# UN PROGRAMA GENERAL DE ELEMENTOS FINITOS EN PARALELO

**Gustavo C. Buscaglia, Enzo A. Dari, Adrián J. Lew y Marcelo A. Raschi**
Centro Atómico Bariloche e Instituto Balseiro
8400 Bariloche, Argentina

## RESUMEN

Se describe la implementación de un programa general de elementos finitos utilizando computación distribuida sobre un cluster de tipo Beowulf, y se muestran aplicaciones a fluidodinámica con evaluación de performance.

## ABSTRACT

The implementation of a general finite element system on a parallel environment consisting of a Beowulf-type cluster is described, together with applications to fluid dynamics. Some overall performance and speedup results are reported.

## INTRODUCCION

En este último **MECOM** del milenio es sin duda innecesario abundar sobre la importancia del paralelismo en la mecánica computacional y cálculo científico en general. Vemos cómo los requerimientos tecnológicos llevan a modelos más complejos, con el consecuente crecimiento del número de incógnitas. Las aproximaciones bidimensionales a problemas en tres dimensiones resultan ya insatisfactorias, aunque siguen vigentes desde luego como preliminares.

En la última década hemos visto múltiples aplicaciones del paralelismo en mecánica computacional (ver, por ejemplo, Refs.[10–12]) pero sin duda menor que la que se preveía diez años atrás. El elevado costo de equipos específicamente diseñados para este fin, tales como los IBM SPx, Intel Paragon, Connection Machines, etc., ha contribuido a mantener relativamente acotada esta tecnología a los centros de supercomputación y universidades asociadas. A esto se suma el esfuerzo de mantenimiento y actualización de estos equipos específicos, y la dificultad de justificar este esfuerzo sin una demanda tecnológica elevada y mantenida en el tiempo. Por otro lado, la inercia generada por códigos probados que funcionan en modo secuencial, cuya reprogramación dista de ser tarea fácil, ha pesado en el mismo sentido. No existe, ni se prevé, un compilador que paralelice automáticamente programas originalmente secuenciales. La alternativa más directa para paralelizar un programa es la de utilizar equipos de memoria compartida, en cuyo caso con pequeñas alteraciones y ciertas instrucciones de compilación se puede obtener performances razonables. Lamentablemente, esta alternativa es también la más costosa económicamente, y los equipos tipo PC que funcionan con memoria compartida (e.g., Dual Pentium) tienen un rendimiento decepcionante y reducido número de procesadores.

La estrategia más promisoria en cuanto a portabilidad, independencia del hardware, performance, ampliación y costo, es la *computación distribuida* por *intercambio de mensajes*, en particular implementada sobre *clusters de workstations*. El Proyecto Beowulf[13] ha impulsado esta tecnología con dos condiciones adicionales: equipos de bajo costo tipo PC,

y software de libre acceso, tanto para el sistema operativo como para las bibliotecas de intercambio de mensajes. Los clusters Beowulf ya han brindado resultados importantes en cálculo científico, si bien todavía su uso es escaso en nuestro ámbito de mecánica computacional.

Hacia fines de 1997 decidimos encarar la construcción de programas de elementos finitos aptos para su utilización en clusters Beowulf. El presente trabajo describe los avances realizados y las opciones que surgen durante la implementación, mostrando también una estrategia que funciona, aunque desde luego dista de ser la única.

Existen básicamente dos posibilidades en lo referido al desarrollo de herramientas para la generación de códigos de elementos finitos. La primera es la identificación de los procedimientos clave, comunes a una amplia gama de algoritmos, y su implementación en rutinas de biblioteca que puedan ser utilizadas mediante llamadas desde un programa principal que queda a cargo del usuario. En este caso la programación es, por así decirlo, de alto nivel.

La segunda posibilidad, que fue adoptada como punto de partida en nuestro trabajo y no está tan difundida, consiste en la **abstracción** global de un programa de elementos finitos, llevando un código de base común a un gran número de problemas y algoritmos. En efecto, en un programa de elementos finitos, gran parte del código es independiente del problema específico, el cual sólo aparece en la formulación variacional discreta y se traduce finalmente en una matriz y un segundo miembro elementales a ser ensamblados. La gestión de memoria, la manipulación y resolución de sistemas algebraicos, el input/output, la integración numérica, los mapeos paramétricos, son comunes a una gran cantidad de problemas y pueden ser programados en forma genérica. En particular, las etapas más costosas en términos de memoria y tiempo de CPU pueden ser programadas en paralelo una vez y luego ya no ser alteradas. De esta manera se consigue disminuir el tiempo de generación de códigos, minimizar la brecha entre la Formulación Variacional Discreta (FVD) y el código fuente, eficiencia al aprovechar el paralelismo y evitar que un exceso de generalidad produzca un código lento, adaptabilidad a un gran número de esquemas numéricos y programación de bajo nivel por parte del usuario, sin necesidad de conocer herramientas de paralelismo para utilizarlo.

Para ser más concretos, la conclusión final de este esfuerzo de abstracción es que **todos los programas tienen el mismo** `main`. La diferencia, por ejemplo, entre un programa de fluidos y uno de difusión térmica, se reduce a dos rutinas elementales. Esto nos permitió reutilizar todas las rutinas elementales previas en forma casi directa. Otra ventaja es que un especialista en elementos finitos "secuenciales" no tiene ninguna dificultad en generar un programa paralelo sin saber nada de paralelismo.

La siguiente elección fundamental fue la utilización de bibliotecas especiales para computación distribuida disponibles en forma libre para manejo del algebra lineal y de la partición de la malla. En el primer caso nos decidimos por PETSc (Portable Extensible Toolkit for Scientific Computing[14]), desarrollado en Argonne Natl. Lab., de muy buen soporte técnico y en continua expansión. Otra ventaja es que es desarrollado por el mismo equipo que realizó el `mpich`,[15] la implementación de MPI que utilizamos como interfaz de intercambio de mensajes para el cluster de PCs. Para la distribución del problema entre procesos se escogió METIS,[16] una biblioteca de distribución libre que permite generar la partición del grafo de la malla en $p$ partes de iguales o diferentes tamaños.

## HARDWARE Y SOFTWARE DE BASE

Nuestra implementación se realizó sobre un cluster de computadoras personales interconectadas. Dicho cluster está compuesto por un server y 10 nodos (ver figura 1), contando todos con dos procesadores Pentium II, con reloj de 300 MHz, trabajando en modo de multiprocesamiento simétrico (SMP). El server cuenta además con 512 MB de memoria RAM, 36 GB en discos tipo UWSCSI, placa de red fast ethernet (100 MBps), unidades de disco flexible y cinta, CD-ROM, monitor, teclado y mouse. Los nodos, por su parte tienen 256 MB de memoria RAM instalada, disco rígido IDE de 4.3GB, unidad de disco flexible, placa de red fast ethernet y placa de video tipo VGA. Se cuenta con un único monitor, teclado y mouse para todos los nodos, conectados a través de un switch fast ethernet (3Com SuperStack II Switch 3300), que permite obtener 100 MBit/s pico efectivos en cada puerto (200 MBit/s pico en modo full dúplex).

Para la evaluación de la velocidad de cálculo utilizamos `Linpack`,[8] consistente en la resolución de un sistema de ecuaciones lleno mediante eliminación de Gauss con pivoteo. En la figura 2 se muestran los resultados obtenidos para los nodos de nuestro cluster, para distintos tamaños de matrices y discriminando la utilización de uno o ambos

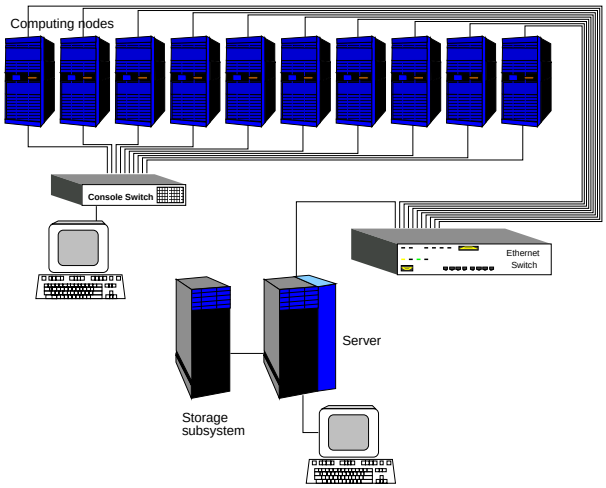



Figura 1: Cluster ppf

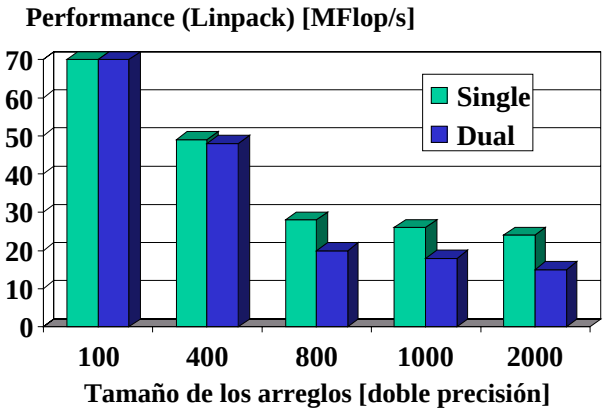



Figura 2: Performance de los nodos.

| Longitud de mensajes | SMP | | Fast Ethernet | |
|---|---|---|---|---|
| | Latencia | Ancho de Banda | Latencia | Ancho de Banda |
| $< 1$ KB | 181 $\mu$s | 384 Mbit/s | 222 $\mu$s | 61.6 Mbit/s |
| $> 15$ KB | 350 $\mu$s | 273 Mbit/s | 367 $\mu$s | 88 Mbit/s |

Tabla 1: Velocidad de transmisión de mensajes utilizando MPI

procesadores. Se observa que cuando el tamaño del problema es grande, de manera que la memoria cache de los procesadores (512 KBytes) no es suficiente, la performance de cada procesador disminuye notablemente, debido a que la limitante pasa a ser la velocidad de transferencia del bus de memoria. El efecto se acentúa para dos procesadores trabajando en modo SMP, ya que en ese caso deben compartir el bus de memoria.

Por otra parte, la utilización de dos procesadores en cada motherboard permite aprovechar el bus de memoria para intercambiar mensajes entre los mismos, obteniéndose una velocidad de transferencia mucho mayor que la de la red ethernet. Esta última puede evaluarse, por ejemplo, utilizando el programa `tcpblast`, que reporta 96 Mbit/s utilizando el protocolo UDP y de 94 Mbit/s con el protocolo TCP. La comunicación bidireccional (Full-Duplex) reporta 96 Mbit/s para el protocolo UDP, mientras que cae a 80 Mbit/s para TCP. La velocidades de transmisión de mensajes, utilizando las rutinas de MPI se muestran en la tabla 1, donde se discriminaron mensajes pequeños (10 Bytes - 1 KByte) y mensajes largos (15 KBytes - 70 KBytes). Estos valores se obtuvieron utilizando la red Fast Ethernet y la memoria compartida de las máquinas duales. La conveniencia de la utilización de múltiples procesadores por máquina es muy dependiente del tipo de aplicación. Si la cantidad o tamaño de los mensajes es muy elevada, puede resultar conveniente la opción de multiprocesadores, mientras que si la relación volumen de cálculo a comunicaciones es muy alta, la red ethernet resulta suficiente y conviene mantener cada procesador con un bus de memoria exclusivo. Los problemas grandes de elementos finitos, con un número moderado de procesadores están ubicados en la segunda de estas situaciones.

El sistema operativo instalado en todas las máquinas es Linux. Además se han instalado los paquetes para intercambio de mensajes `PVM`,[9] `MPI` (más específicamente: `mpich`[15]), el paquete de resolución de ecuaciones diferenciales `PETSc`[14] y el paquete de gestión de procesos DQS.

# GESTION DEL PARALELISMO - PETSC

La mayor parte del tiempo de procesamiento en un programa de elementos finitos se reparte entre el cálculo de las matrices elementales y la resolución del sistema de ecuaciones resultantes. La primera de éstas etapas es trivialmente paralelizable: se asigna un cierto número de elementos a cada proceso, proporcional a la velocidad del procesador donde será ejecutado. La resolución del sistema de ecuaciones, por su parte, requiere de un análisis más cuidadoso.

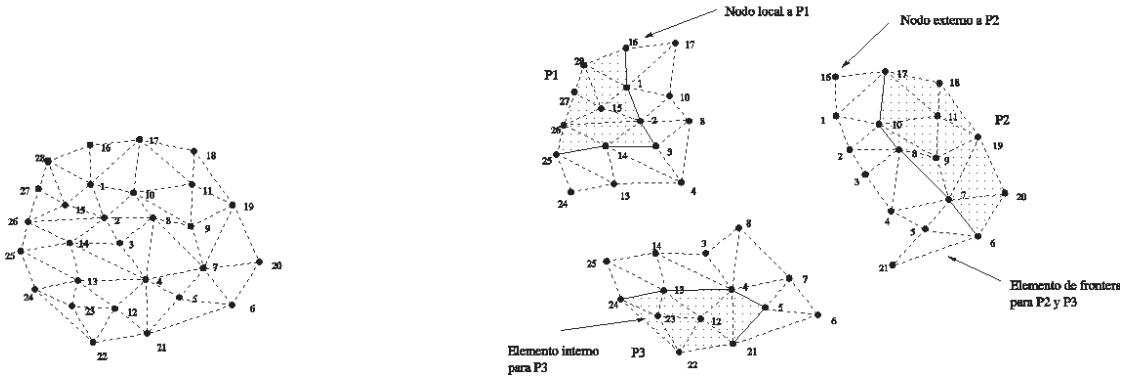



Figura 3: Esquema de distribución de información de una red entre procesadores.

Trataremos con problemas de gran escala, por lo cual utilizaremos métodos iterativos y técnicas de matrices ralas. La matriz del sistema de ecuaciones constituye la mayor parte de la memoria requerida por el programa y deberá ser almacenada por partes en los distintos procesadores, al igual que los vectores segundo miembro y solución. La forma de particionar la matriz determinará el número y tamaño de los mensajes a intercambiar entre los procesadores.

La metodología utilizada en Par-GPFEP para la distribución del problema es como sigue: Se ensambla la estructura de la matriz del sistema de ecuaciones en base a la red de elementos finitos donde se desea resolver el problema, suponiendo una incógnita por nodo. A continuación se determina la partición del problema utilizando algoritmos de partición de grafos multinivel (concretamente, rutinas del paquete `METIS`[16]). De esta manera quedan definidos los nodos de la malla asignados a cada procesador. Es posible asignar a cada procesador una fracción del número de nodos proporcional a su velocidad para mantener el balance de carga. Todos los campos, sean escalares, vectoriales o tensoriales, son almacenados en vectores distribuidos entre los procesadores de la misma manera, de acuerdo a esta asignación inicial de los nodos. De la misma manera las matrices a resolver para cada campo se distribuyen por filas, según la correspondencia de las incógnitas con los nodos de la malla.

En la figura 3 ejemplificamos el proceso de partición para una malla de unos pocos nodos y tres procesadores. Se muestra en la misma la información almacenada en cada procesador: además de los nodos asignados (o nodos internos), se almacenan todos los elementos con algún nodo asignado al procesador y los nodos de dichos elementos. Esta información es necesaria para el cálculo de todas las matrices elementales que contribuirán a las filas de la matriz global asignadas al procesador. De esta manera algunas matrices elementales son calculadas por más de un procesador, pero se evita el intercambio de mensajes durante esta etapa del cálculo. Nótese que, suponiendo regularidad de la malla, una alta relación incógnitas / procesos y un número de incógnitas por nodo fijo, el número de filas de la matriz asignadas a cada procesador es proporcional a su velocidad, de manera que el producto matriz-vector en paralelo será efectuado aproximadamente en el mismo tiempo por todos los procesadores.

Para la resolución del sistema lineal utilizamos el paquete `PETSc`,[14] basado en MPI, y que por lo tanto utiliza el paradigma de programación SPMD. La distribución de datos e incógnitas, según discutimos en párrafos anteriores se realiza en `PETSc` a través de objetos provistos por el paquete, en particular `Vec` (vector distribuido) y `Mat` (matriz distribuida). `PETSc` provee instrucciones específicas para manejar estos objetos, por citar algunas (autoexplicativas): `VecCreate`, `MatCreateAij`, `VecSetValues`, `MatSetValues` `VecAssembly`, `MatAssembly`. La utilización de objetos distribuidos mantiene los detalles de intercomunicación de los procesos aislados del código fuente de elementos finitos. No es preciso programar explícitamente el envío de mensajes.

`PETSc` permite utilizar un conjunto variado de solvers y precondicionadores de manera muy flexible. Mediante los objetos `Vec` y `Mat` ya mencionados, entre otros, se configura un objeto `SLES` (Linear Equation Solver) y la instrucción `SLESSolve` permite resolverlo modificando método y precondicionador a través de argumentos y/o línea de comandos.

# IMPLEMENTACION DE ELEMENTOS FINITOS: PAR-GPFEP

Como ya mencionáramos, para implementar métodos de elementos finitos en entornos distribuidos existen varias

opciones. La opción elegida por nosotros y delineada más abajo se centra en la abstracción de un `main` único, general, válido para una amplia clase de problemas y conteniendo todas las instrucciones específicas de paralelismo. Este `main` deberá compilarse con rutinas FORTRAN 77 estándar que contienen la formulación específica de cada problema. El resultado es un sistema de generación de programas que hemos dado en llamar PAR-GPFEP.[17] Nuestro objetivo es la resolución de problemas de fluidos y de transporte y difusión. PAR-GPFEP se ha utilizado con ecuaciones de calor en materiales compuestos, ecuaciones de Navier-Stokes con diversas formulaciones estabilizadas, modelo k-epsilon de turbulencia,[18] convección-difusión, y ecuación de Reynolds, entre otros.

## Abstracción de un programa de elementos finitos

Supongamos que se tiene un problema de evolución involucrando NFIELDS campos acoplados a ser resuelto por un algoritmo de FRACSTEP pasos fraccionados, cada uno de los cuales es lineal. Consideremos asimismo que todas las incógnitas se interpolan de igual manera ("equal-order formulation") sobre una malla de NODTOT nodos.
Definimos los siguientes arreglos

`Vec v:` vector de valores nodales de los campos incógnita al tiempo n+1.

`Vec va:` vector de valores nodales en el tiempo n, inicializado a la condición inicial al comenzar el programa.

`Vec b(fracstep):` Arreglo de `FRACSTEPS` vectores, con el segundo miembro del sistema lineal de cada paso fraccionado.

`Vec x(fracstep):` Arreglo de `FRACSTEPS` vectores incógnita del sistema lineal de cada paso fraccionado.

`Mat A(fracstep):` Arreglo de `FRACSTEPS` matrices correspondiente al sistema lineal de cada paso fraccionado.

Sea `NFA(i)` el número de campos a resolver en el paso fraccionado `i`, entonces la dimensión de `b(i)` y `x(i)` es `NODTOT*NFA(i)`, que es a la vez el número de filas y columnas de `A(i)`. Un esquema muy simplificado del programa sería:

- Lectura de datos y distribución de la malla en los $p$ procesos.
- Set v ⟵ Initial condition.
- Loop temporal
  1. va ⟵ v  (copia de v sobre va)
  2. Loop sobre pasos fraccionados
     - (A) fijar el sistema lineal a $A(IFRAC)$, $x(IFRAC)$, $b(IFRAC)$
     - (B) $A(IFRAC)$ ⟵ 0, $b(IFRAC)$ ⟵ 0
     - (C) Loop sobre los elementos de la malla
       - (a) llamar a la rutina que calcula la contribución elemental $A_e(IFRAC)$ y $b_e(IFRAC)$ correspondiente a la FVD del paso fraccionado IFRAC.($\diamondsuit$)
       - (b) ensamblar los valores elementales en $A(IFRAC)$ y $b(IFRAC)$
     - (D) resolver $A(IFRAC).x(IFRAC) = b(IFRAC)$
     - (E) procedimiento x2v: actualizar $v$, colocando o sumando los valores de $x(IFRAC)$ en los lugares correspondientes

Se identifican en lo anterior sin dificultad los pasos centrales de una amplia gama de programas de elementos finitos. El boceto anteriormente descripto tiene la particularidad de que, al cambiar la FVD por cambiar el problema a resolver o cambiar el método numérico, lo único que cambia es una zona de programación en la cual existe gran experiencia y que difícilmente pueda cambiar por utilizarse computación distribuida o por la aparición de nuevos algoritmos.

## Programación de una rutina elemental

PAR-GPFEP cuenta con una abstracción de los procedimientos a realizar para la construcción de la matriz elemental, aunque apenas si describiremos esta etapa por ser razonablemente estándar de elementos finitos. Vayamos a un ejemplo concreto sólo para mostrar lo que finalmente programa el usuario. Consideremos la siguiente formulación variacional discreta para el problema de convección-difusión (Galerkin, Crank-Nicholson)
*Hallar* $u_h^{n+1} \in V_h$ *tal que*

$$\int_\Omega \left[ \frac{u_h^{n+1} - u_h^n}{\triangle t} + \frac{1}{2} b \left( \nabla u_h^{n+1} + \nabla u_h^n \right) \right] v_h \, d\Omega + \int_\Omega \frac{1}{2} \kappa \left( \nabla u_h^{n+1} + \nabla u_h^n \right) \nabla v_h \, d\Omega = \int_\Omega f \, v_h \, d\Omega + \int_{\Gamma_1} g \, v_h \, d\Gamma \qquad \forall v_h \in \dot{V}_h \tag{1}$$

donde $\triangle t$ es la longitud del paso de tiempo, $b$ el campo de velocidad, $\kappa$ la constante de difusión, $f$ una fuente volumétrica y $g$ el flujo difusivo entrante por el sector $\Gamma_1$ de la frontera.

En este caso, al implementarlo en PAR-GPFEP consideraremos que hay dos campos, de forma tal de poder leer el campo $b$ de un archivo y utilizarlo en la FVD, sin embargo el campo $b$ no se resuelve nunca. En el único paso fraccionado presente en este algoritmo se resuelve únicamente $u_h^{n+1}$, y $b$ es una incógnita ficticia, nunca se la modifica. Esto quiere decir que la matriz $A(1)$ tendrá sólo `NODTOT` filas y columnas, de ninguna manera se armará una matriz de $2\times$`NODTOT` filas y columnas con `NODTOT` ecuaciones redundantes.

El lugar de cada rutina elemental en que el usuario debe programar la formación del residuo y la matriz elemental se encuentra en un bucle sobre los puntos de Gauss del elemento. En cada paso del bucle deben sumarse los aportes de dicho punto de Gauss a las integrales sobre el elemento. Entonces, considerando $\kappa$, $f$ y $g$ constantes conocidas, los bloques a programar por el usuario son:

**En la rutina elemental sobre la malla de volumen:**

```
C---------------------------------------------------------------------
      rk    = parmat(1)
      ff    = parmat(2)

C Elementary Right Hand Side
      do i=1,nod
         rhse(i) = rhse(i) + dj*p(i)*(vara(1,1)/delcon + ff)
         do isd=1,nsd
            rhse(i) = rhse(i) +
     *         dj*(-0.5*vara(isd,2)*dvara(1,isd,1)*p(i)-
     *         0.5*rk*dvara(1,isd,1)*dp(i,isd))
         enddo
      enddo

C Elementary matrix
      do i=1,nod
         do j=1,nod
            ae(i,j) = ae(i,j) + p(i)*p(j)/delcon*dj
            do isd=1,nsd
                ae(i,j) = ae(i,j) +
     *                    dj*(0.5*vara(isd,2)*dp(j,isd)*p(i) +
     *                    0.5*rk*dp(j,isd)*dp(i,isd))
            enddo
         enddo
      enddo
C---------------------------------------------------------------------
```

**En la rutina elemental sobre la malla de superficie:**

```
C---------------------------------------------------------------------
      gg = parmat(3)
      if(igr.eq.1) then
         do i=1,nod
            rhse(i) = rhse(i) + gg*p(i)*dj
         enddo
      endif
C---------------------------------------------------------------------
```

donde se utilizaron las siguientes variables que están a disposición del usuario:

`vara(ico, ivar)`: valor de la componente `ico` de la variable `ivar` en el punto de Gauss actual y en el último paso de tiempo.

`dvara(ico, isd, ivar)`: derivada en la dirección `isd` de `vara`.

`p(inod)`: valor de la función de forma en el punto de Gauss actual, con valor 1 en el nodo `inod` y cero en los restantes.

`dp(inod, isd)`: derivada de `p(inod)` en la dirección `isd`.

`parmat`: arreglo que contiene valores de parámetros materiales que se dan en el archivo de entrada.

`nod:` número de nodos del elemento.

`dj:` valor del peso del punto de Gauss actual multiplicado por el valor del jacobiano de la transformación que mapea el elemento actual desde el elemento de referencia.

`delcon:` valor del paso de tiempo.

`ae(i,j):` matriz elemental.

`rhse(i):` residuo elemental.

`igr:` número de grupo al que pertenece el elemento actual.

Existen otros datos disponibles por el usuario, pero no se listan por razones de brevedad.

En las programaciones anteriores puede verse que

1. Las líneas en la que se suma sobre `rhse(i)` y sobre `ae(i,j)` son una expresión prácticamente literal de (1).
2. El programa pone a disposición del usuario todas las variables, por lo que la generación del código involucra apenas 20 líneas de programación.

Al compilar las líneas anteriores con la biblioteca de PAR-GPFEP se genera un código paralelo de elementos finitos para la aplicación prevista, con el método numérico planteado (Galerkin, Crank-Nicholson) y sin restricción sobre el número de dimensiones espaciales. Notar que hasta aquí no se mencionó una palabra de paralelismo, únicamente se dijo que al compilar se genera un código paralelo. Esto indica que efectivamente no es necesario involucrarse con las bibliotecas de Message-Passing para generar los códigos paralelos mediante PAR-GPFEP. El lector puede deducir la variedad de esquemas numéricos que es posible implementar manipulando adecuadamente los datos listados.

# EJEMPLOS DE APLICACION

## Validación de un resolutor de Navier-Stokes 3D

Este programa fue generado con anterioridad a la instalación del cluster Beowulf. Demuestra el comportamiento de PAR-GPFEP en infraestructuras no diseñadas específicamente para paralelismo, concretamente una red de 7 procesadores heterogéneos (4 Pentium 120 MHz, 1 Pentium 200 MHz y 2 PentiumPro de 200 MHz). La red de conexión era tipo ethernet de 10 Mbit/s (10 veces más lenta que el cluster Beowulf actual).

El esquema numérico utilizado, conocido como SPGP se debe a Codina y Blasco[7] y agrega a las incógnitas velocidad y presión, aunque desacoplada a través de fraccionamiento del paso temporal, una incógnita vectorial adicional, correspondiente a la proyección del gradiente de presión.

Para validar y evaluar el método computacional en casos tridimensionales escogimos calcular la cavidad cúbica. El problema consiste en un cubo lleno de fluido en el que en una de las caras se impone una velocidad constante en dirección paralela a una de las aristas, y en las restantes caras se mantiene la condición de no deslizamiento.

Además del sentido de circulación principal en planos ortogonales al eje $y$, existe una circulación desde el plano $y = \frac{a}{2}$ a las paredes laterales y viceversa, lo que indica que el flujo es completamente tridimensional. Ni siquiera en el plano de simetría se puede obviar la tridimensionalidad del flujo, pues las paredes laterales actúan frenando la velocidad del mismo. Esto se refleja en que para un dado $Re$ (con $Re = Ud/\nu$, donde $U$ es la velocidad en la cara superior y $d$ es el lado del cubo), las velocidades en la cavidad cuadrada son mayores y con un perfil diferente que las velocidades en el plano de simetría de la cavidad cúbica.[1] Koseff[3–5] realizó experimentos en una cavidad con $a/d = 3:1$ y encontró que el flujo en el plano de simetría sigue siendo diferente y más lento que el correspondiente a la cavidad cuadrada.

La complicada estructura tridimensional del flujo, esquematizada en la figura 4, provoca que ya para bajos valores de $Re$, como ser $Re = 1000$, las mallas que deban utilizarse sean bastante finas, teniendo en cuenta singularidades cerca de algunas de las aristas.

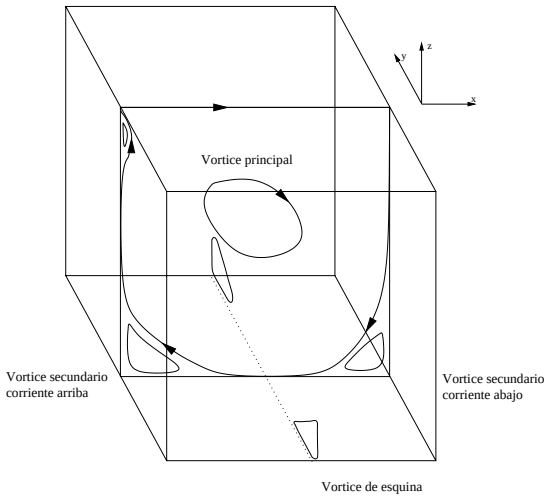



Figura 4: Esquema de las estructuras principales del flujo.

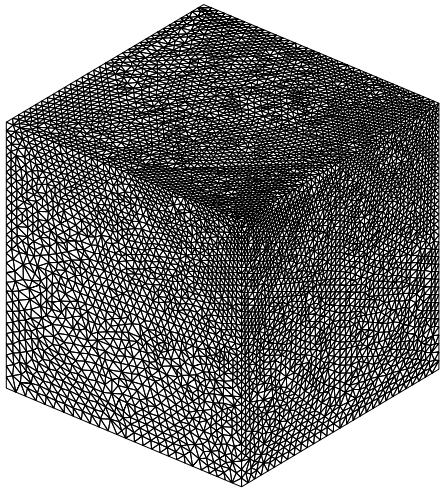
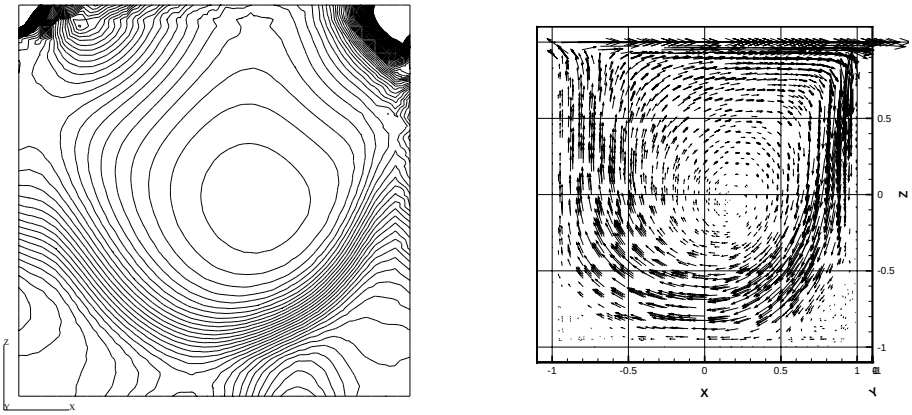


Figura 5: Malla utilizada, compuesta de 50979 nodos y 271265 tetraedros.




(a) (b)

Figura 6: $Re$=1000. Contornos de presión (a) y proyección de la velocidad (b) en el planos y=0.

En nuestro caso utilizamos la malla que se observa en la figura 5, de 50979 nodos y 271265 tetraedros lineales para resolver los casos de $Re = 1000$ y $Re = 3200$, de los cuales mostramos el primero de ellos por brevedad. El número total de incógnitas es de 356853. El método de resolución del sistema de ecuaciones de Navier-Stokes fue GMRES(30) precondicionado con Block-ILU(0), utilizando aproximadamente un total de 450 Mb de memoria RAM. Cada paso demoró entre 3 y 6 minutos.

**Caso** $Re = 1000$**:** Se halla un estado estacionario cuyos contornos de presión y proyección de la velocidad en el plano medio $y$, se muestran en la figura 6, en la que se puede observar que nuestra malla aún es un poco gruesa. Sin embargo, la comparación entre los contornos de presión y las proyecciones de la velocidad coinciden muy bien cualitativamente con los resultados presentados en[6] ,[2] y.[1] En estos trabajos al igual que en el nuestro se pueden distinguir las estructuras principales del flujo, compuestas por el vórtice central, el vórtice secundario corriente abajo, el vórtice secundario corriente arriba y los vórtices de las esquinas. Existe también una pequeña recirculación en la zona superior de la cara $x = 0$. Los vórtices de las esquinas son los principales responsables de que el efecto del frenado de la pared se traslade al vórtice central, entre otros efectos que se estudian en Ref.[3]

## Otros ejemplos: Navier-Stokes, Convección y Difusión

El mismo solver de la sección anterior fue aplicado a flujo arterial en una colaboración con el Laboratorio Nacional de Computación Científica de Brasil. Algunos resultados pueden verse en la figura 7.

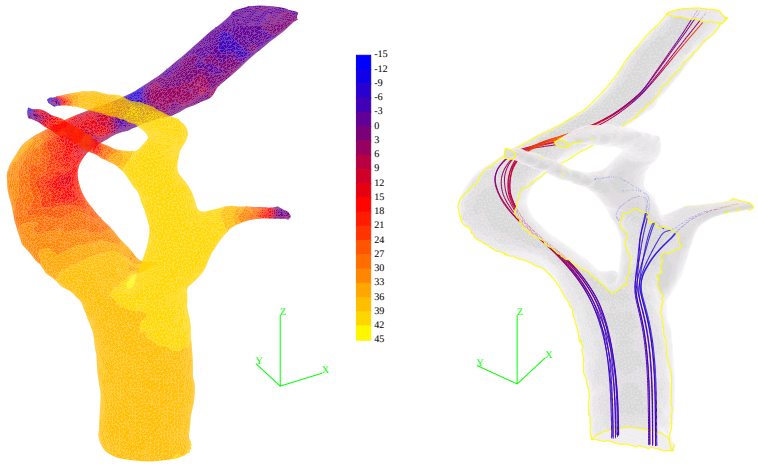


Figura 7: Contornos de presión (izquierda) y algunas trayectorias (derecha) en la arteria carótida.

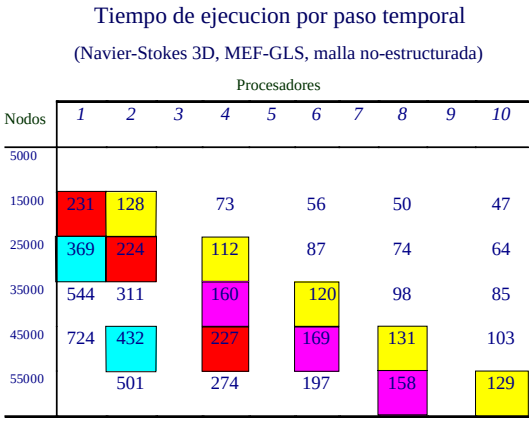



Figura 8: Tiempo de cálculo por paso temporal.

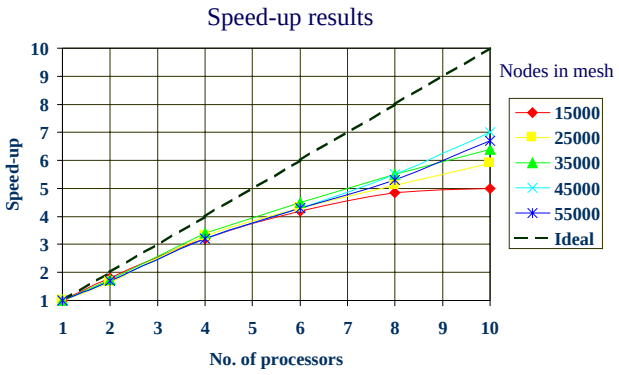
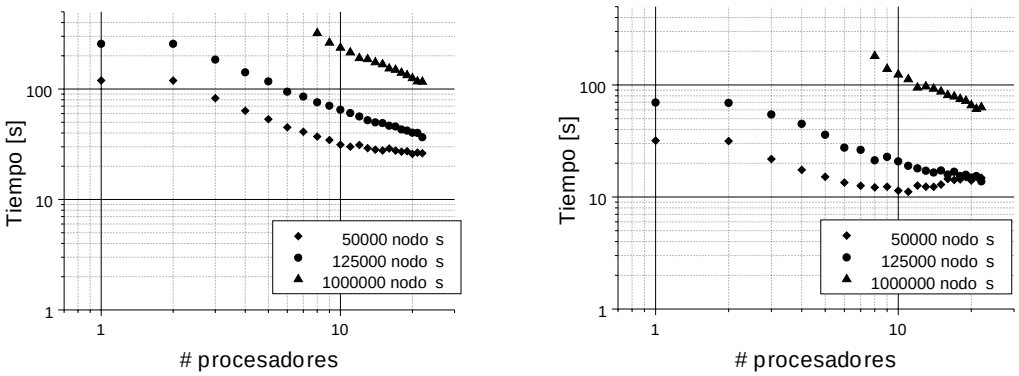



Figura 9: Speedup para distintos tamaños de malla.

En PAR-GPFEP se implementó asimismo estabilización GLS para Navier-Stokes que es más eficiente que el método SPGP para tratar estacionarios. Se realizó un estudio relativamente detallado de performance para el caso de flujo de corte alrededor de una esfera. El interés en este problema proviene de la evaluación de coeficientes de lift en partículas (aerosoles, sedimentos). En la figura 8 se muestran resultados de tiempo de ejecución en función del tamaño de la malla y del número de procesadores, mientras que la figura 9 muestra las curvas de speedup correspondientes. Se observa que al aumentar el número de procesadores se tiende a una saturación del speedup, esto es debido a que también se produce un aumento en el número de mensajes intercambiados.

Por último, incluimos un análisis de tiempos de resolución para problemas de convección - difusión. Las corridas fueron realizadas en el cluster Beowulf descrito en la segunda sección, utilizando tanto el server como los nodos del mismo. Los procesos se asignaron de a pares a ambos procesadores de un mismo motherboard. Se trataron casos con 1000000, 125000 y 50000 nodos. Para el primer caso, el mínimo número de procesadores necesarios fue 8, debido a los requerimientos de memoria RAM del problema. En la figura 10 se muestran los tiempos correspondientes a la resolución del problema total y a la resolución del sistema lineal, en función del número de procesadores utilizados. No se muestran las curvas de tiempos de ensamblaje de la matriz por corresponder a un speedup prácticamente ideal. Esto es debido a que en nuestra implementación durante el ensamblaje no hay intercambio de mensajes entre los procesadores. El proceso de resolución del sistema lineal, en cambio, involucra un gran intercambio de información, lo que genera la aparición de un número óptimo de procesadores para cada caso. En la Fig. 10 puede observarse que para 50000 nodos la reducción de tiempo por paralelización satura para del orden de 8 procesadores. Para 125000 nodos esto ocurre para $\sim 14$ mientras que para $10^6$ nodos hasta 20 procesadores se siguen observando mejoras. Cuando el número de procesadores es inferior a estos límites se observa en los gráficos logarítmicos una zona con comportamiento ideal (pendiente $= -1$). Notar el nulo speedup al pasar de un procesador a dos, lo que evidencia el bajo rendimiento del Dual Pentium en SMP para problemas de tamaño superior al cache.




Tiempos Totales Resolución del Sistema Lineal

Figura 10: Tiempos utilizados en la resolución de los problemas, en función del número de procesadores.

# AGRADECIMIENTOS

Este trabajo contó con soporte financiero a través del PICT 97 No. 12-000000-00982 de la Agencia Nacional de Promoción Científica y Tecnológica. EAD y GCB pertenecen a la Carrera del Investigador del CONICET.

# Referencias

[1] G. Guj and F. Stella, *J. Comp. Phys.*,**106**, 286-298 (1993).

[2] B. Jiang, T. Lin and L. Povinelli, *Comp. Meth. Appl. Mech. Eng.*,**114**, 213-231 (1994).

[3] J. Koseff and R. Street, *J. Fluids Eng.*,**106**, 385-389 (1984).

[4] J. Koseff and R. Street, *J. Fluids Eng.*,**106**, 390-398 (1984).

[5] A. Prasad and J. Koseff, *Physics of Fluids A*, **1**(2), 208-218 (1989).

[6] L. Tang, T. Cheng and T. Tsang, *Int. J. Num. Meth. Fluids*,**21**, 413-432 (1995).

[7] R. Codina and J. Blasco, "Stabilized finite element method for the transient Navier-Stokes equations based on a pressure gradient projection", Comp. Meth. Appl. Mech. and Engn., to appear.

[8] `http://www.netlib.org/benchmark/linpack-pc.c`.

[9] `http://www.epm.ornl.gov/pvm/pvm_home.html`.

[10] W. G. Habashi, Editor, *Solution techniques for large-scale CFD problems*, CMAS: Comp. Meth. in Appl. Sci., Wiley, 1995.

[11] M. Papadrakakis, Editor, *Parallel solution methods in computational mechanics*, Wiley Series in Solving Large-Scale Problems in Mechanics, Wiley, 1997.

[12] K. D. Papailiou *et al*, Editors, *Computational Fluid Dynamics'98*, Proceedings of ECCOMAS'98, Wiley, 1998.

[13] Beowulf Project at NASA Goddard Space Flight Center, `http://cesdis.gsfs.nasa.gov/linux/beowulf/`.

[14] `http://www.mcs.anl.gov/petsc/`. Por S. Balay, W. Gropp, L. Curfman McInnes y B. Smith.

[15] W. Gropp and E. Lusk, "User's guide for `mpich`, a portable implementation of MPI", Math. and Comp. Sci. Div., Argonne Natl. Lab. (1996).

[16] G. Karypis and V. Kumar, "METIS, unstructured graph partitioning and sparse matrix ordering system", Dept. Comp. Sci., Univ. Minnesota (1995). `http://www.cs.umn.edu/ karypis`.

[17] A. Lew, *El método de elementos finitos en entornos computacionales de alta performance*, Trab. Final, Inst. Balseiro, 1998.

[18] A. Lew, G. C. Buscaglia and P. Carrica, "A robust equal-order finite element formulation for the k-epsilon turbulence model", submitted (1998).