Asociación Argentina
de Mecánica Computacional

Mecánica Computacional Vol XXIX, págs. 3077-3091 (artículo completo)
Eduardo Dvorkin, Marcela Goldschmit, Mario Storti (Eds.)
Buenos Aires, Argentina, 15-18 Noviembre 2010

ANÁLISIS DE LA PERFORMANCE MEDIANTE EL USO DE "VALGRIND" DE UN CÓDIGO COMPUTACIONAL PARA LA SIMULACIÓN DEL COMPORTAMIENTO AERODINÁMICO DE VEHÍCULOS AÉREOS NO TRIPULADOS

Alejandro Llanos[a], **Luis Ceballos**[b,c] **y Sergio Preidikman**[b,c,d]

[a] *Magíster en Ciencias de la Ingeniería mención Modelación Matemática. Universidad de La Frontera, Av. Francisco Salazar 01145. Temuco, Chile, a.llanos01@ufromail.cl*

[b] *Facultad de Ingeniería, Universidad Nacional de Río Cuarto, Ruta Nacional 36 km 601, 5800 Río Cuarto Córdoba, Argentina, lceballos@ing.unrc.edu.ar, http://www.ing.unrc.edu.ar*

[c] *Departamento de Estructuras, Facultad de C. E. F. y N., Universidad Nacional de Córdoba Casilla de Correo 916, 5000 Córdoba, Argentina, spreidik@efn.uncor.edu, http://www.efn.uncor.edu*

[d] *Consejo Nacional de Investigaciones Científicas y Tecnológicas, Av. Rivadavia 1917, CP C1033AAJ, Buenos Aires, Argentina, http://www.conicet.gov.ar*

Palabras Clave: Valgrind, *profiling*, memoria cache, aerodinámica inestacionaria y no-lineal, método de red de vórtices

Resumen. En este trabajo se presenta un análisis de la performance, mediante el uso de la herramienta Valgrind, de un código computacional que implementa un modelo basado en el método de red de vórtices inestacionario y no-lineal con el fin de estudiar el comportamiento aerodinámico de vehículos aéreos no-tripulados con configuraciones de alas unidas y de gran alargamiento. Para realizar el análisis se utilizan las herramientas Callgrind y Cachegrind, las cuales forman parte del framework Valgrind. Callgrind, permite obtener un esquema global y local de las rutinas del código, obteniéndose el porcentaje de tiempo que estas ocupan durante toda la ejecución, la cantidad de veces que son ejecutadas y con cuáles otras rutinas están relacionadas. Cachegrind, trabaja como un simulador de memorias cache L1 y L2 proporcionando contadores de performance en base a la interacción del código con estas memorias, lo que permite identificar problemas de acceso a datos e instrucciones en ciertas regiones del código. Luego, y a partir de una serie de pruebas, se proponen algunas medidas correctivas sobre aquellas rutinas que concentran el mayor porcentaje de tiempo durante la ejecución del programa. Estas medidas logran un incremento en el rendimiento de la aplicación de un 71% sobre la versión secuencial, y se alcanza un speedup de 10,3 sobre la versión paralelizada.

1 INTRODUCCIÓN

Este trabajo está enfocado en el análisis de la performance de un código computacional denominado AeroVANT (Ceballos et al., 2008a, 2008b, 2010). Esta aplicación implementa el método de red de vórtices inestacionario y no lineal (NUVLM) y permite simular el comportamiento aerodinámico inestacionario y no-lineal de vehículos aéreos no tripulados o "Unmanned Aerial Vehicles" (UAVs) con configuraciones de alas unidas y de gran alargamiento. Este tipo de UAVs tienen aplicaciones en actividades civiles, científicas, comerciales y militares. Estas actividades son desarrolladas a grandes altitudes, en régimen de vuelo subsónico, por un tiempo muy prolongado, y a bajo costo.

En lo que respecta a los requerimientos computacionales, numerosas implementaciones "seriales" del NUVLM (Ceballos et al., 2008b, Roccia et al., 2010, Verstrate et al., 2009) mostraron que se insumen cuantiosos recursos de cómputo. El uso de técnicas tales como recortar las estelas o aprovechar la simetría del problema (Katz y Plotkin, 2001, Ravetta, 2005) permiten que las implementaciones computacionales del NUVLM puedan ejecutarse más rápido, a costo de restringir el tipo de problemas que pueden ser atacados. Por ello, uno de los caminos a seguir para lograr mayores velocidades de ejecución consiste en implementar técnicas de performance que permitan aprovechar de una manera más eficiente los recursos computacionales que se disponen al momento de realizar las ejecuciones del código.

En un trabajo previo (Ceballos et al., 2010) destinado a mejorar los tiempos de ejecución, se desarrolló una estrategia de paralelización explícita, cuya implementación computacional permitió obtener un speedup del orden de 3,5 o un incremento del 72 % en el rendimiento de la aplicación, para ejecuciones realizadas con un procesador Intel® QuadCore® de 2,4 GHz ocupando sus 4 threads.

En este artículo se presenta un análisis de performance de la versión secuencial de este código. El análisis es realizado por medio de las componentes Cachegrind y Callgrind de la herramienta de análisis Valgrind (http://valgrind.org/). Estas componentes, permiten identificar las rutinas que consumen el mayor tiempo durante la ejecución del programa, conocer la interacción entre ellas y verificar problemas de acceso a las memorias cache L1 y L2.

Este trabajo está organizado de la siguiente forma: en la sección 2, se presenta una descripción general de la aplicación AeroVANT del NUVLM y del algoritmo que lo implementa; en la sección 3 se presenta la herramienta Valgrind; los análisis de performance están descritos en la sección 4, mostrándose los análisis en base a los tiempos de ejecución del programa en la sección 4.1 y los análisis de interacción con las memorias cache L1 y L2 en la sección 4.2; los resultados se muestran en la sección 5, dentro de la cual se presentan propuestas de mejoras al código (sección 5.1), detallándose en la sección 5.2 la ganancia de performance sobre la versión secuencial, y en la sección 5.3 la ganancia sobre la versión paralelizada; finalmente, algunas conclusiones y observaciones son realizadas en la sección 6.

2 LA APLICACIÓN AEROVANT

2.1 Descripción de la Aplicación AeroVANT

La aplicación AeroVANT consta de tres partes. La principal, es un código que implementa un modelo aerodinámico basado en el NUVLM. Otra parte corresponde a la implementación de un preprocesador capaz de generar automáticamente configuraciones de UAVs de alas unidas, y preparar la geometría de esas configuraciones para que sean tratadas por el código principal de la herramienta. La tercera parte es un código que permite postprocesar los resultados para que puedan, con la ayuda de un software de visualización, representarse gráficamente. En la Figura 1, se muestra un esquema de la organización de la herramienta computacional AeroVANT.

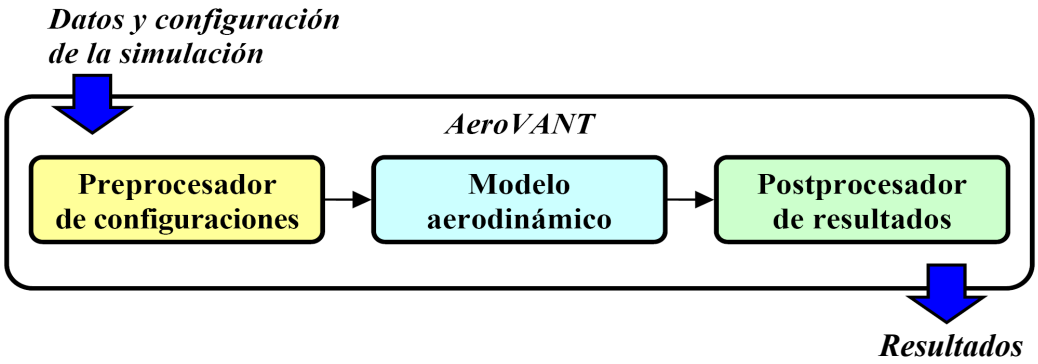

Figura 1: Esquema de la organización de AeroVANT.

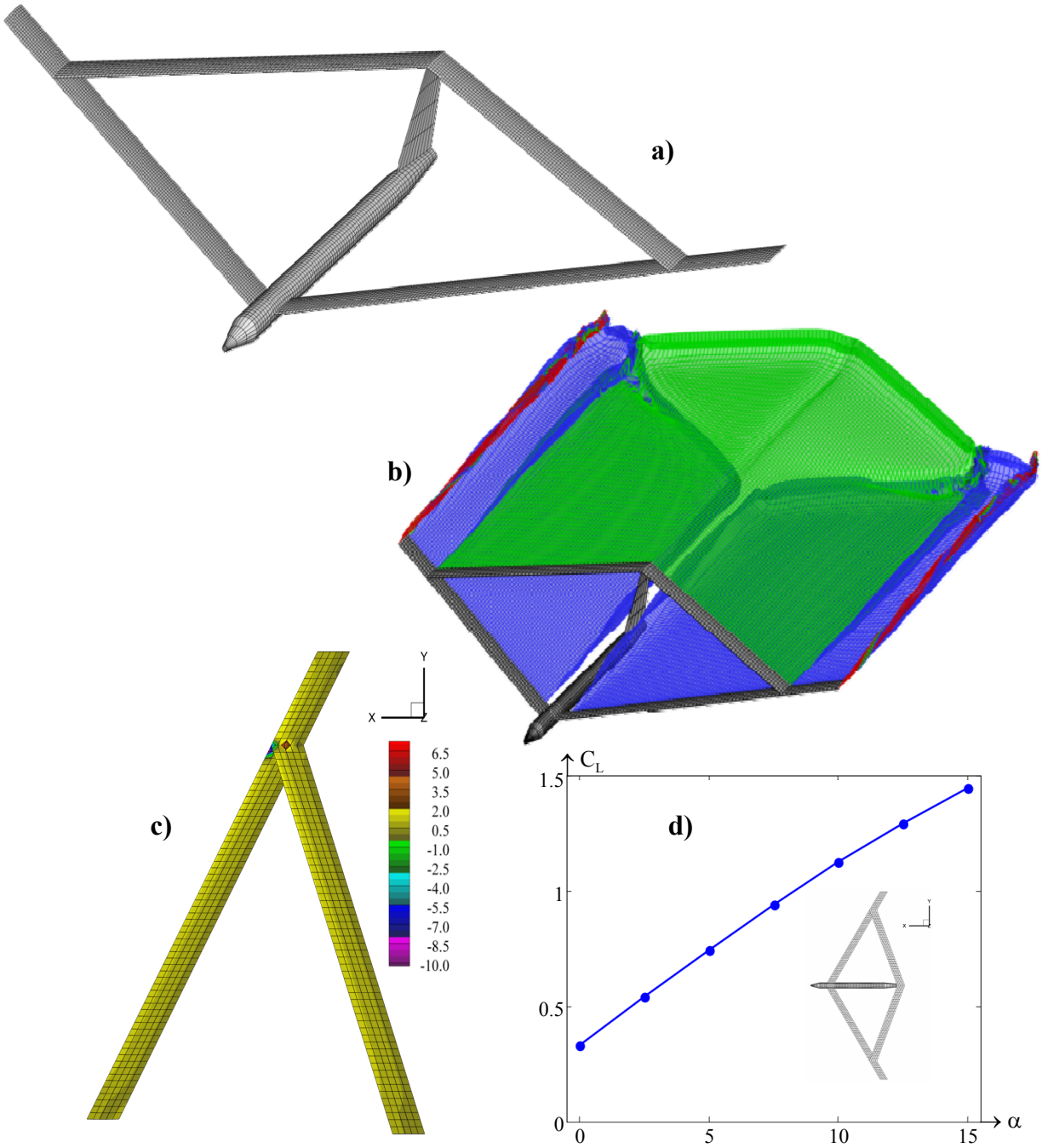

Figura 2: Resultados obtenidos con la herramienta AeroVANT.

En la Figura 2 se muestran algunos ejemplos de resultados obtenidos con el programa AeroVANT. En la Figura 2a, se muestra una malla aerodinámica típica generada para un UAV con una configuración de alas unidas. En la Figura 2b se muestra, para la misma configuración de la Figura 2a, la evolución de las estelas luego de 100 pasos de simulación. En la Figura 2c se presenta, en una vista superior de las alas delanteras y traseras derechas del UAV de la Figura 2a, la distribución del coeficiente de presión sobre estas dos superficies sustentadoras. En la Figura 2d puede observarse el trazado de una curva que muestra la variación del coeficiente de sustentación C_L versus el ángulo de ataque α.

El Preprocesador de configuraciones es un código computacional escrito con el lenguaje de programación Matlab®. Se utilizó ese lenguaje porque los requerimientos computacionales del preprocesador no son elevados. En el caso de la implementación del modelo aerodinámico, los requerimientos computacionales son muchísimo más importantes, por ello el código computacional se escribió utilizando Fortran 95. Aunque el Postproceso de resultados no es exigente en la demanda de recursos computacionales, por simplicidad en la escritura del código, también se utilizó Fortran 95 para su programación.

2.2 El Modelo Aerodinámico

El modelo aerodinámico basado en el NUVLM permite modelar correctamente no-linealidades aerodinámicas asociadas con grandes ángulos de ataque, deformaciones estáticas, y flujos dominados por vorticidad en los que el fenómeno conocido como "vortex bursting" no ocurre. El modelo predice correctamente la emisión de vorticidad desde un cuerpo (o varios), inmerso en el seno de un fluido, hacia el campo del flujo. Esta vorticidad es transportada por el flujo de aire desde el cuerpo hacia el fluido y forma así las estelas. La distribución de la vorticidad en las estelas y la forma de las mismas son, también, parte de la solución del problema. El NUVLM es un método confiable y muy buen predictor de cargas aerodinámicas altamente inestacionarias y no-lineales.

En flujos sobre superficies sólidas donde el número de Reynolds es alto, se genera vorticidad por efectos viscosos en capas muy delgadas, llamadas capas límites, que están pegadas a las superficies del cuerpo. Los efectos viscosos son responsables de la existencia de las capas límites. Parte de esta vorticidad es emitida desde los bordes filosos de los sólidos y transportada por el fluido, formando las estelas. El campo de velocidades asociado con toda esta vorticidad interactúa con la llamada corriente libre: mientras las condiciones de borde de no-penetración y no-deslizamiento son satisfechas sobre las superficies sólidas generadoras de vorticidad, la vorticidad en las estelas se mueve libremente en el fluido de forma tal que no se produzcan saltos de presión a través de las estelas.

El método de red de vórtices inestacionario esta basado en la idea de representar las capas límites y las estelas mediante sábanas vorticosas. Nos referiremos a estos dos tipos de sábanas vorticosas como 'sábanas adheridas' (bound-vortex sheets) y 'sábanas libres' (free-vortex sheets).

El flujo asociado con la vorticidad en la estela cercana al sólido afecta el flujo alrededor del mismo sólido y por lo tanto las cargas actuantes sobre ella. Debido a que la vorticidad presente en las estelas en un instante dado fue generada y convectada desde el ala en un tiempo anterior, las cargas aerodinámicas dependen de la historia del movimiento; las estelas contienen la "historia". El campo de velocidades, asociado con la vorticidad existente en un punto del espacio, decae con la distancia a dicho punto; en consecuencia, a medida que la vorticidad en la estela va siendo transportada flujo abajo, su influencia decrece y por lo tanto se dice que "el historiador" va perdiendo memoria.

Para más detalles acerca del NUVLM pueden consultarse los trabajos de Konstadinopoulos et al. (1981) o Preidikman (1998).

2.3 Algoritmo del Modelo Aerodinámico

Como ya se mencionó, la parte principal de la herramienta computacional desarrollada en este trabajo implementa el modelo aerodinámico basado en NUVLM. En la Tabla 1, se lista un algoritmo del NUVLM que ha sido tomado como base para la implementación computacional. Para más detalles, se recomienda consultar (Ceballos et al. 2008b, 2010).

Modelo Aerodinámico (NUVLM)
1. Lectura y ordenamiento de datos. 2. Cómputo de la matriz de coeficientes de influencia aerodinámicos. 3. Cómputo del lado derecho del sistema de ecuaciones. 4. Solución del sistema de ecuaciones: cálculo de la circulación. 5. Para cada paso de tiempo se realizan los siguientes cálculos: 5.1. Convección de las partículas de fluido (Convect). 5.1.1. Calcular influencias de sábanas vorticosas adheridas (Vbnd). 5.1.2. Calcular influencias de sábanas vorticosas libres -emitidas desde las alas delanteras- (Vwakels). 5.1.3. Calcular influencias de sábanas vorticosas libres -emitidas desde las alas traseras- (Vwakeht). 5.2. Cómputo del nuevo lado derecho del sistema de ecuaciones (Rhs). 5.2.1. Calcular influencias de sábanas vorticosas libres -emitidas desde las alas delanteras- (Vwakels). 5.2.2. Calcular influencias de sábanas vorticosas libres -emitidas desde las alas traseras- (Vwakeht). 5.3. Solución del nuevo sistema de ecuaciones. 5.4. Cómputo de los coeficientes de presión y cargas (Loadsls). 5.5. Almacenamiento de los resultados obtenidos en el paso de tiempo actual.

Tabla1: Algoritmo del modelo aerodinámico.

El primer paso, esto es, el punto 1 del algoritmo de la Tabla 1, consiste en leer y ordenar los datos necesarios para que el código del NUVLM pueda ejecutarse. Los datos leídos en este paso son: *i*) algunos parámetros que permiten configurar la simulación, y *ii*) datos de la geometría de la configuración del UAV debidamente preparados por el código preprocesador.

En el segundo y tercer paso se computa un sistema de ecuaciones algebraicas lineales que tiene como solución los valores de circulación de los anillos vorticosos que conforman la sabana adherida discretizada. Es importante destacar que en esta instancia, debido a que aún no se ha comenzado la emisión de vorticidad desde los bordes de fuga y punteras de las alas, el lado derecho calculado no posee ningún aporte de la velocidad asociada a la sábana vorticosa libre. El aporte de la mencionada velocidad recién aparece a partir del paso 5.2 del algoritmo de la Tabla 1, luego que ha sido realizada la convección en el paso 5.1. Los cálculos de influencias 5.1 se realizan sobre partículas de fluido que pertenecen a las sábanas libres y los cálculos de influencias 5.2 se realizan sobre los denominados puntos de control de la malla aerodinámica.

Los pasos 4 y 5.3 se refieren a la resolución del sistema de ecuaciones algebraicas lineales, para ello se optó por un método numérico basado en la factorización $\mathbf{A} = \mathbf{LU}$. La matriz de coeficientes del sistema de ecuaciones posee todas sus entradas distintas de cero y no es simétrica. Cada vez que se realiza la secuencia de cálculos del paso 5 se usa la misma matriz

del sistema de ecuaciones y sólo se producen cambios en el vector lado derecho. La matriz permanece sin cambios, pues la geometría del UAV es la misma en todo momento, es decir, no se producen deformaciones en el vehículo. El lado derecho cambia en cada ejecución del conjunto de pasos 5, pues la influencia de la velocidad asociada a la sábana vorticosas libre cambia debido al movimiento libre de la mencionada sábana.

En el paso 5.4 del algoritmo de la Tabla 1 se realiza el cálculo de los coeficientes de presión y las cargas aerodinámicas y, por último, en el paso 5.5 se almacenan los resultados. En este paso se realiza el postproceso de resultados y se los preparan para que puedan ser visualizados en las formas en que se muestran en la Figura 2.

3 DESCRIPCIÓN DE VALGRIND

Valgrind es un framework de herramientas de análisis de performance, de licencia GNU, que ayudan a mejorar el rendimiento de las aplicaciones (Nethercote y Seward , 2003). Valgrind simula la ejecución de la aplicación, compilada en base a sus archivos objetos *.o, para un análisis detallado del código. Las aplicaciones ejecutadas bajo este framework, sufren un factor de retardo de 2 a 20 veces o más, dependiendo la herramienta que sea utilizada. Algunas plataformas soportadas por Valgrind son: x86 y AMD64, entre otras (http://valgrind.org/).

Dentro de las utilidades que posee Valgrind se encuentran algunos tipos de debugging, profiling o similares. Las herramientas de tipo profiling, seleccionadas para este estudio de performance, son Cachegrind y Callgrind. Cachegrind, simula cómo el programa AeroVANT interactúa con los niveles de memoria L1 y L2, entregando diferentes contadores de performance en base a esta interacción. Las memorias se simulan como en los actuales procesadores, con dos memorias independientes para L1, correspondientes al cache de datos e instrucciones, y con un segundo nivel de memoria unificada en L2, para datos e instrucciones. Los contadores otorgados por Cachegrind se muestran en la siguiente tabla:

Contadores de Performance en Cachegrind
a) Instruction Fetch
b) Data Read Access
c) Data Write Access
d) L1 Data Read Misses,
e) L1 Data Write Misses
f) L1 Instruction Fetch Misses
g) L2 Data Read Misses
h) L2 Data Write Misses
i) L2 Instruction Fetch Misses

Tabla 2: Contadores de acceso a la memoria cache L1 y L2.

Los contadores a), b) y c) (Tabla 2) muestran el número total de Captura de Instrucciones, Lectura de Datos y Escritura de Datos, respectivamente. Por otro lado, los contadores d) hasta el i) corresponden al número de fallos que se producen en los niveles L1 y L2, producto del acceso a memoria por a), b) y c). Por último, Cachegrind permite modificar las características de las memorias L1 y L2 a simular, en cuanto a tamaño de la memoria, asociatividad y tamaño de línea. Por defecto, toma las características del procesador donde fue instalado. Las ejecuciones realizadas bajo Cachegrind pueden tardar hasta 20 veces del tiempo de ejecución normal.

Otra herramienta tipo profiling utilizada es Callgrind, la cual genera un informe detallado sobre cada rutina del programa, entregando el porcentaje de tiempo que éstas ocupan respecto al tiempo total de ejecución, el número de veces que éstas son ejecutadas y con cuáles otras rutinas interaccionan. El uso de esta herramienta en conjunto con Cachegrind entrega además una información detallada sobre el comportamiento de cada rutina con los niveles de memoria L1 y L2.

Adicionalmente, para este análisis se utiliza la herramienta de visualización denominada KCachegrind (http://kcachegrind.sf.net), la cual permite obtener una mejor visión general de la información generada por Cachegrind y Callgrind (Weidendorfer et al., 2004). KCachegrind transforma la información en tablas y grafos para su fácil interpretación. Este es un software libre y está bajo la licencia GPL V2.

4 ANÁLISIS DE PERFORMANCE CON VALGRIND

En esta sección se presentan distintas mediciones de performance hechas con las herramientas computacionales Valgrind 3.5 y KCachegrind 0.4.6, sobre la versión secuencial del algoritmo AeroVant, escrito en Fortran 95 y utilizando el compilador GNU de Fortran, GFortran.

Las ejecuciones han sido realizadas bajo el sistema operativo Ubuntu 9.04, en un equipo de escritorio con procesador de 64 bits, Intel® Core 2 Quad® Q6600 de 2.40 GHz, el cual posee 4 x (32KB x 32KB) de memoria cache L1, 2 x 4MB de memoria cache L2 y memoria RAM DDR2 de 2Gb y 1066 MHz. Esto significa que, para efectos de análisis de Valgrind sobre la versión secuencial del algoritmo, se utiliza solo un core o núcleo, el cual accede a 32 KB x 32 KB de memoria cache L1, 4MB de memoria compartida L2. También se considera la variable de asociatividad para las memorias del procesador, la cual corresponde a: 8-way associative para ambas L1 y 16-way associative para la memoria cache L2.

En lo que sigue de la sección, se considera un mallado de uso frecuente, de 1425 paneles. También se consideran T = 30 pasos de simulación y se compila el código sin ninguna de las opciones de optimización del compilador GFortran.

4.1 Análisis de la Estructura del Código y Tiempo de Ejecución

En esta sección se muestra la estructura del código, la medición del porcentaje de tiempo que ocupan las principales rutinas del programa y su interacción con otras rutinas, mediante el uso de la herramienta Callgrind. Esto permite obtener un primer enfoque de donde concentrar esfuerzos para lograr una mejor performance.

En la Figura 3 se muestra, a modo de árbol, las rutinas principales del código, el número de veces que estas son ejecutadas o llamadas y el porcentaje de tiempo que ocupan en su ejecución. Este porcentaje, incluye el porcentaje de tiempo propio de la rutina y el de las rutinas hijas. Al analizar la Figura 3, se puede detectar rápidamente que el tiempo de ejecución está dominado por el conjunto de rutinas Vwakels, Vwakeht y Vbnd, con un 98,3 % del total del tiempo de ejecución, al sumar sus porcentajes. Este conjunto de rutinas, tiene como rutinas padres a Shedding, Rhs y Loadsls, las cuales poseen un porcentaje de tiempo del 71, 25,5 y 1,8 %, respectivamente, lo que suma un total del 98,3 % del tiempo de ejecución del código. La rutina Loadsls, fue descartada en la figura por su bajo porcentaje de influencia.

También se puede apreciar, con una simple suma sobre el porcentaje de tiempo del conjunto de rutinas mencionadas, que las rutinas padres, Shedding y Rhs, no tienen un consumo de tiempo significativo por sí solas.

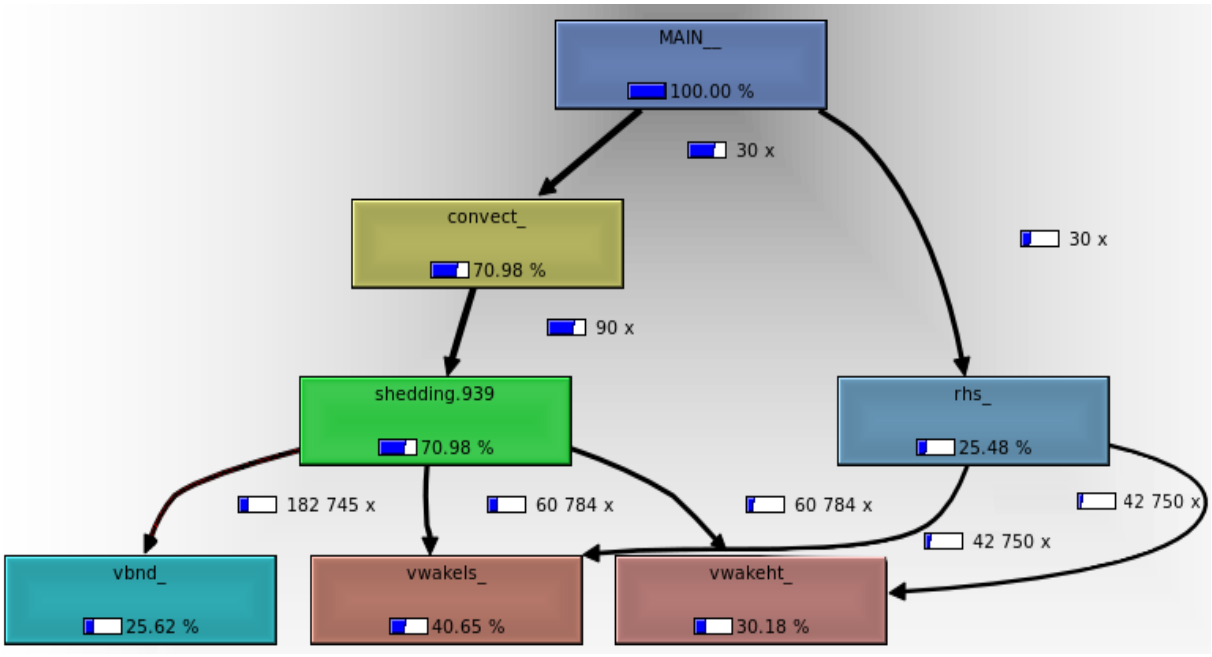

Figura 3: Imagen generada por KCachegrind, de acuerdo a las salidas de Callgrind.

En un análisis al código del conjunto de rutinas Vwakels, Vwakeht y Vbnd, se observa que las tres realizan las mismas operaciones, pero con distintas cantidad de variables y número de veces, por lo que su trabajo como rutinas independientes es una ayuda a la eficiencia del código. También se logra apreciar que las operaciones más usadas por estas rutinas son la multiplicación, suma y resta de matrices y vectores del tipo double y tamaño 3x3 y 1x3, respectivamente. Este tamaño de datos descarta la búsqueda de mejores algoritmos para dichas operaciones. Otra apreciación sobre estas rutinas se refiere a que cada una tiene asociada las tres siguientes funciones: producto cruz entre arreglos (Cross_product), producto punto entre arreglos (Vortxl) y la suma de las componentes de un vector (Unit_vector). En la Tabla 3 y 4, se muestran para estas funciones, el porcentaje de tiempo que ocupan en el transcurso de la ejecución del programa y la cantidad de veces que son llamadas por parte del conjunto de rutinas mencionado.

	Vwakels	Vwakeht	Vbnd
Cross_product	3,57 %	2,66 %	2,32 %
Vortxl	6,86 %	5,10 %	4,70 %
Unit_vector	2,22 %	1,65 %	1,52 %

Tabla 3: Porcentaje del tiempo demorado por cada función durante la ejecución del programa.

	Vwakels	Vwakeht	Vbnd
Cross_product	136.183.310	101.217.325	92.960.625
Vortxl	1.089.673.548	809.738.600	743.685.000
Unit_vector	1.089.673.548	809.738.600	743.685.000

Tabla 4: Número de llamadas que realiza el conjunto de rutinas a cada función.

Se puede apreciar, al sumar todos los porcentajes involucrados en la Tabla 3, que el conjunto de funciones representa un total del 30,6% sobre el tiempo total de ejecución del código. Además, se aprecia la cantidad mayor de veces que estas funciones son ejecutadas, respecto al número de llamados realizados para Vwakels y Vwakeht (103.534) y para Vbnd (201.795), como se muestra en la Figura 3. Esto, junto al tráfico de datos, como matrices o vectores, requeridos para ejecutar cada función, implica un alto costo computacional.

4.2 Análisis de la Memoria Cache L1 y L2

En esta sección se muestran los contadores de performance del algoritmo del NUVLM, sobre las memorias cache L1 y L2, en base a tres tipos de interacciones: la captura de instrucciones "Instruction Fetch", la lectura de datos "Data Read Access" y la escritura de datos "Data Write Access". La ejecución se realiza mediante la herramienta Callgrind en conjunto con Cachegrind, sin ninguna opción de optimización en la compilación del código.

En la Tabla 5, se detalla el número total de Instruction Fetch (IF), Data Read Access (DRA) y Data Write Access (DWA) realizadas por el programa (Main), así como por sus rutinas principales (Vwakels, Vwakeht y Vbnd).

	Main	Vwakels	Vwakeht	Vbnd
IF	4.781.642.100.993	1.952.498.580.164	1.449.890.278.973	1.316.692.748.783
DRA	1.505.222.682.348	623.088.916.654	462.678.589.514	394.882.631.895
DWA	873.992.568.452	359.873.752.370	267.059.488.905	239.562.499.696

Tabla 5: Número total de acceso a memoria, durante la ejecución del programa.

El conjunto de rutinas principales realiza cerca el 99% del total de Instruction Fetch, el 98% del total de Data Read Access y el 99% del total de Data Write Access. Una vez más se muestra que aquí se concentra casi la totalidad del costo cómputo.

En la Tabla 6, se muestran aquellas rutinas que poseen mayores fallas en los accesos a memorias L1 y L2. En estas fallas se considera la suma del total de Instruction Fetch Misses, Data Read Misses y Data Write Misses, por nivel de memoria.

	Main	LUdcmp	VWakels	Vwakeht	Vbnd
L1 miss sum	1.451.000.824	841.859.607	171.787.728	79.250.026	230.391.721
L2 miss sum	108.743.840	100.131.197	31.909	10.306	94.439

Tabla 6: Número total de fallas de acceso a la memoria Cache L1 y L2.

La rutina LUdcmp realiza una descomposición **LU** de la matriz inicial de datos. De la Tabla 6 se infiere que esta rutina contiene el 58% de las pérdidas de acceso a la memoria L1 y el 92% a la L2. Como este proceso se realiza solo una vez en la ejecución del programa, se descartan labores para mejorar la eficiencia. En resumen, a partir de la información dada en las Tablas 5 y 6, la magnitud de cache Misses en todo el programa dista de ser un punto de preocupación en la eficiencia del algoritmo, puesto que se encuentra bajo el 0,02% de las pérdidas. Incluso la concentración de pérdidas en la rutina LUdcmp, no es preocupante, al menos que ésta pase a ser parte frecuente en la ejecución del programa.

5 RESULTADOS

En esta sección se presentan algunas mejoras realizadas sobre el código del NUVLM. También se analiza cómo éstas influyen en la gestión de la memoria cache, en el rendimiento del programa y en la ganancia obtenida sobre una versión paralelizada del código. La versión paralelizada del programa se muestra en detalle en un artículo anterior (Ceballos et al., 2010) y fue desarrollada utilizando las bibliotecas de OpenMP embebidas en el compilador Fortran utilizado, GNU GFortran.

5.1 Propuesta de Mejoras y Revisiones al Código

Del análisis realizado en la sección 4, surgen algunos puntos a tener en cuenta: *i*) Se reonoce a Vwakels, Vwakeht y Vbnd, como el conjunto de rutinas que consume el 98,28% del tiempo de ejecución del programa. *ii*) Al conocer el comportamiento de estas rutinas con las memorias L1 y L2, se descartan los problemas de acceso y *iii*) se detecta un alto flujo de datos en las funciones anidadas a este conjunto de rutinas. A partir de esto, se realizan revisiones y medidas correctivas, exclusivamente en las rutinas Vwakels, Vwakeht y Vbnd, dentro de las cuales se pueden apreciar las siguientes:

A) Se opta por quitar los llamados a las funciones Cross_product, Vortxl y Unit_vector, por escribir sus códigos explícitamente en el conjunto de rutinas mencionado. Al realizar este cambio y analizar dichas funciones, se encuentra la posibilidad de fusionar los loops de las rutinas Cross_product y Unit_vector. Esto conduce a reducir el número de operaciones y comparaciones propias de un loop y evitar la referencia excesiva a datos que se repiten en ambas rutinas.

B) Otro factor que se revisa es el correcto acceso a las matrices que operan dentro del código, puesto que en Fortran el almacenamiento de una matriz en la memoria cache se organiza por columnas, contrario al caso en C/C++ donde el almacenamiento se organiza por filas. También la baja cantidad de problemas de acceso a las memorias L1 y L2, como se muestra en la Tabla 6, se debe en parte al reducido tamaño de las matrices que se operan dentro de estas rutinas, por lo que se descarta la utilización de algoritmos tipo blocking que mejoran la eficiencia de acceso a la memoria cache.

C) Otro punto sencillo a mejorar, pero no menor, es el de evitar las divisiones en el procesador, ya que son costosas. Por lo que se hicieron modificaciones con tal de reducir esta operación, definiendo nuevas variables para calcular explícitamente la inversa de un valor o resultado de una expresión. Este cambio significa una mejora del 5% aproximadamente, respecto a la versión original del código.

D) Por último, de las opciones de optimización propias del compilador, solo se utiliza el nivel de optimización “-O2”.

5.2 Influencia de la Mejora del Código en la Gestión de las Memorias Cache

A continuación se muestran los resultados de las mejoras realizadas en base al tiempo de ejecución del programa y su interacción con las memorias cache. Las mediciones de estas mejoras se realizan sobre la versión secuencial del código, y los pasos de simulación tomados para las comparaciones son: T = 30, 35, 40, 70, 100 y 150.

En la Figura 4, se muestra la primera comparación realizada entre el código original y la versión mejorada. Aunque la opción de optimización “-O2” se propone como una mejora al

rendimiento, ambas versiones se compilan con esta opción con el fin de apreciar las mejoras que fueron realizadas explícitamente en el código. Los resultados muestran que la nueva versión del algoritmo tiene una mejora aproximada del 58% respecto al tiempo de la versión anterior y si contamos que la opción de optimización "-O2" no era usada anteriormente, esta mejora asciende a un 71%. Estos porcentajes se mantiene para un cantidad de pasos superior a T = 30.

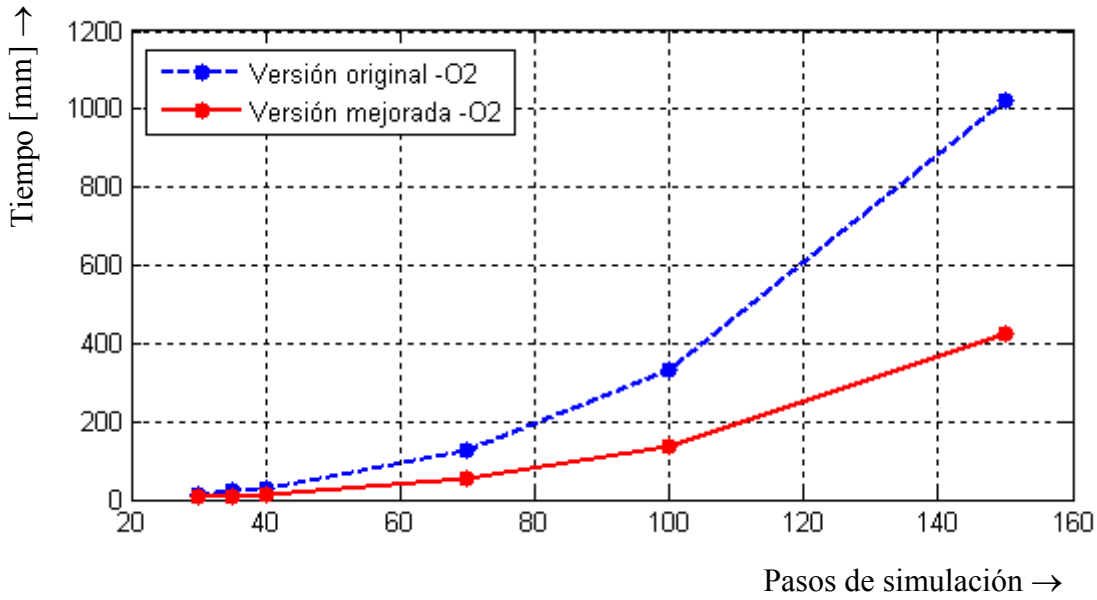

Figura 4: Comparativa de los tiempos de ejecución de ambas versiones secuenciales del código.

En relación al comportamiento del código mejorado sobre las memorias cache, se muestran las siguientes tablas comparativas en base a los contadores de performance provistos por Valgrind. Ambos códigos ejecutados para una cantidad de pasos de T = 30.

La Tabla 7, muestra el número de direcciones de instrucciones almacenadas o Instruction Fetch, para ambas versiones del código. Los resultados de esta tabla están dados para todo el programa (Main) y las tres funciones principales Vwakels, Vwakeht y Vbnd.

Instruction Fetch	Código original	Código mejorado	% de mejora
Main	4.781.642.100.993	1.377.763.268.307	71,2
Vwakels	1.952.498.580.164	559.783.547.375	71,3
Vwakeht	1.449.890.278.973	414.157.968.989	71,4
Vbnd	1.316.692.748.783	385.322.817.993	70,74

Tabla 7: Comparativa del número de Instruction Fetch, para ambas versiones del código.

En las Tablas 8 y 9, se muestra el número total de acceso a las memorias cache por lectura y escritura, respectivamente. La mejora obtenida en todo el programa es de un 77% en el acceso por lectura y de un 75% en el acceso por escritura. El aumento superior sobre este tipo de interacciones respecto al visto en la Tabla 7, se ve reflejado en escribir explícitamente las rutinas Cross_product, Vortxl y Unit_vector en el conjunto de rutinas principales, ya que se elimina la transferencia de las variables de entradas de cada función, por tanto su lectura y escritura.

D. Read Access	Código original	Código mejorado	% de mejora
Main	1.505.222.682.348	339.351.270.344	77,5
Vwakels	623.088.916.654	142.874.484.869	77,1
Vwakeht	462.678.589.514	98.876.593.486	78,6
Vbnd	394.882.631.895	93.054.459.823	76,4

Tabla 8: Comparativa del número de Data Read Access, para ambas versiones del código.

D. Write Access	Código original	Código mejorado	% de mejora
Main	873.992.568.452	217.274.422.542	75,14
Vwakels	359.873.752.370	89.444.061.777	75,15
Vwakeht	267.059.488.905	66.408.364.203	75,13
Vbnd	239.562.499.696	59.873.932.256	75,01

Tabla 9: Comparativa del número de Data Write Access, para ambas versiones del código.

Aunque las rutinas principales Vwakels, Vwakeht y Vbnd, no presentan un número importante de fallas de acceso a memoria L1 o L2, como se mencionó en la sección 4.2, se puede apreciar que las mejoras en el código lograron el decrecimiento de éstas fallas en un 8,9% para el acceso a memoria L1 y de un 0,04% para el acceso a memoria L2. La tabla 10 y 11 muestran los resultados en detalle.

L1 misses	Código original	Código mejorado	% de mejora
Main L1 misses	1.451.000.824	1.382.378.532	8,92
Vwakels L1 misses	171.787.728	124.915.560	50,56
Vwakeht L1 misses	79.250.026	65.500.746	29,62
Vbnd L1 misses	230.391.721	223.545.928	5,11

Tabla 10: Comparativa del número de Data Write Access, para ambas versiones del código.

L2 misses	Código original	Código mejorado	% de mejora
Main L2 misses	108.743.840	108.693.277	0,04
Vwakels L2 misses	31.909	18.416	57,19
Vwakeht L2 misses	10.306	6.644	46,27
Vbnd L2 misses	94.439	90.576	5,31

Tabla 11: Comparativa del número de Data Write Access, para ambas versiones del código.

5.3 Análisis de la Performance sobre la Versión Paralelizada del Código

En esta sección se muestra una comparativa del speedup logrado, en la versión paralelizada del algoritmo, con las mejoras propuestas y sin ellas. El speedup se calcula realizando el cociente entre el tiempo de ejecución del programa secuencial y el tiempo que ocupa el programa paralelizado al utilizar *p* hilos de ejecución o threads.

Las simulaciones se realizaron en un equipo QuadCore descrito en la sección 4, ejecutadas para los $p = 4$ threads disponibles. La Figura 5 muestra esta comparativa para los pasos de simulación T = 30, 35, 40, 70, 100 y 150.

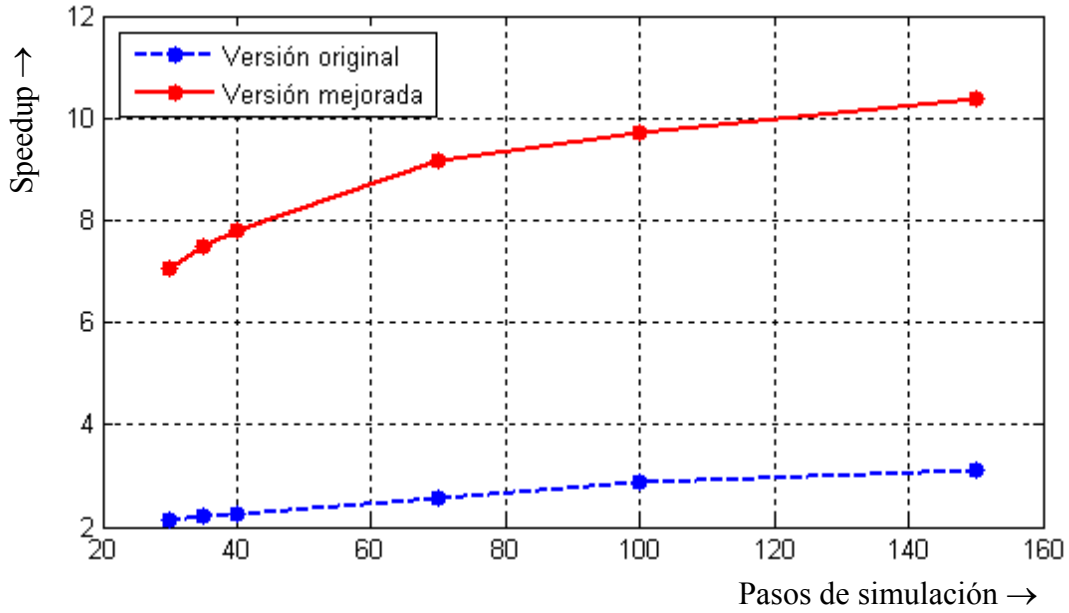

Figura 5: Comparativa del speedup, sobre ambas versiones paralelizadas del código.

Los resultados muestran que el speedup alcanzado para un número de pasos T = 150, es de 10,3 para la versión mejorada y de 3,1 para la versión sin modificaciones. Esto significa una mejora del 70%, aproximadamente, en el rendimiento práctico de la aplicación.

6 CONCLUSIONES

En este trabajo se presentó un análisis de la performance de la herramienta computacional AeroVANT, la cual permite realizar simulaciones del comportamiento aerodinámico inestacionario y no-lineal de vehículos aéreos no tripulados con configuraciones de alas unidas. En trabajos anteriores, investigadores de las Universidades Nacionales de Río Cuarto y Córdoba (Argentina), desarrollaron versiones secuenciales y paralelizadas de un código computacional robusto, cuyo núcleo principal implementa un modelo aerodinámico basado en el método de red de vórtices inestacionario y no lineal.

El análisis de la performance del código AeroVANT se realizó con la asistencia del software Valgrind; un framework que posee herramientas de análisis dinámico y estático. El uso de estas herramientas permitió obtener una mejor comprensión del comportamiento del código durante su ejecución. Este análisis logró:

a) identificar aquellas rutinas que consumen las mayores cantidades de tiempo de ejecución;
b) conocer la interacción de estas rutinas con otras, e identificar el número de llamados; y
c) observar una reducida cantidad de problemas en el acceso a las memorias cache L1 y L2.

El análisis del comportamiento del código, permitió delinear algunas acciones correctivas para modificar la versión original y lograr así, un incremento en su rendimiento. Las acciones correctivas consistieron en:

a) fusionar rutinas, para evitar la excesiva referencia a variables y aprovechar esta instancia para reorganizar bucles internos, con el afán de reducir comparaciones;
b) evitar divisiones directas con la introducción de nuevas variables; y
c) aprovechar la opción "-O2" de optimización del compilador.

La implementación de estas acciones correctivas, permitió obtener mejoras en la versión secuencial del código y en su versión paralelizada, alcanzando incrementos del 71% y del 70% respectivamente. Por último, la mejora en la versión secuencial del código pudo ser apreciada, no solo en el ahorro de tiempo, sino también en el comportamiento del código con los niveles de memoria L1 y L2.

Como trabajo futuro, está previsto realizar mejoras a la rutina que realiza la descomposición **LU** de la denominada matriz de coeficientes aerodinámicos, con el fin de lograr una mayor eficiencia ante el número de pérdidas de accesos a las memorias cache L1 y L2. Las mejoras que se obtengan en la rutina mencionada, impactarán favorablemente en el desempeño de futuras modificaciones de la aplicación AeroVANT, en las que la descomposición pasará a ser una tarea mucho mas frecuente que en la versión actual.

REFERENCIAS

Ceballos L., Barone A., Flores A. y Preidkman S., Desarrollo de una estrategia de paralelización explícita para el método de red de vórtices inestacionario y no lineal. *II Congreso Argentino de Ingeniería Mecánica*. Noviembre, San Juan, Argentina, 2010.

Ceballos L., Preidikman S., y Massa J., Generador paramétrico de geometrías de UAVs de alas unidas orientado al método no-lineal e inestacionario de red de vórtices. *Mecánica Computacional*, 27: 2983-3007, 2008a.

Ceballos L., Preidikman S. y Massa J., Herramienta computacional para simular el comportamiento aerodinámico de vehículos aéreos no tripulados con una configuración de alas unidas. *Mecánica Computacional*, 27: 3169-3188, 2008b.

Katz J. y Plotkin A., *Low-Speed Aerodynamics.* 2nd Edition, Cambridge Aerospace Series, Cambridge, UK, 2001.

Konstadinopoulos P., Mook D.T. and Nayfeh A.H., A numerical method for general unsteady aerodynamics. AIAA-81-1877. *AIAA Atmospheric Flight Mechanics Conference*, August 19-21, Albuquerque, New Mexico, 1981.

Nethercote, N. and Seward, J., Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.

Preidikman S., *Numerical simulations of interactions among aerodynamics, structural dynamics, and control systems*, Ph.D. Dissertation, Department of Engineering Science and Mechanics. Virginia Polytechnic Institute and State University, Blacksburg, VA, 1998.

Ravetta P.A., Desarrollo de simulaciones numéricas para el estudio aeroelástico del control de actitud de generadores eólicos medianos, Tesis de Magíster, Facultad de Ingeniería. Universidad Nacional de Río Cuarto, Argentina, 2005.

Roccia B., Preidikman S., y Massa J., Implementación de un modelo no-lineal e inestacionario para estudiar la aerodinámica de un micro-vehículo aéreo en "hover". *II Congreso*

Argentino de Ingeniería Mecánica. Noviembre, San Juan, Argentina, 2010.

Verstraete, M., Ceballos, L., Preidikman, S., Aviones No-Tripulados Inspirados en el Vuelo Natural con Alas que Mutan: Aspectos Aerodinámicos. *Mecánica Computacional*, 28: 2975-2993, 2009.

Weidendorfer, S., Kowarschik, M. and Trinitis, C., A Tool Suite for Simulation Based Analysis of Memory Access Behavior. *Proceedings of the 4th International Conference on Computational Science (ICCS 2004)*, Krakow, Poland, June 2004.