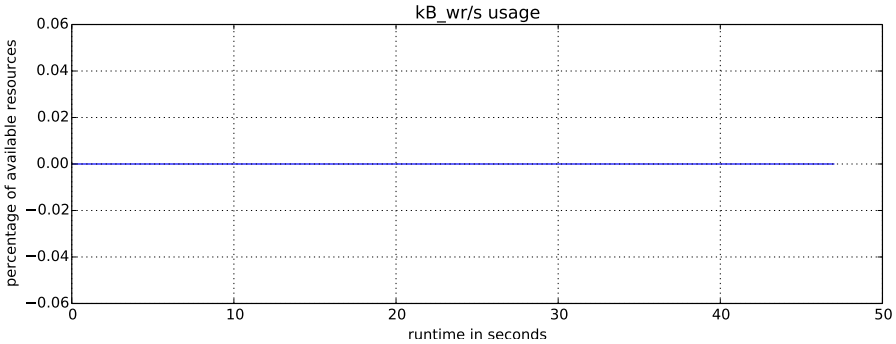

Figure 8: Writes to Disk

5.3 Hotspots

This subsection shows annotated code guiding the optimization efforts.

```
[kernel.kallsyms] with build id fa854edfe97c596cddeac7049b080cbba9de2775 not found, continuing without symbols
         : #pragma omp parallel for shared(a,b,c)
         :   for (i = 0; i < size; ++i) {
         :     for (j = 0; j < size; ++j) {
         :       for (k = 0; k < size; ++k) {
         :         c[i+j*size] += a[i+k*size] * b[k+j*size];
    0.20 :   4009e8:       movss  (rcx),xmm0
   77.73 :   4009ec:       add    r9,rcx
    8.32 :   4009ef:       mulss  (r8,rdx,4),xmm0
    7.20 :   4009f5:       add    $0x1,rdx
         :   }
         : #pragma omp parallel for shared(a,b,c)
         :   for (i = 0; i < size; ++i) {
         :     for (j = 0; j < size; ++j) {
         :       for (k = 0; k < size; ++k) {
         :         c[i+j*size] += a[i+k*size] * b[k+j*size];
    0.86 :   4009fb:       addss  xmm0,xmm1
    4.10 :   4009ff:       movss  xmm1,(rsi)
         :   }
         : #pragma omp parallel for shared(a,b,c)
         :   for (i = 0; i < size; ++i) {
         :     for (j = 0; j < size; ++j) {
         :       for (k = 0; k < size; ++k) {
--
         :   for (i = 0; i < size; ++i) {
         : #include <stdlib.h>
         : #include <stdio.h>
```

```
       : int main()
       : {
  1.01 :   4007b8:       lea    (rdx,rsi,1),edi
       :   for (i = 0; i < size; ++i) {
       :     for (j = 0; j < size; ++j) {
       :       a[i+j*size] = (float) (i + j);
       :       b[i+j*size] = (float) (i - j);
       :   int i, j, k;
       :   for (i = 0; i < size; ++i) {
       :     for (j = 0; j < size; ++j) {
       :       a[i+j*size] = (float) (i + j);
 17.57 :   4007c3:       cvtsi2ss edi,xmm0
  0.34 :   4007c7:       mov    esi,edi
       :   float *c = malloc(sizeof(float) * size * size);
       :   int i, j, k;
       :   for (i = 0; i < size; ++i) {
       :     for (j = 0; j < size; ++j) {
       :       a[i+j*size] = (float) (i + j);
  1.01 :   4007ce:       movss  xmm0,(r10,rcx,1)
       :       b[i+j*size] = (float) (i - j);
 31.76 :   4007d4:       cvtsi2ss edi,xmm0
  1.01 :   4007d8:       movss  xmm0,(r9,rcx,1)
 26.01 :   4007de:       add    r11,rcx
       :   float *c = malloc(sizeof(float) * size * size);
       :   int i, j, k;
       :   for (i = 0; i < size; ++i) {
       :     for (j = 0; j < size; ++j) {
  2.03 :   4007e1:       cmp    edx,ebx
       :   float *b = malloc(sizeof(float) * size * size);
       :   float *c = malloc(sizeof(float) * size * size);
       :   int i, j, k;
       :   for (i = 0; i < size; ++i) {
       :       b[i+j*size] = (float) (i - j);
```

6 Low Level

This section provide details about low level details such as vectorization and performance counters.

6.1 Vectorization Report

This subsection provide details about vectorization status of the program loops.

```
Analyzing loop at matrix.c:26
matrix.c:26: note: not vectorized: multiple nested loops.
matrix.c:26: note: bad loop form.
Analyzing loop at matrix.c:26
matrix.c:26: note: not vectorized: not suitable for strided load _41 = *_40;
matrix.c:26: note: bad data references.
Analyzing loop at matrix.c:27
matrix.c:27: note: step unknown.
matrix.c:27: note: reduction used in loop.
matrix.c:27: note: Unknown def-use cycle pattern.
matrix.c:27: note: Unsupported pattern.
matrix.c:27: note: not vectorized: unsupported use in stmt.
matrix.c:27: note: unexpected pattern.
matrix.c:24: note: vectorized 0 loops in function.
matrix.c:24: note: not consecutive access _10 = .omp_data_i_9(D)->size;
matrix.c:24: note: Failed to SLP the basic block.
matrix.c:24: note: not vectorized: failed to find SLP opportunities in basic block.
matrix.c:24: note: not vectorized: not enough data-refs in basic block.
matrix.c:24: note: not vectorized: not enough data-refs in basic block.
matrix.c:26: note: not vectorized: not enough data-refs in basic block.
matrix.c:24: note: not vectorized: not enough data-refs in basic block.
matrix.c:24: note: not consecutive access .omp_data_i_9(D)->j = .omp_data_i__j_lsm.9_106;
matrix.c:24: note: Failed to SLP the basic block.
matrix.c:24: note: not vectorized: failed to find SLP opportunities in basic block.
matrix.c:24: note: not vectorized: not enough data-refs in basic block.
matrix.c:24: note: Failed to SLP the basic block.
matrix.c:24: note: not vectorized: failed to find SLP opportunities in basic block.
```

```
matrix.c:24: note: not consecutive access pretmp_123 = *pretmp_122;
matrix.c:24: note: Failed to SLP the basic block.
matrix.c:24: note: not vectorized: failed to find SLP opportunities in basic block.
matrix.c:26: note: not vectorized: not enough data-refs in basic block.
matrix.c:24: note: not consecutive access .omp_data_i_9(D)->k = _10;
matrix.c:24: note: Failed to SLP the basic block.
matrix.c:24: note: not vectorized: failed to find SLP opportunities in basic block.
matrix.c:24: note: not vectorized: not enough data-refs in basic block.
matrix.c:24: note: not vectorized: not enough data-refs in basic block.
matrix.c:28: note: can't determine dependence between *_40 and *pretmp_122
matrix.c:28: note: can't determine dependence between *_46 and *pretmp_122
matrix.c:28: note: SLP: step doesn't divide the vector-size.
matrix.c:28: note: Unknown alignment for access: *(pretmp_113 + (sizetype) ((long unsigned int) pretmp_118 * 4))
matrix.c:28: note: not consecutive access _41 = *_40;
matrix.c:28: note: not consecutive access *pretmp_122 = _49;
matrix.c:28: note: Failed to SLP the basic block.
matrix.c:28: note: not vectorized: failed to find SLP opportunities in basic block.
matrix.c:24: note: not vectorized: not enough data-refs in basic block.
matrix.c:24: note: not vectorized: not enough data-refs in basic block.
Analyzing loop at matrix.c:16
matrix.c:16: note: not vectorized: not suitable for strided load *_27 = _29;
matrix.c:16: note: bad data references.
Analyzing loop at matrix.c:17
matrix.c:17: note: not vectorized: not suitable for strided load *_27 = _29;
matrix.c:17: note: bad data references.
matrix.c:4: note: vectorized 0 loops in function.
matrix.c:7: note: not vectorized: not enough data-refs in basic block.
matrix.c:8: note: not vectorized: not enough data-refs in basic block.
matrix.c:4: note: not vectorized: not enough data-refs in basic block.
matrix.c:18: note: not consecutive access *_27 = _29;
matrix.c:18: note: not consecutive access *_32 = _34;
matrix.c:18: note: not consecutive access *_36 = 0.0;
matrix.c:18: note: Failed to SLP the basic block.
matrix.c:18: note: not vectorized: failed to find SLP opportunities in basic block.
matrix.c:16: note: not vectorized: not enough data-refs in basic block.
matrix.c:4: note: not vectorized: not enough data-refs in basic block.
matrix.c:4: note: not vectorized: not enough data-refs in basic block.
matrix.c:24: note: misalign = 0 bytes of ref .omp_data_o.1.c
matrix.c:24: note: misalign = 8 bytes of ref .omp_data_o.1.b
matrix.c:24: note: misalign = 0 bytes of ref .omp_data_o.1.a
matrix.c:24: note: misalign = 8 bytes of ref .omp_data_o.1.size
matrix.c:24: note: misalign = 12 bytes of ref .omp_data_o.1.j
matrix.c:24: note: misalign = 0 bytes of ref .omp_data_o.1.k
matrix.c:24: note: Build SLP failed: unrolling required in basic block SLP
matrix.c:24: note: Build SLP failed: unrolling required in basic block SLP
matrix.c:24: note: Failed to SLP the basic block.
matrix.c:24: note: not vectorized: failed to find SLP opportunities in basic block.
matrix.c:4: note: not vectorized: not enough data-refs in basic block.
matrix.c:10: note: not vectorized: not enough data-refs in basic block.
matrix.c:4: note: not vectorized: not enough data-refs in basic block.
```

The details above shows the list of loops in the program and if they are being vectorized or not. These reports can pinpoint areas where the compiler cannot apply vectorization and related optimizations. It may be possible to modify your code or communicate additional information to the compiler to guide the vectorization and/or optimizations.

6.2 Counters Report

This subsection provides details about software and hardware counters.

```
 Performance counter stats for './matrix' (3 runs):

     187780.108759 task-clock (msec)         #    3.845 CPUs utilized            ( +-  0.58 )
               361 context-switches          #    0.002 K/sec                    ( +-  2.51 )
                 5 cpu-migrations            #    0.000 K/sec                    ( +- 18.90 )
             1,741 page-faults               #    0.009 K/sec                    ( +-  0.02 )
   <not supported> cycles
   <not supported> stalled-cycles-frontend
   <not supported> stalled-cycles-backend
   <not supported> instructions
```

```
<not supported> branches
<not supported> branch-misses

   48.842899674 seconds time elapsed                                          ( +-  0.90 )
```

The details above shows counters that provide low-overhead access to detailed performance information using internal registers of the CPU.

References

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

[2] Amdahl, Gene M., *Validity of the single processor approach to achieving large scale computing capabilities*. Communications of the ACM, Proceedings of the April 18-20, 1967, spring joint computer conference Pages 483-485, 1967.

[3] John L. Gustafson, *Reevaluating Amdahl's Law*. Communications of the ACM, Volume 31 Pages 532-533, 1988.

[4] OpenMP Architecture Review Board, *OpenMP Application Program Interface. http://www.openmp.org*, 3.0, May 2008.

A.2. Propagación de Calor en 2 Dimensiones

heat2d Performance Report

20140913-174730

Abstract

This performance report is intended to support performance analysis and optimization activities. It includes details on program behavior, system configuration and capabilities, including support data from well-known performance analysis tools.

This report was generated using hotspot version 0.1. Homepage http://www.github.com/moreandres/hotspot. Full execution log can be found at ~/.hotspot/heat2d/20140913-174730/hotspot.log.

Contents

1 Program

This section provides details about the program being analyzed.

1. Program: heat2d.
   Program is the name of the program.
2. Timestamp: 20140913-174730.
   Timestamp is a unique identifier used to store information on disk.
3. Parameters Range: [4096, 4608, 5120, 5632, 6144, 6656, 7168, 7680].
   Parameters range is the problem size set used to scale the program.

2 System Capabilities

This section provides details about the system being used for the analysis.

2.1 System Configuration

This subsection provides details about the system configuration.

The hardware in the system is summarized using a hardware lister utility. It reports exact memory configuration, firmware version, mainboard configuration, CPU version and speed, cache configuration, bus speed and others.

The hardware configuration can be used to contrast the system capabilities to well-known benchmarks results on similar systems.

```
memory      7985MiB System memory
processor   Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz
bridge      440FX - 82441FX PMC [Natoma]
bridge      82371SB PIIX3 ISA [Natoma/Triton II]
storage     82371AB/EB/MB PIIX4 IDE
network     82540EM Gigabit Ethernet Controller
bridge      82371AB/EB/MB PIIX4 ACPI
storage     82801HM/HEM (ICH8M/ICH8M-E) SATA Controller [AHCI mode]
```

The software in the system is summarized using the GNU/Linux platform string.

```
Linux-3.13.0-32-generic-x86_64-with-Ubuntu-14.04-trusty
```

The software toolchain is built upon the following components.

1. Host: ubuntu

2. Distribution: `Ubuntu, 14.04, trusty`.
   This codename provides LSB (Linux Standard Base) and distribution-specific information.
3. Compiler: `gcc (Ubuntu 4.8.2-19ubuntu1) 4.8.2`.
   Version number of the compiler program.
4. C Library: `GNU C Library (Ubuntu EGLIBC 2.19-0ubuntu6.3) stable release version 2.19`.
   Version number of the C library.

The software configuration can be used to contrast the system capabilities to well-known benchmark results on similar systems.

2.2 System Performance Baseline

This subsection provides details about the system capabilities.

A set of performance results is included as a reference to contrast systems and to verify hardware capabilities using well-known synthetic benchmarks.

The HPC Challenge benchmark [1] consists of different tests:

1. HPL: the Linpack TPP benchmark which measures the floating point rate of execution for solving a linear system of equations.
2. DGEMM: measures the floating point rate of execution of double precision real matrix-matrix multiplication.
3. PTRANS (parallel matrix transpose): exercises the communications where pairs of processors communicate with each other simultaneously.
4. RandomAccess: measures the rate of integer random updates of memory (GUPS).
5. STREAM: a simple synthetic benchmark program that measures sustainable memory bandwidth (in GB/s).
6. FFT: measures the floating point rate of execution of double precision complex one-dimensional Discrete Fourier Transform (DFT).

Table 1: Benchmarks

Benchmark	Value	Unit
hpl	0.00343933 TFlops	tflops
dgemm	0.939712 GFlops	mflops
ptrans	0.373827 GBs	MB/s
random	0.0275466 GUPs	MB/s
stream	5.34933 MBs	MB/s
fft	1.06575 GFlops	MB/s

Most programs will have a dominant compute kernel that can be approximated by the ones above, the results helps to understand the available capacity.

3 Workload

This section provides details about the workload behavior.

3.1 Workload Footprint

The workload footprint impacts on memory hierarchy usage.

```
heat2d: ELF 64-bit LSB  executable, x86-64, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux
```

Binaries should be stripped to better fit inside cache.

```
struct _IO_FILE {
int                        _flags;              /*     0     4 */

/* XXX 4 bytes hole, try to pack */

char *                     _IO_read_ptr;        /*     8     8 */
char *                     _IO_read_end;        /*    16     8 */
char *                     _IO_read_base;       /*    24     8 */
char *                     _IO_write_base;      /*    32     8 */
char *                     _IO_write_ptr;       /*    40     8 */
char *                     _IO_write_end;       /*    48     8 */
char *                     _IO_buf_base;        /*    56     8 */
/* --- cacheline 1 boundary (64 bytes) --- */
char *                     _IO_buf_end;         /*    64     8 */
char *                     _IO_save_base;       /*    72     8 */
char *                     _IO_backup_base;     /*    80     8 */
```

```
char *                    _IO_save_end;         /*    88     8 */
struct _IO_marker *       _markers;             /*    96     8 */
struct _IO_FILE *         _chain;               /*   104     8 */
int                       _fileno;              /*   112     4 */
int                       _flags2;              /*   116     4 */
__off_t                   _old_offset;          /*   120     8 */
/* --- cacheline 2 boundary (128 bytes) --- */
short unsigned int        _cur_column;          /*   128     2 */
signed char               _vtable_offset;       /*   130     1 */
char                      _shortbuf[1];         /*   131     1 */

/* XXX 4 bytes hole, try to pack */

_IO_lock_t *              _lock;                /*   136     8 */
__off64_t                 _offset;              /*   144     8 */
void *                    __pad1;               /*   152     8 */
void *                    __pad2;               /*   160     8 */
void *                    __pad3;               /*   168     8 */
void *                    __pad4;               /*   176     8 */
size_t                    __pad5;               /*   184     8 */
/* --- cacheline 3 boundary (192 bytes) --- */
int                       _mode;                /*   192     4 */
char                      _unused2[20];         /*   196    20 */

/* size: 216, cachelines: 4, members: 29 */
/* sum members: 208, holes: 2, sum holes: 8 */
/* last cacheline: 24 bytes */
};
struct _IO_marker {
struct _IO_marker *       _next;                /*     0     8 */
struct _IO_FILE *         _sbuf;                /*     8     8 */
int                       _pos;                 /*    16     4 */

/* size: 24, cachelines: 1, members: 3 */
/* padding: 4 */
/* last cacheline: 24 bytes */
};
```

Showing layout of data structures. Reorganizing such data to remove alignment holes. Improve CPU cache utilization.

More information `sevendwarves`https://www.kernel.org/doc/ols/2007/ols2007v2-pages-35-44.pdf

3.2 Workload Stability

This subsection provides details about workload stability.

1. Execution time:
   (a) problem size range: `4096 - 8192`
   (b) geomean: `5.16793` seconds
   (c) average: `5.17504` seconds
   (d) stddev: `0.27067`
   (e) min: `4.80759` seconds
   (f) max: `5.50437` seconds
   (g) repetitions: `8` times

   The histogram plots the elapsed times and shows how they fit in a normal distribution sample.

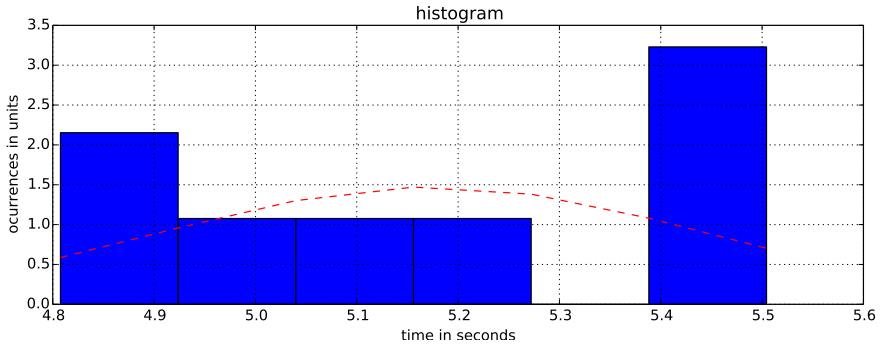

Figure 1: Results Distribution

The workload should run for at least one minute to fully utilize system resources. The execution time of the workload should be stable and the standard deviation less than 3 units.

3.3 Workload Optimization

This section shows how the program reacts to different optimization levels.

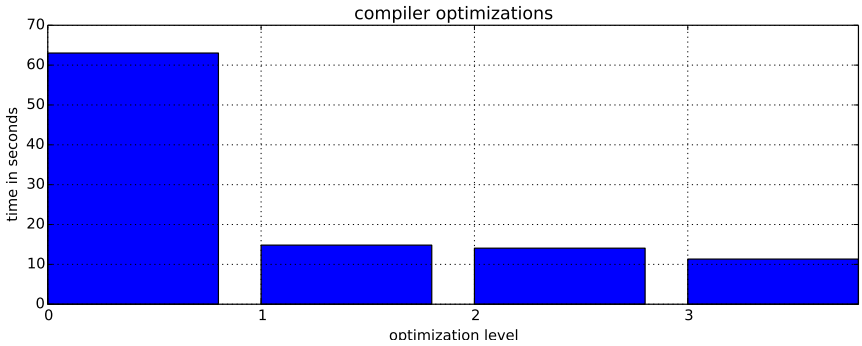

Figure 2: Optimization Levels

4 Scalability

This section provides details about the scaling behavior of the program.

4.1 Problem Size Scalability

A chart with the execution time when scaling the problem size.

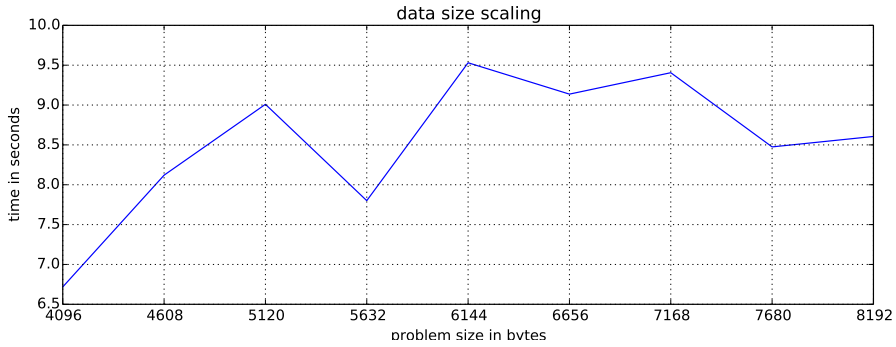

Figure 3: Problem size times

The chart will show how computing time increases when increasing problem size. There should be no valleys or bumps if processing properly balanced across computational units.

4.2 Computing Scalability

A chart with the execution time when scaling computation units.

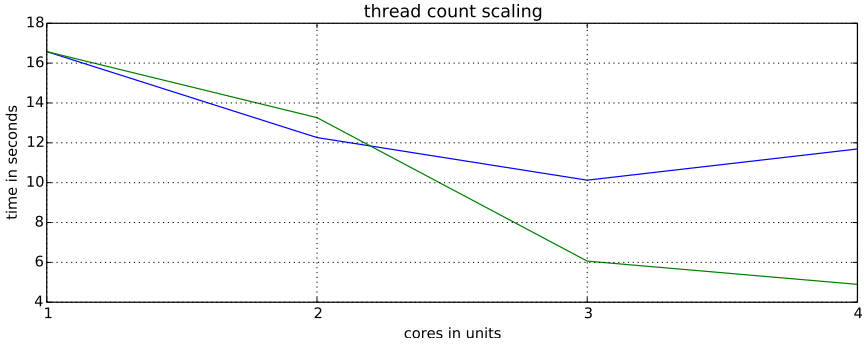

Figure 4: Thread count times

The chart will show how computing time decreases when increasing processing units. An ideal scaling line is provided for comparison.

The parallel and serial fractions of the program can be estimated using the information above.

1. Parallel Fraction: `0.52039`.
   Portion of the program doing parallel work.
2. Serial: `0.47961`.
   Portion of the program doing serial work.

Optimization limits can be estimated using scaling laws.

1. Amdalah Law for 1024 procs: `2.08504 times`.
   Optimizations are limited up to this point when scaling problem size. [**?**]
2. Gustafson Law for 1024 procs: `533.36271 times`.
   Optimizations are limited up to this point when not scaling problem size. [3]

5 Profile

This section provides details about the execution profile of the program and the system.

5.1 Program Profiling

This subsection provides details about the program execution profile.

5.1.1 Flat Profile

The flat profile shows how much time your program spent in each function, and how many times that function was called.

```
Flat profile:
Each sample counts as 0.01 seconds.
     cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 26.27      3.49     3.49                             compute_one_iteration._omp_fn.1 (heat2d.c:78 @ 400d97)
 13.29      5.26     1.77                             compute_one_iteration._omp_fn.1 (heat2d.c:83 @ 400d93)
  9.29      6.50     1.24                             compute_one_iteration._omp_fn.1 (heat2d.c:82 @ 400d8f)
  9.14      7.71     1.21                             compute_one_iteration._omp_fn.1 (heat2d.c:83 @ 400d70)
  9.14      8.93     1.21                             compute_one_iteration._omp_fn.1 (heat2d.c:75 @ 400bd0)
  7.55      9.93     1.00                             compute_one_iteration._omp_fn.1 (heat2d.c:79 @ 400d50)
  7.40     10.91     0.98                             compute_one_iteration._omp_fn.1 (heat2d.c:81 @ 400d89)
  4.68     11.54     0.62                             compute_one_iteration._omp_fn.1 (heat2d.c:80 @ 400d69)
  4.53     12.14     0.60                             compute_one_iteration._omp_fn.1 (heat2d.c:80 @ 400d83)
  3.77     12.64     0.50                             compute_one_iteration._omp_fn.1 (heat2d.c:83 @ 400d63)
  2.57     12.98     0.34                             compute_one_iteration._omp_fn.1 (heat2d.c:83 @ 400d7b)
  1.81     13.22     0.24                             compute_one_iteration._omp_fn.1 (heat2d.c:79 @ 400d5c)
Call graph
granularity: each sample hit covers 2 byte(s) for 0.07 of 13.35 seconds
index  time    self  children    called     name
                                                 <spontaneous>
[15]     0.1    0.01    0.00                 compute_one_iteration (heat2d.c:54 @ 400ee0) [15]
[20]     0.0    0.00    0.00   74410         frame_dummy [20]
```

The table shows where to focus optimization efforts to maximize impact.

5.2 System Profiling

This subsection provide details about the system execution profile.

5.2.1 System Resources Usage

The following charts shows the state of system resources during the execution of the program.

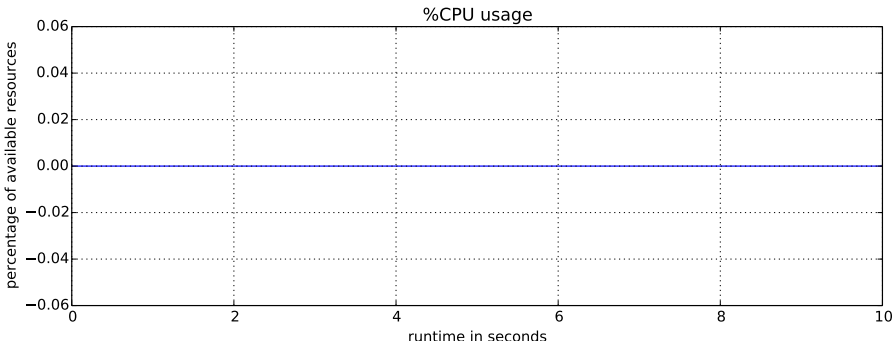

Figure 5: CPU Usage

Note that this chart is likely to show as upper limit a multiple of 100% in case a multicore system is being used.

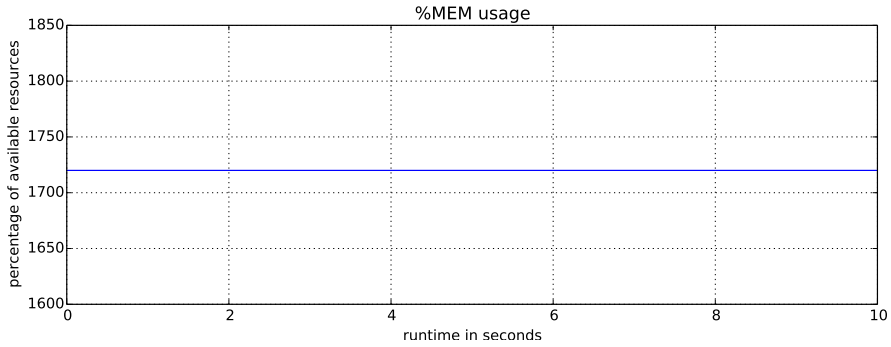

Figure 6: Memory Usage

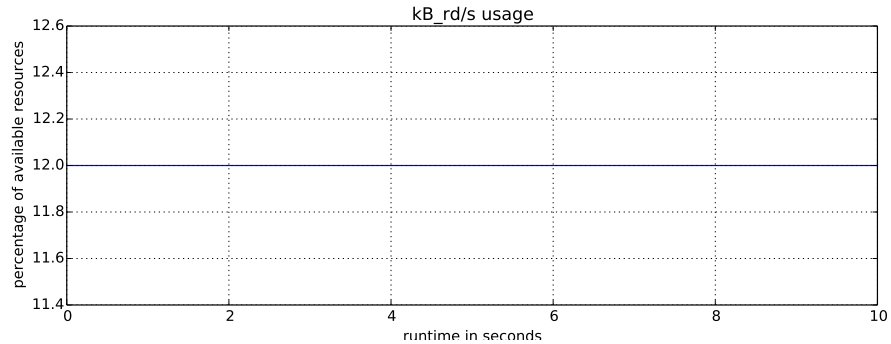

Figure 7: Reads from Disk

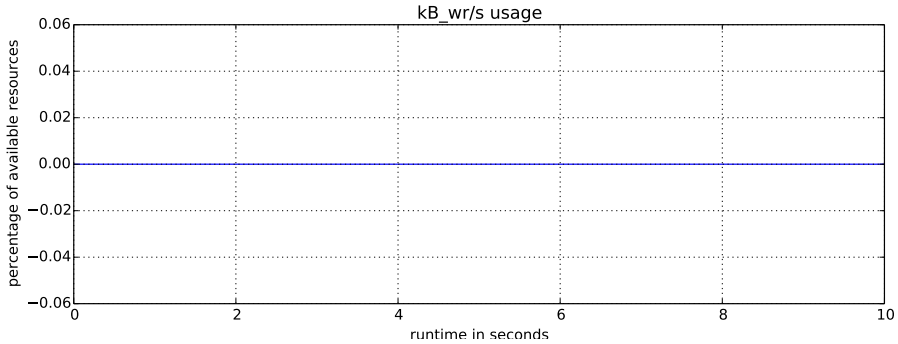

Figure 8: Writes to Disk

5.3 Hotspots

This subsection shows annotated code guiding the optimization efforts.

```
[kernel.kallsyms] with build id fa854edfe97c596cddeac7049b080cbba9de2775 not found, continuing without symbols
         :     {
         :       if (i < 256 || i > 768)
 Percent | Source code & Disassembly of libc-2.19.so
------------------------------------------------------------
         : Disassembly of section .text:
   36.36 :   83877:       mov    (rax),rax
```

```
   18.18 :   8387a:       test   rax,rax
--------------------------------------------------------------
         : Disassembly of section .text:
   33.33 :   74e3:       retq
 Percent | Source code & Disassembly of libc-2.19.so
----------------------------------------------------------
         : Disassembly of section .text:
--------------------------------------------------------------
         : Disassembly of section .text:
```

6 Low Level

This section provide details about low level details such as vectorization and performance counters.

6.1 Vectorization Report

This subsection provide details about vectorization status of the program loops.

```
Analyzing loop at heat2d.c:46
heat2d.c:46: note: not vectorized: loop contains function calls or data references that cannot be analyzed
heat2d.c:46: note: bad data references.
heat2d.c:43: note: vectorized 0 loops in function.
heat2d.c:43: note: not vectorized: not enough data-refs in basic block.
heat2d.c:43: note: not vectorized: not enough data-refs in basic block.
heat2d.c:43: note: not vectorized: not enough data-refs in basic block.
heat2d.c:46: note: not vectorized: not enough data-refs in basic block.
heat2d.c:43: note: not vectorized: not enough data-refs in basic block.
heat2d.c:43: note: not vectorized: not enough data-refs in basic block.
heat2d.c:43: note: not vectorized: not enough data-refs in basic block.
heat2d.c:75: note: versioning for alias required: can't determine dependence between solution[pretmp_108][pretmp_1
heat2d.c:75: note: versioning not yet supported for outer-loops.
heat2d.c:75: note: bad data dependence.
Analyzing loop at heat2d.c:77
heat2d.c:77: note: versioning for alias required: can't determine dependence between solution[pretmp_108][pretmp_1
heat2d.c:77: note: versioning for alias required: can't determine dependence between solution[pretmp_108][pretmp_1
heat2d.c:77: note: versioning for alias required: can't determine dependence between solution[pretmp_108][i_3][j_2
heat2d.c:77: note: versioning for alias required: can't determine dependence between solution[pretmp_108][i_3][_29
heat2d.c:77: note: versioning for alias required: can't determine dependence between solution[pretmp_108][i_3][j_3
heat2d.c:77: note: misalign = 8 bytes of ref solution[pretmp_108][pretmp_112][j_39]
heat2d.c:77: note: misalign = 8 bytes of ref solution[pretmp_108][pretmp_113][j_39]
heat2d.c:77: note: misalign = 0 bytes of ref solution[pretmp_108][i_3][j_26]
heat2d.c:77: note: misalign = 0 bytes of ref solution[pretmp_108][i_3][_29]
heat2d.c:77: note: misalign = 8 bytes of ref solution[pretmp_108][i_3][j_39]
heat2d.c:77: note: misalign = 8 bytes of ref solution[pretmp_106][i_3][j_39]
heat2d.c:77: note: num. args = 4 (not unary/binary/ternary op).
heat2d.c:77: note: not ssa-name.
heat2d.c:77: note: use not simple.
heat2d.c:77: note: num. args = 4 (not unary/binary/ternary op).
heat2d.c:77: note: not ssa-name.
heat2d.c:77: note: use not simple.
heat2d.c:77: note: num. args = 4 (not unary/binary/ternary op).
heat2d.c:77: note: not ssa-name.
heat2d.c:77: note: use not simple.
heat2d.c:77: note: num. args = 4 (not unary/binary/ternary op).
heat2d.c:77: note: not ssa-name.
heat2d.c:77: note: use not simple.
heat2d.c:77: note: num. args = 4 (not unary/binary/ternary op).
heat2d.c:77: note: not ssa-name.
heat2d.c:77: note: use not simple.
Vectorizing loop at heat2d.c:77
heat2d.c:75: note: vectorized 1 loops in function.
heat2d.c:75: note: not vectorized: not enough data-refs in basic block.
heat2d.c:75: note: not vectorized: not enough data-refs in basic block.
heat2d.c:75: note: not consecutive access pretmp_106 = next_gen;
heat2d.c:75: note: not consecutive access pretmp_108 = cur_gen;
heat2d.c:75: note: not consecutive access pretmp_110 = diff_constant;
heat2d.c:75: note: Failed to SLP the basic block.
heat2d.c:75: note: not vectorized: failed to find SLP opportunities in basic block.
heat2d.c:75: note: not vectorized: not enough data-refs in basic block.
```

```
heat2d.c:75: note: not vectorized: not enough data-refs in basic block.
heat2d.c:75: note: not vectorized: not enough data-refs in basic block.
heat2d.c:75: note: not vectorized: not enough data-refs in basic block.
heat2d.c:75: note: not vectorized: not enough data-refs in basic block.
heat2d.c:79: note: can't determine dependence between solution[pretmp_108][pretmp_112][j_140] and solution[pretmp_
heat2d.c:79: note: can't determine dependence between solution[pretmp_108][pretmp_113][j_140] and solution[pretmp_
heat2d.c:79: note: can't determine dependence between solution[pretmp_108][i_3][j_146] and solution[pretmp_106][i_
heat2d.c:79: note: can't determine dependence between solution[pretmp_108][i_3][_149] and solution[pretmp_106][i_3
heat2d.c:79: note: can't determine dependence between solution[pretmp_108][i_3][j_140] and solution[pretmp_106][i_
heat2d.c:79: note: SLP: step doesn't divide the vector-size.
heat2d.c:79: note: Unknown alignment for access: solution
heat2d.c:79: note: SLP: step doesn't divide the vector-size.
heat2d.c:79: note: Unknown alignment for access: solution
heat2d.c:79: note: SLP: step doesn't divide the vector-size.
heat2d.c:79: note: Unknown alignment for access: solution
heat2d.c:79: note: SLP: step doesn't divide the vector-size.
heat2d.c:79: note: Unknown alignment for access: solution
heat2d.c:79: note: SLP: step doesn't divide the vector-size.
heat2d.c:79: note: Unknown alignment for access: solution
heat2d.c:79: note: SLP: step doesn't divide the vector-size.
heat2d.c:79: note: Unknown alignment for access: solution
heat2d.c:79: note: Failed to SLP the basic block.
heat2d.c:79: note: not vectorized: failed to find SLP opportunities in basic block.
heat2d.c:75: note: not vectorized: not enough data-refs in basic block.
heat2d.c:75: note: not vectorized: not enough data-refs in basic block.
heat2d.c:79: note: not vectorized: no vectype for stmt: vect_var_.72_169 = MEM[(double[2][302][302] *)vect_psoluti
 scalar_type: vector(2) double
heat2d.c:79: note: Failed to SLP the basic block.
heat2d.c:79: note: not vectorized: failed to find SLP opportunities in basic block.
heat2d.c:75: note: not vectorized: not enough data-refs in basic block.
heat2d.c:75: note: not vectorized: not enough data-refs in basic block.
heat2d.c:43: note: not vectorized: not enough data-refs in basic block.
Analyzing loop at heat2d.c:64
heat2d.c:64: note: misalign = 0 bytes of ref solution[pretmp_34][i_41][300]
heat2d.c:64: note: misalign = 8 bytes of ref solution[pretmp_34][i_41][301]
heat2d.c:64: note: not consecutive access _19 = solution[pretmp_34][i_41][300];
heat2d.c:64: note: not vectorized: complicated access pattern.
heat2d.c:64: note: bad data access.
Analyzing loop at heat2d.c:57
heat2d.c:57: note: not vectorized: control flow in loop.
heat2d.c:57: note: bad loop form.
heat2d.c:53: note: vectorized 0 loops in function.
heat2d.c:53: note: not consecutive access pretmp_34 = cur_gen;
heat2d.c:53: note: Failed to SLP the basic block.
heat2d.c:53: note: not vectorized: failed to find SLP opportunities in basic block.
heat2d.c:59: note: not vectorized: not enough data-refs in basic block.
heat2d.c:60: note: misalign = 8 bytes of ref solution[pretmp_34][i_40][1]
heat2d.c:60: note: misalign = 0 bytes of ref solution[pretmp_34][i_40][0]
heat2d.c:60: note: not consecutive access _12 = solution[pretmp_34][i_40][1];
heat2d.c:60: note: not consecutive access solution[pretmp_34][i_40][0] = _12;
heat2d.c:60: note: Failed to SLP the basic block.
heat2d.c:60: note: not vectorized: failed to find SLP opportunities in basic block.
heat2d.c:62: note: misalign = 0 bytes of ref solution[pretmp_34][i_40][0]
heat2d.c:62: note: not consecutive access solution[pretmp_34][i_40][0] = 2.0e+1;
heat2d.c:62: note: Failed to SLP the basic block.
heat2d.c:62: note: not vectorized: failed to find SLP opportunities in basic block.
heat2d.c:57: note: not vectorized: not enough data-refs in basic block.
heat2d.c:53: note: not vectorized: not enough data-refs in basic block.
heat2d.c:53: note: not vectorized: not enough data-refs in basic block.
heat2d.c:66: note: misalign = 0 bytes of ref solution[pretmp_34][i_41][300]
heat2d.c:66: note: misalign = 8 bytes of ref solution[pretmp_34][i_41][301]
heat2d.c:66: note: not consecutive access _19 = solution[pretmp_34][i_41][300];
heat2d.c:66: note: not consecutive access solution[pretmp_34][i_41][301] = _19;
heat2d.c:66: note: Failed to SLP the basic block.
heat2d.c:66: note: not vectorized: failed to find SLP opportunities in basic block.
heat2d.c:53: note: not vectorized: not enough data-refs in basic block.
heat2d.c:70: note: not vectorized: not enough data-refs in basic block.
Analyzing loop at heat2d.c:28
heat2d.c:28: note: not vectorized: number of iterations cannot be computed.
heat2d.c:28: note: bad loop form.
```

```
heat2d.c:18: note: vectorized 0 loops in function.
heat2d.c:20: note: misalign = 0 bytes of ref final
heat2d.c:20: note: not consecutive access final = 1.024e+3;
heat2d.c:20: note: Failed to SLP the basic block.
heat2d.c:20: note: not vectorized: failed to find SLP opportunities in basic block.
heat2d.c:22: note: not vectorized: not enough data-refs in basic block.
heat2d.c:26: note: not vectorized: not enough data-refs in basic block.
heat2d.c:36: note: not vectorized: not enough data-refs in basic block.
heat2d.c:18: note: not vectorized: not enough data-refs in basic block.
heat2d.c:30: note: not vectorized: not enough data-refs in basic block.
heat2d.c:18: note: not vectorized: not enough data-refs in basic block.
```

The details above shows the list of loops in the program and if they are being vectorized or not. These reports can pinpoint areas where the compiler cannot apply vectorization and related optimizations. It may be possible to modify your code or communicate additional information to the compiler to guide the vectorization and/or optimizations.

6.2 Counters Report

This subsection provides details about software and hardware counters.

```
 Performance counter stats for './heat2d' (3 runs):

      43611.260053 task-clock (msec)         #    3.700 CPUs utilized            ( +-  3.62 )
             2,112 context-switches          #    0.048 K/sec                    ( +- 12.35 )
                12 cpu-migrations            #    0.000 K/sec                    ( +- 18.74 )
               538 page-faults               #    0.012 K/sec                    ( +-  0.06 )
   <not supported> cycles
   <not supported> stalled-cycles-frontend
   <not supported> stalled-cycles-backend
   <not supported> instructions
   <not supported> branches
   <not supported> branch-misses

      11.785455878 seconds time elapsed                                          ( +-  4.91 )
```

The details above shows counters that provide low-overhead access to detailed performance information using internal registers of the CPU.

References

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

[2] Amdahl, Gene M., *Validity of the single processor approach to achieving large scale computing capabilities*. Communications of the ACM, Proceedings of the April 18-20, 1967, spring joint computer conference Pages 483-485, 1967.

[3] John L. Gustafson, *Reevaluating Amdahl's Law*. Communications of the ACM, Volume 31 Pages 532-533, 1988.

[4] OpenMP Architecture Review Board, *OpenMP Application Program Interface*. *http://www.openmp.org*, 3.0, May 2008.

A.3. Mandel

`mandel` Performance Report

20140913-174745

Abstract

This performance report is intended to support performance analysis and optimization activities. It includes details on program behavior, system configuration and capabilities, including support data from well-known performance analysis tools.

This report was generated using `hotspot` version `0.1`. Homepage `http://www.github.com/moreandres/hotspot`. Full execution log can be found at `~/.hotspot/mandel/20140913-174745/hotspot.log`.

Contents

1 Program

This section provides details about the program being analyzed.

1. Program: `mandel`.
   Program is the name of the program.
2. Timestamp: `20140913-174745`.
   Timestamp is a unique identifier used to store information on disk.
3. Parameters Range: `[8192, 9216, 10240, 11264, 12288, 13312, 14336, 15360]`.
   Parameters range is the problem size set used to scale the program.

2 System Capabilities

This section provides details about the system being used for the analysis.

2.1 System Configuration

This subsection provides details about the system configuration.

The hardware in the system is summarized using a hardware lister utility. It reports exact memory configuration, firmware version, mainboard configuration, CPU version and speed, cache configuration, bus speed and others.

The hardware configuration can be used to contrast the system capabilities to well-known benchmarks results on similar systems.

```
memory      7985MiB System memory
processor   Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz
bridge      440FX - 82441FX PMC [Natoma]
bridge      82371SB PIIX3 ISA [Natoma/Triton II]
storage     82371AB/EB/MB PIIX4 IDE
network     82540EM Gigabit Ethernet Controller
bridge      82371AB/EB/MB PIIX4 ACPI
storage     82801HM/HEM (ICH8M/ICH8M-E) SATA Controller [AHCI mode]
```

The software in the system is summarized using the GNU/Linux platform string.

```
Linux-3.13.0-32-generic-x86_64-with-Ubuntu-14.04-trusty
```

The software toolchain is built upon the following components.

1. Host: `ubuntu`

2. Distribution: `Ubuntu, 14.04, trusty`.
   This codename provides LSB (Linux Standard Base) and distribution-specific information.
3. Compiler: `gcc (Ubuntu 4.8.2-19ubuntu1) 4.8.2`.
   Version number of the compiler program.
4. C Library: `GNU C Library (Ubuntu EGLIBC 2.19-0ubuntu6.3) stable release version 2.19`.
   Version number of the C library.

The software configuration can be used to contrast the system capabilities to well-known benchmark results on similar systems.

2.2 System Performance Baseline

This subsection provides details about the system capabilities.

A set of performance results is included as a reference to contrast systems and to verify hardware capabilities using well-known synthetic benchmarks.

The HPC Challenge benchmark [1] consists of different tests:

1. HPL: the Linpack TPP benchmark which measures the floating point rate of execution for solving a linear system of equations.
2. DGEMM: measures the floating point rate of execution of double precision real matrix-matrix multiplication.
3. PTRANS (parallel matrix transpose): exercises the communications where pairs of processors communicate with each other simultaneously.
4. RandomAccess: measures the rate of integer random updates of memory (GUPS).
5. STREAM: a simple synthetic benchmark program that measures sustainable memory bandwidth (in GB/s).
6. FFT: measures the floating point rate of execution of double precision complex one-dimensional Discrete Fourier Transform (DFT).

Table 1: Benchmarks

Benchmark	Value	Unit
hpl	0.00370076 TFlops	tflops
dgemm	0.945634 GFlops	mflops
ptrans	0.969288 GBs	MB/s
random	0.025916 GUPs	MB/s
stream	4.71698 MBs	MB/s
fft	1.13118 GFlops	MB/s

Most programs will have a dominant compute kernel that can be approximated by the ones above, the results helps to understand the available capacity.

3 Workload

This section provides details about the workload behavior.

3.1 Workload Footprint

The workload footprint impacts on memory hierarchy usage.

```
mandel: ELF 64-bit LSB  executable, x86-64, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux
```

Binaries should be stripped to better fit inside cache.

```
struct _IO_FILE {
int                        _flags;              /*     0     4 */

/* XXX 4 bytes hole, try to pack */

char *                     _IO_read_ptr;        /*     8     8 */
char *                     _IO_read_end;        /*    16     8 */
char *                     _IO_read_base;       /*    24     8 */
char *                     _IO_write_base;      /*    32     8 */
char *                     _IO_write_ptr;       /*    40     8 */
char *                     _IO_write_end;       /*    48     8 */
char *                     _IO_buf_base;        /*    56     8 */
/* --- cacheline 1 boundary (64 bytes) --- */
char *                     _IO_buf_end;         /*    64     8 */
char *                     _IO_save_base;       /*    72     8 */
char *                     _IO_backup_base;     /*    80     8 */
```

```
char *                    _IO_save_end;         /*    88     8 */
struct _IO_marker *       _markers;             /*    96     8 */
struct _IO_FILE *         _chain;               /*   104     8 */
int                       _fileno;              /*   112     4 */
int                       _flags2;              /*   116     4 */
__off_t                   _old_offset;          /*   120     8 */
/* --- cacheline 2 boundary (128 bytes) --- */
short unsigned int        _cur_column;          /*   128     2 */
signed char               _vtable_offset;       /*   130     1 */
char                      _shortbuf[1];         /*   131     1 */

/* XXX 4 bytes hole, try to pack */

_IO_lock_t *              _lock;                /*   136     8 */
__off64_t                 _offset;              /*   144     8 */
void *                    __pad1;               /*   152     8 */
void *                    __pad2;               /*   160     8 */
void *                    __pad3;               /*   168     8 */
void *                    __pad4;               /*   176     8 */
size_t                    __pad5;               /*   184     8 */
/* --- cacheline 3 boundary (192 bytes) --- */
int                       _mode;                /*   192     4 */
char                      _unused2[20];         /*   196    20 */

/* size: 216, cachelines: 4, members: 29 */
/* sum members: 208, holes: 2, sum holes: 8 */
/* last cacheline: 24 bytes */
};
struct _IO_marker {
struct _IO_marker *       _next;                /*     0     8 */
struct _IO_FILE *         _sbuf;                /*     8     8 */
int                       _pos;                 /*    16     4 */

/* size: 24, cachelines: 1, members: 3 */
/* padding: 4 */
/* last cacheline: 24 bytes */
};
struct complextype {
float                     real;                 /*     0     4 */
float                     imag;                 /*     4     4 */

/* size: 8, cachelines: 1, members: 2 */
/* last cacheline: 8 bytes */
};
```

Showing layout of data structures. Reorganizing such data to remove alignment holes. Improve CPU cache utilization.

More information `sevendwarves`https://www.kernel.org/doc/ols/2007/ols2007v2-pages-35-44.pdf

3.2 Workload Stability

This subsection provides details about workload stability.

1. Execution time:
   (a) problem size range: `8192 - 16384`
   (b) geomean: `2.76947` seconds
   (c) average: `2.77083` seconds
   (d) stddev: `0.08857`
   (e) min: `2.71044` seconds
   (f) max: `2.99845` seconds
   (g) repetitions: `8` times

   The histogram plots the elapsed times and shows how they fit in a normal distribution sample.

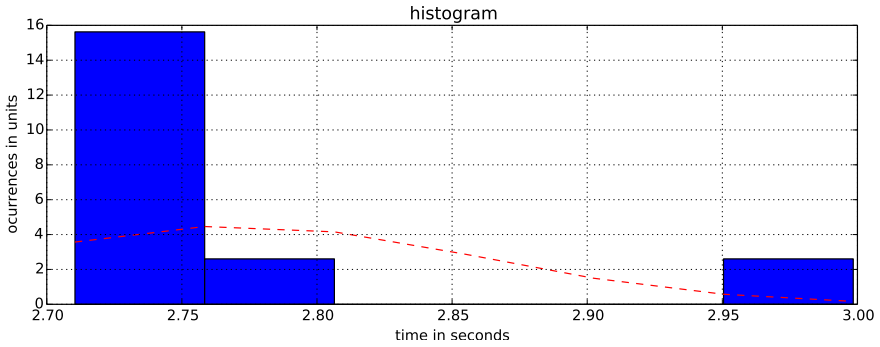

Figure 1: Results Distribution

The workload should run for at least one minute to fully utilize system resources. The execution time of the workload should be stable and the standard deviation less than 3 units.

3.3 Workload Optimization

This section shows how the program reacts to different optimization levels.

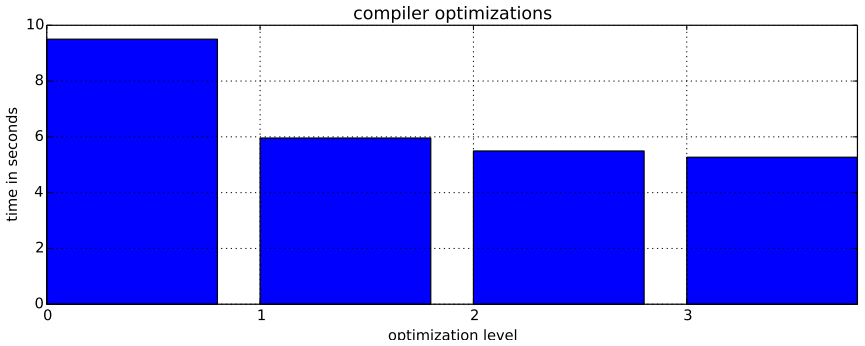

Figure 2: Optimization Levels

4 Scalability

This section provides details about the scaling behavior of the program.

4.1 Problem Size Scalability

A chart with the execution time when scaling the problem size.

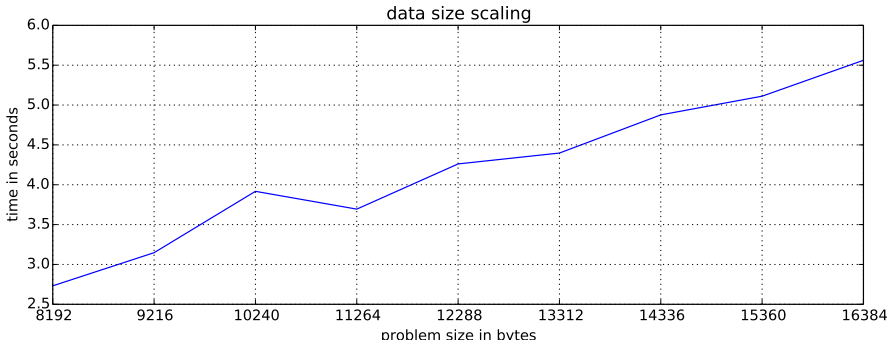

Figure 3: Problem size times

The chart will show how computing time increases when increasing problem size. There should be no valleys or bumps if processing properly balanced across computational units.

4.2 Computing Scalability

A chart with the execution time when scaling computation units.

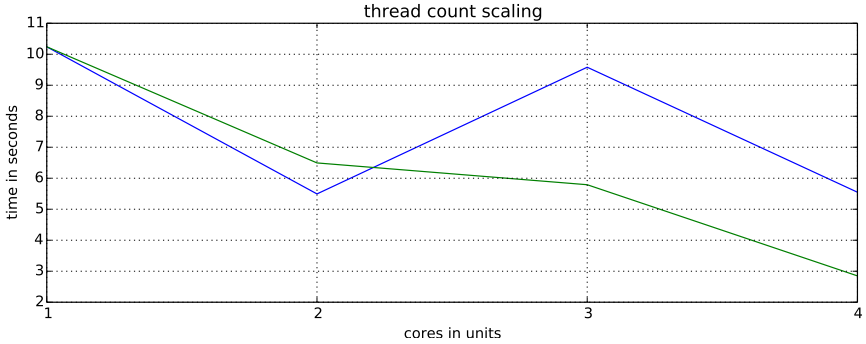

Figure 4: Thread count times

The chart will show how computing time decreases when increasing processing units. An ideal scaling line is provided for comparison.

The parallel and serial fractions of the program can be estimated using the information above.

1. Parallel Fraction: `0.92707`.
   Portion of the program doing parallel work.
2. Serial: `0.07293`.
   Portion of the program doing serial work.

Optimization limits can be estimated using scaling laws.

1. Amdalah Law for 1024 procs: `13.71143 times`.
   Optimizations are limited up to this point when scaling problem size. [**?**]
2. Gustafson Law for 1024 procs: `949.39071 times`.
   Optimizations are limited up to this point when not scaling problem size. [3]

5 Profile

This section provides details about the execution profile of the program and the system.

5.1 Program Profiling

This subsection provides details about the program execution profile.

5.1.1 Flat Profile

The flat profile shows how much time your program spent in each function, and how many times that function was called.

```
Flat profile:
Each sample counts as 0.01 seconds.
     cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 46.82      4.65     4.65                             main._omp_fn.0 (mandel.c:45 @ 400afd)
 14.03      6.04     1.39                             main._omp_fn.0 (mandel.c:48 @ 400b0f)
 11.03      7.14     1.10                             main._omp_fn.0 (mandel.c:43 @ 400aed)
  7.07      7.84     0.70                             main._omp_fn.0 (mandel.c:43 @ 400ad8)
  3.76      8.21     0.37                             main._omp_fn.0 (mandel.c:42 @ 400ae1)
  3.61      8.57     0.36                             main._omp_fn.0 (mandel.c:48 @ 400ac0)
  3.46      8.91     0.34                             main._omp_fn.0 (mandel.c:43 @ 400ad1)
  3.41      9.25     0.34                             main._omp_fn.0 (mandel.c:46 @ 400ace)
  3.35      9.59     0.33                             main._omp_fn.0 (mandel.c:43 @ 400ae5)
  2.69      9.85     0.27                             main._omp_fn.0 (mandel.c:36 @ 400aa4)
```

The table shows where to focus optimization efforts to maximize impact.

5.2 System Profiling

This subsection provide details about the system execution profile.

5.2.1 System Resources Usage

The following charts shows the state of system resources during the execution of the program.

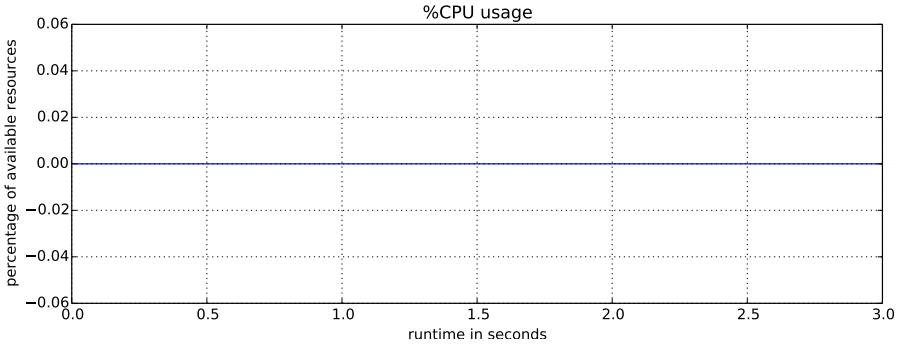

Figure 5: CPU Usage

Note that this chart is likely to show as upper limit a multiple of 100% in case a multicore system is being used.

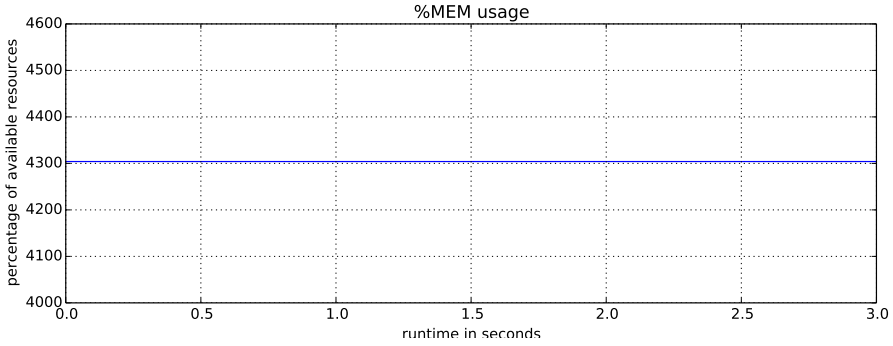

Figure 6: Memory Usage

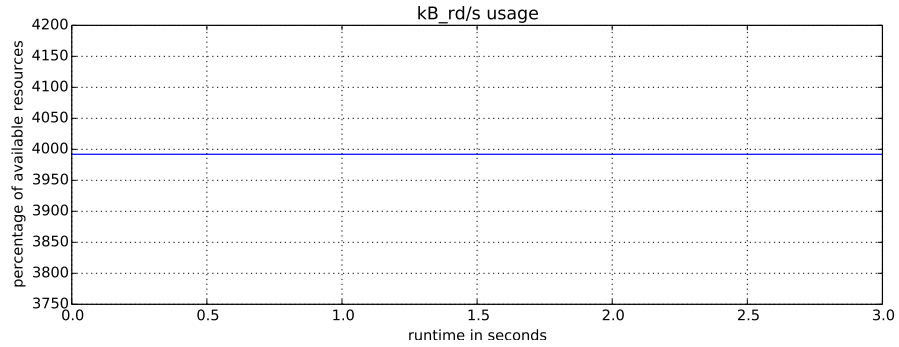

Figure 7: Reads from Disk

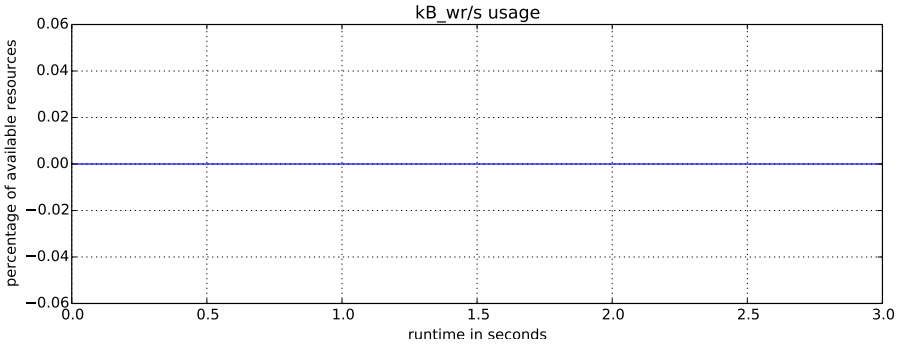

Figure 8: Writes to Disk

5.3 Hotspots

This subsection shows annotated code guiding the optimization efforts.

```
[kernel.kallsyms] with build id fa854edfe97c596cddeac7049b080cbba9de2775 not found, continuing without symbols
         :             z.real = temp;
         :             lensq = z.real * z.real + z.imag * z.imag;
         :             k++;
         :           }
         :         while (lensq < 4.0 && k < iters);
    4.60 :   400968:       cmp    edx,eax
         :         do
         :           {
         :             temp = z.real * z.real - z.imag * z.imag + c.real;
         :             z.imag = 2.0 * z.real * z.imag + c.imag;
         :             z.real = temp;
    7.41 :   400970:       movaps xmm1,xmm0
         :         c.imag = Y_MAX - i * (Y_MAX - Y_MIN) / Y_RESN;
         :         k = 0;
         :         do
         :           {
         :             temp = z.real * z.real - z.imag * z.imag + c.real;
         :             z.imag = 2.0 * z.real * z.imag + c.imag;
         :             z.real = temp;
         :             lensq = z.real * z.real + z.imag * z.imag;
         :             k++;
    0.36 :   400976:       add    $0x1,eax
         :         k = 0;
         :         do
         :           {
         :             temp = z.real * z.real - z.imag * z.imag + c.real;
```

```
      :             z.imag = 2.0 * z.real * z.imag + c.imag;
 6.06 :   400979:       unpcklps xmm2,xmm2
      :         c.imag = Y_MAX - i * (Y_MAX - Y_MIN) / Y_RESN;
      :         k = 0;
      :         do
      :           {
      :             temp = z.real * z.real - z.imag * z.imag + c.real;
      :             z.imag = 2.0 * z.real * z.imag + c.imag;
 0.38 :   400983:       cvtps2pd xmm2,xmm2
 6.35 :   400986:       cvtps2pd xmm0,xmm0
      :         c.imag = Y_MAX - i * (Y_MAX - Y_MIN) / Y_RESN;
      :         k = 0;
      :         do
      :           {
      :             temp = z.real * z.real - z.imag * z.imag + c.real;
 0.54 :   400989:       subss  xmm3,xmm1
      :             z.imag = 2.0 * z.real * z.imag + c.imag;
 5.82 :   40098d:       addsd  xmm0,xmm0
      :         c.imag = Y_MAX - i * (Y_MAX - Y_MIN) / Y_RESN;
      :         k = 0;
      :         do
      :           {
      :             temp = z.real * z.real - z.imag * z.imag + c.real;
      :             z.imag = 2.0 * z.real * z.imag + c.imag;
 0.52 :   400999:       addsd  xmm6,xmm0
 6.48 :   40099d:       unpcklpd xmm0,xmm0
      :             z.real = temp;
      :             lensq = z.real * z.real + z.imag * z.imag;
 5.28 :   4009a5:       movaps xmm1,xmm0
 2.18 :   4009a8:       mulss  xmm1,xmm0
 3.72 :   4009ac:       movaps xmm2,xmm3
 0.29 :   4009af:       mulss  xmm2,xmm3
20.22 :   4009b3:       addss  xmm3,xmm0
      :             k++;
      :           }
      :         while (lensq < 4.0 && k < iters);
17.15 :   4009b7:       ucomiss xmm0,xmm5
      :         if (k >= iters)
      :           res[i][j] = 0;
      :         else
      :           res[i][j] = 1;
      :   if (getenv("N"))
```

6 Low Level

This section provide details about low level details such as vectorization and performance counters.

6.1 Vectorization Report

This subsection provide details about vectorization status of the program loops.

```
mandel.c:31: note: not vectorized: multiple nested loops.
mandel.c:31: note: bad loop form.
Analyzing loop at mandel.c:33
mandel.c:33: note: not vectorized: control flow in loop.
mandel.c:33: note: bad inner-loop form.
mandel.c:33: note: not vectorized: Bad inner loop.
mandel.c:33: note: bad loop form.
Analyzing loop at mandel.c:42
mandel.c:42: note: not vectorized: control flow in loop.
mandel.c:42: note: bad loop form.
mandel.c:31: note: vectorized 0 loops in function.
mandel.c:31: note: not vectorized: not enough data-refs in basic block.
mandel.c:31: note: not vectorized: not enough data-refs in basic block.
mandel.c:31: note: not consecutive access pretmp_142 = .omp_data_i_50(D)->res;
mandel.c:31: note: Failed to SLP the basic block.
mandel.c:31: note: not vectorized: failed to find SLP opportunities in basic block.
mandel.c:31: note: not vectorized: not enough data-refs in basic block.
mandel.c:31: note: not vectorized: not enough data-refs in basic block.
```

```
mandel.c:31: note: not vectorized: not enough data-refs in basic block.
mandel.c:31: note: not vectorized: not enough data-refs in basic block.
mandel.c:36: note: not consecutive access pretmp_129 = .omp_data_i_50(D)->iters;
mandel.c:36: note: Failed to SLP the basic block.
mandel.c:36: note: not vectorized: failed to find SLP opportunities in basic block.
mandel.c:42: note: not vectorized: not enough data-refs in basic block.
mandel.c:31: note: not vectorized: not enough data-refs in basic block.
mandel.c:50: note: not vectorized: not enough data-refs in basic block.
mandel.c:33: note: not vectorized: not enough data-refs in basic block.
mandel.c:53: note: SLP: step doesn't divide the vector-size.
mandel.c:53: note: Unknown alignment for access: *pretmp_142
mandel.c:53: note: Failed to SLP the basic block.
mandel.c:53: note: not vectorized: failed to find SLP opportunities in basic block.
mandel.c:51: note: SLP: step doesn't divide the vector-size.
mandel.c:51: note: Unknown alignment for access: *pretmp_142
mandel.c:51: note: Failed to SLP the basic block.
mandel.c:51: note: not vectorized: failed to find SLP opportunities in basic block.
mandel.c:48: note: not vectorized: not enough data-refs in basic block.
mandel.c:31: note: not vectorized: not enough data-refs in basic block.
mandel.c:31: note: not vectorized: not enough data-refs in basic block.
mandel.c:31: note: not vectorized: not enough data-refs in basic block.
mandel.c:31: note: not vectorized: not enough data-refs in basic block.
mandel.c:28: note: not vectorized: not enough data-refs in basic block.
mandel.c:29: note: not vectorized: not enough data-refs in basic block.
mandel.c:31: note: misalign = 0 bytes of ref .omp_data_o.1.res
mandel.c:31: note: misalign = 8 bytes of ref .omp_data_o.1.iters
mandel.c:31: note: not consecutive access .omp_data_o.1.res = &res;
mandel.c:31: note: not consecutive access .omp_data_o.1.iters = iters_1;
mandel.c:31: note: Failed to SLP the basic block.
mandel.c:31: note: not vectorized: failed to find SLP opportunities in basic block.
mandel.c:57: note: not vectorized: not enough data-refs in basic block.
mandel.c:19: note: not vectorized: no vectype for stmt: res ={v} {CLOBBER};
 scalar_type: int[1024][1024]
mandel.c:19: note: Failed to SLP the basic block.
mandel.c:19: note: not vectorized: failed to find SLP opportunities in basic block.
```

The details above shows the list of loops in the program and if they are being vectorized or not. These reports can pinpoint areas where the compiler cannot apply vectorization and related optimizations. It may be possible to modify your code or communicate additional information to the compiler to guide the vectorization and/or optimizations.

6.2 Counters Report

This subsection provides details about software and hardware counters.

```
 Performance counter stats for './mandel' (3 runs):

      10295.944705 task-clock (msec)         #    1.991 CPUs utilized            ( +-  3.00 )
                68 context-switches          #    0.007 K/sec                    ( +- 10.08 )
                 7 cpu-migrations            #    0.001 K/sec
               690 page-faults               #    0.067 K/sec                    ( +-  0.05 )
   <not supported> cycles
   <not supported> stalled-cycles-frontend
   <not supported> stalled-cycles-backend
   <not supported> instructions
   <not supported> branches
   <not supported> branch-misses

       5.170300043 seconds time elapsed                                          ( +-  2.88 )
```

The details above shows counters that provide low-overhead access to detailed performance information using internal registers of the CPU.

References

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

[2] Amdahl, Gene M., *Validity of the single processor approach to achieving large scale computing capabilities.* Communications of the ACM, Proceedings of the April 18-20, 1967, spring joint computer conference Pages 483-485, 1967.

[3] John L. Gustafson, *Reevaluating Amdahl's Law.* Communications of the ACM, Volume 31 Pages 532-533, 1988.

[4] OpenMP Architecture Review Board, *OpenMP Application Program Interface. http://www.openmp.org*, 3.0, May 2008.