Análisis y evaluación del modelo DSM en algoritmos numéricos: UPC versus MPI

J.V. Espí
Dept. de Ciencia de la Computación e Inteligencia Artificial
Universidad de Alicante
E-03071 Alicante
jvespi@gmail.com

H. Migallón
Dept. de Física y Arquitectura de Computadores
Universidad Miguel Hernández
E-03202 Elche
hmigallon@umh.es

V. Migallón, J. Penadés
Dept. de Ciencia de la Computación e Inteligencia Artificial
Universidad de Alicante
E-03071 Alicante
violeta@dccia.ua.es
jpenades@dccia.ua.es

Resumen

En este trabajo se hace un estudio de la viabilidad del modelo DSM (Distributed Shared Memory) a través del lenguaje UPC. Para ello se han implementado, tanto en UPC como en MPI, diversos algoritmos relacionados con la computación matricial y la resolución de sistemas no lineales. Los resultados obtenidos ilustran el rendimiento y productividad de UPC frente a MPI en dos arquitecturas paralelas.

1. Introducción

La investigación en técnicas de paralelización en computación es una necesidad que viene dándose desde hace décadas. Los esfuerzos en la construcción de procesadores rápidos mediante el desarrollo de circuitos electrónicos cada vez más veloces y eficientes ya hace tiempo que demostró ser insuficiente. Desde un principio surgió la necesidad de disponer de una capacidad de cálculo extraordinaria que con la tecnología electrónica de entonces (y la de ahora) no se podía satisfacer. Numerosos problemas del ámbito científico requieren escalar el procesamiento mucho más allá de las capacidades de un procesador, añadiéndose en la actualidad otras necesidades de cálculo que van más allá del ámbito científico. En sus inicios, pocas organizaciones podían permitirse disponer de un superordenador. Este equipamiento solía basarse en un modelo de computación vertical, donde unos pocos procesadores de altísimas prestaciones proporcionaban un gran rendimiento. Por otro lado, han proliferado otros modelos de procesamiento, digamos, basados en computación horizontal, donde la potencia de cálculo se logra a base de distribuir la carga de trabajo entre un número de procesadores mayor, aún no teniendo cada procesador las altas prestaciones del modelo anterior. Esto permite abaratar los costes y acercar la supercomputación a un número mayor de organizaciones educativas, empresariales, etc.

La computación paralela es una disciplina ya consolidada que cuenta con un desarrollo maduro. Dentro de este ámbito, existen dos paradigmas básicos: el modelo de memoria compartida y el modelo de memoria distribuida. El objetivo de este trabajo es, a través de la revisión de estos modelos, profundizar en el análisis del paradigma híbrido de los dos anteriores conocido como el modelo de Memoria Distribuida Compartida (DSM, Distributed Shared Memory) [2]. Concretamente, la investigación se centra en el uso del lenguaje de programación UPC (Unified Parallel C, http://upc.lbl.gov/) y la infraestructura software GASNET (Global Address Space Network, http://gasnet.cs.berkeley.edu/). En la Sección 2 se hace una breve descripción del

modelo DSM y el lenguaje de programación UPC. En la Sección 3 se muestra la investigación experimental llevada a cabo. El estudio se ha basado en el desarrollo mediante UPC y MPI de diferentes algoritmos relacionados con la computación matricial y la resolución de sistemas no lineales y su análisis y comparación en diversos entornos paralelos. Por último, en la Sección 4 se presentan las conclusiones obtenidas en este trabajo.

2. El modelo DSM. UPC

Como su nombre indica, el modelo DSM es un híbrido entre el modelo de memoria compartida y el de memoria distribuida, tratando de aunar las ventajas de cada uno de ellos. En esencia, es bastante parecido al paradigma de memoria compartida, ya que la comunicación entre procesos se define en el lenguaje mediante accesos a memoria compartida. Este diseño es en realidad una abstracción que se crea en la implementación, sea software o hardware, ya que a un nivel inferior encontramos que algunos accesos a memoria no se realizan localmente, sino que tiene que participar una interfaz de comunicaciones que permita direccionar todo el mapa de memoria. Mientras que en memoria distribuida la comunicación entre procesos se realiza intercambiando mensajes entre los procesadores, y en memoria compartida se realiza compartiendo el espacio de memoria, en DSM también se efectúa compartiendo el espacio de memoria, pero de forma virtual. Así, cuando se desea referenciar una variable que se encuentra en una posición de memoria asociada a otro procesador, a un nivel inferior se realiza una comunicación para obtener esa variable. Dicho de otra manera, estamos utilizando el modelo NUMA (Non Uniform Memory Access).

En los últimos años se han desarrollado diferentes soluciones software para implementar la infraestructura NUMA en un cluster de ordenadores. De todas las existentes se ha elegido una cuyo grado de madurez, soporte y mantenimiento evolutivo es suficientemente sólido. Su nombre es UPC (Unified Parallel C) y técnicamente es una extensión del lenguaje ISO C 99 que proporciona rutinas para la programación paralela. Al igual que OpenMP y MPI, implementa el paradigma SPMD (Single Program Multiple Data) y las características fundamentales que podemos destacar son [6]:

- Modelo explícito de computación paralela: Se definen claramente tareas independientes que se ejecutan en los procesadores, donde cada una de ellas ejecuta el mismo código. En terminología UPC estas tareas se denominan threads. La especificación del lenguaje no indica si existe correspondencia entre los UPC-threads y los propios del sistema operativo. Esto quiere decir que cada implementación puede libremente definir los threads como procesos y/o hilos de ejecución.
- Espacio de direcciones compartido: Existen dos tipos de datos disponibles para los threads en UPC, el público y el privado. Esta distinción se logra introduciendo una palabra clave llamada shared, de forma que una variable compartida está visible por cualquier thread en ejecución. Una misma dirección compartida se refiere a la misma posición en memoria física desde cualquier procesador. Por otra parte, al basarse las comunicaciones en variables compartidas, las comunicaciones son unilaterales, lo que redunda en la sencillez del código; esto tiene como contrapartida que no se dispone del sincronismo implícito que supone el establecimiento de comunicaciones bilaterales.
- Primitivas de sincronización: UPC no asume ningún tipo de sincronización implícita en las interacciones entre los threads. Cualquier tipo de sincronización debe ser realizada explícitamente mediante bloqueos (locks), barriers y memory fences.
- Modelo de consistencia de memoria: Se definen dos modelos de acceso a memoria, estricto (strict) y relajado (relaxed). En el estricto, se garantiza que las variables compartidas dentro de un thread se referenciarán en el orden codificado, evitando cualquier tipo de optimización o pipelining que pudiera realizar el compila-

dor. Además provoca la inmediata actualización de cualquier escritura en una variable compartida para hacerla visible al resto de threads. En el modelo relajado, nada de esto ocurre, y por lo tanto podemos obtener rendimientos mejores.

El origen del lenguaje UPC arranca a finales de los 90, cuando un consorcio de organizaciones académicas, institucionales y empresariales de EE.UU. decide crear un nuevo lenguaje DSM. Como resultado se obtiene en febrero de 2001 la primera especificación del lenguaje. Posteriormente, en marzo de 2003 se definiría la versión 1.1 hasta llegar a la actual, la versión 1.2 liberada en mayo de 2005. Se han desarrollado múltiples implementaciones del lenguaje, entre las que destacamos la de Berkeley (BUPC), Michigan Technology (MuPC), GCC UPC, HP UPC, IBM XL Compilers y Cray UPC. Específicamente, en este trabajo se profundizará en la implementación del lenguaje realizada por el Laboratorio de Berkeley en la Universidad de California [6].

3. Análisis y experimentación

En esta sección se explican y analizan los experimentos llevados a cabo durante la investigación. Para dicha experimentación se ha contado con dos máquinas paralelas diferentes, y según las pruebas realizadas se ha empleado una de ellas o las dos. Picolo se trata de un cluster que consta de 24 nodos, donde cada nodo dispone de dos procesadores AMD Opteron 2214 con dos núcleos cada uno y 4 GB de RAM. La velocidad del reloj es de 1 Ghz. Las redes de interconexión son Gigabit Ethernet e Infiniband, aunque esta última no estaba disponible por lo que se empleó Ethernet. El computador denominado Bi-quad es una máquina simétrica con dos procesadores Intel Xeon E5320 con 4 núcleos cada uno y 8 GB de RAM. La velocidad del reloj es de $1,86$ Ghz.

A lo largo de los distintos apartados se exponen los algoritmos implementados, analizando el rendimiento de UPC frente a MPI. Concretamente, los indicadores que se han analizado son tiempos totales de ejecución, grado de escalabilidad, el comportamiento en configuración SMP (Symmetric Multi-Processing) y la productividad del código.

3.1. Ancho de banda

En esta primera experimentación se ha medido el rendimiento de la red de interconexión mediante el entorno de ejecución de UPC-GASNET. Para ello se implementó un algoritmo de *ping-pong* tanto en MPI como en UPC, que consistía en transferir un mensaje de tamaño variable entre dos nodos del cluster y medir el tiempo consumido en viajar en la ida y vuelta. Por tanto, el mensaje se transmitiría dos veces. Los códigos se ejecutaron sobre el cluster Picolo, dada la necesidad de disponer de comunicaciones inter-nodos a través de una red. En una primera versión, se utilizaron en UPC las rutinas de transferencia por bloques en versión síncrona, es decir, las del estándar UPC. Las comunicaciones se establecen unilateralmente, es decir, que no es necesario definir un receptor en el thread destino, de ello ya se encarga el entorno de ejecución de UPC. Para el caso de MPI se emplearon también las variantes síncronas MPI_Send y MPI_Recv por ser más eficientes en este caso, ya que sólo se enviaba un mensaje por tamaño y no se podía aprovechar el posible solapamiento en las transferencias existentes en las órdenes asíncronas MPI_Isend/MPI_Irecv. Los resultados se obtuvieron realizando la media de cinco ensayos por cada variante del programa y tamaño del mensaje utilizado, para reducir todo lo posible las posibles interferencias producidas por la propia actividad del sistema operativo u otros tráficos existentes en la red. Los tiempos se muestran en la Figura 1. Queremos hacer notar que para tamaños de mensaje pequeños, hasta 512 bytes, el rendimiento (expresado en MB/s) es sensiblemente mejor para UPC. Sin embargo, conforme el tamaño del mensaje crece, el rendimiento para MPI mejora drásticamente superando claramente a UPC. Esto parece indicar que UPC podría emplearse mejor para la resolución de problemas de grano fino. No obstante, se optó por realizar una segunda prueba utilizando en esta ocasión las funciones asíncronas de comunicación punto a punto que ofrece la implementación de

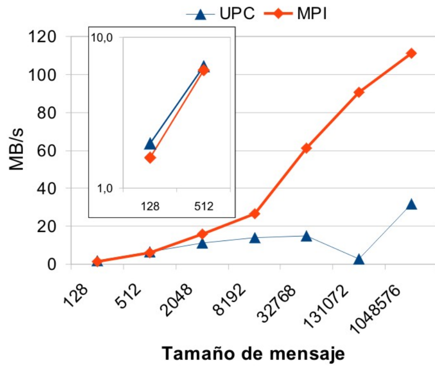

Figura 1: Ping-pong en modo síncrono

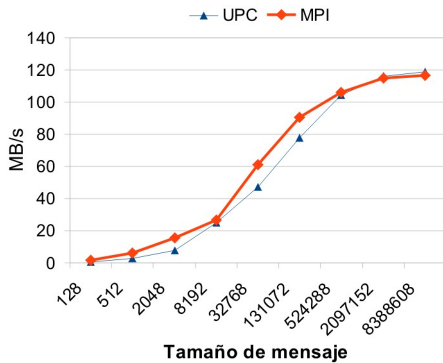

Figura 2: Ping-pong bidireccional y asíncrono

Berkeley. Además, en esta ocasión las comunicaciones se establecen bidireccionalmente, es decir, al estilo MPI, donde participan un emisor y en el destino espera la llegada de datos un receptor. Para esta segunda prueba, los resultados fueron más similares entre UPC y MPI, como se muestra en la Figura 2. Concretamente, para tamaños pequeños, MPI es ligeramente más rápido, aumentando esa diferencia en el segmento de tamaños de mensaje intermedios. Pero conforme el tamaño crece por encima de los 512KB se igualan en rendimiento, llegando incluso a ser superado por UPC a partir de 1 MB.

Conf.	n	N. proc.	Proc. por nodo
1	3000	9	1
2	3000	9	3
3	5000	16	4
4	5000	25	4

Cuadro 1: Configuraciones para el algoritmo de multiplicación de matrices por bloques

3.2. Multiplicación de matrices por bloques

En esta experimentación se implementó el algoritmo de Fox de multiplicación de matrices por bloques. Este algoritmo consiste en dividir las matrices A y B del producto, en $r \times r$ bloques de idéntico tamaño, distribuyendo cada uno de los bloques en un procesador. Se define por tanto una topología de procesadores en malla, donde cada uno de ellos contiene el bloque correspondiente de A y B [4]. Se definen cuatro configuraciones en función del orden de las matrices a multiplicar, el número de procesos, y su distribución (véase Cuadro 1).

Los resultados obtenidos se muestran en la Figura 3. Los tiempos corresponden a la media de cinco ensayos realizados en cada caso. En líneas generales se obtienen mejores tiempos de ejecución con UPC que con MPI salvo en la configuración 4. Por otra parte, si comparamos los resultados obtenidos con UPC1 y UPC2, se observa que UPC aprovecha mejor las capacidades SMP de una máquina. Así, en la configuración dos, se consigue bajar un 20 % el tiempo de ejecución al distribuir tres threads por nodo en lugar de uno, manteniendo la suma total de nueve. Esto tiene sentido, ya que las comunicaciones entre los procesos dentro de un nodo son más eficientes al utilizar pthreads en lugar de tener que atravesar capas de red; por otro lado se evita la necesidad de crear copias en buffer de los datos y los problemas derivados de un acceso demasiado concurrente a la memoria, lo que provoca problemas de cuello de botella. Si nos fijamos en MPI, vemos que mantiene los mismos tiempos al ejecutar los nueve procesos en nueve nodos o en tres (MPI1 y MPI2). Esto parece indicar que MPI no explota suficientemente las capa-

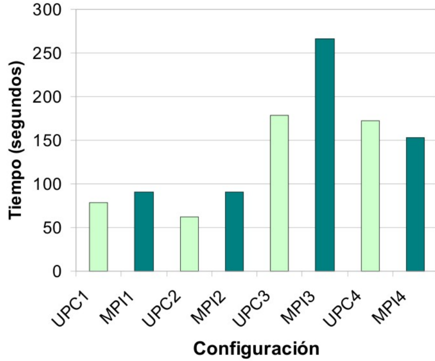

Figura 3: Multiplicación de matrices por bloques

cidades multinúcleo de los nuevos procesadores. Respecto al indicador de productividad, el recuento de líneas de código nos muestra que UPC es más eficiente, al emplearse 93 líneas de código (LOC) frente a las 107 necesitadas en MPI.

3.3. Algoritmo Newton Jacobi por bloques

El siguiente método implementado ha sido el método de *Newton Jacobi por bloques* para la resolución de sistemas no lineales [5]. Sea $F : \Re^n \rightarrow \Re^n$ una aplicación no lineal y consideremos el sistema de ecuaciones no lineales $F(x) = 0$. Supongamos que este sistema tiene una solución $x^\star$. Dado un vector inicial $x^{(0)}$, para resolver dicho sistema, el método de Newton construye una sucesión de iterados de la forma

$$x^{(l+1)} = x^{(l)} - x^{(l+\frac{1}{2})},\ l = 0, 1, 2, \ldots,$$

donde $x^{(l+\frac{1}{2})}$ es la solución del sistema lineal

$$F'(x^{(l)})z = F(x^{(l)}) \qquad (1)$$

y $F'(x)$ la matriz jacobiana.

Para la paralelización del método de Newton se ha utilizado, en la fase de resolución del sistema lineal (1), el método de Jacobi por bloques con factorizaciones LU. Es decir, para cada l se ha utilizado una partición del tipo $F'(x^{(l)}) = M - N$, siendo M es una matriz diagonal por bloques, cuyos bloques diagonales

Conf.	N. proc.	N. nodos	Proc./nodo
1	4	4	1
2	8	4	2
3	8	8	1
4	16	8	2

Cuadro 2: Configuraciones para el algoritmo Newton Jacobi por bloques

coinciden con los de $F'(x^{(l)})$. De esta forma el sistema lineal se resuelve mediante el siguiente proceso iterativo: para $k = 0, 1, 2, \ldots$,

$$z^{(k+1)} = M^{-1}Nz^{(k)} + M^{-1}F'(x^{(l)}),$$

donde la inversa de M se obtiene mediante factorizaciones LU. Dicha factorización consiste en la obtención de una matriz triangular inferior L y una matriz triangular superior U tal que $M = LU$.

El sistema no lineal que se ha utilizado para la evaluación de los algoritmos ha sido $Ax + \Phi(x) = b$, donde $A = tridiag[-1, 8, -1]$, $b = (3, 3, ..., 3)^T$ y $\Phi_i(x_i) = x_i e^{x_i}$, $1 \leq i \leq n$. Como aproximación inicial se ha tomado $x^{(0)} = (0{,}5, 0{,}5, ..., 0{,}5)^T$ y el criterio de parada $\sum_{i=1}^{n} |x_i^{(l)} - x_i^{(l-1)}| < 10^{-7}$.

Se desarrollaron los algoritmos tanto en UPC como en MPI, ejecutándose sobre el cluster Picolo para poder explotar las posibilidades de escalabilidad de cada lenguaje. Se definen cuatro configuraciones, tal y como muestra el Cuadro 2. Cada configuración se probó con diferentes niveles de carga, en este caso variando el orden del sistema, obteniendo resultados similares en todos los casos. La Figura 4 muestra los tiempos obtenidos para una matriz de tamaño 10000 (media de cinco ensayos), en la que se puede apreciar claramente el mejor comportamiento de UPC.

Con el fin de comprobar las capacidades de escalado y explotación de las posibilidades SMP en UPC, se definen cuatro configuraciones más, como muestra el Cuadro 3.

Las configuraciones 5, 6 y 7 se definen con el mismo número de procesos, aunque cambiando la disposición de los mismos, empleando más nodos o más threads dentro de un mismo nodo. Se realizaron diez ensayos por cada configu-

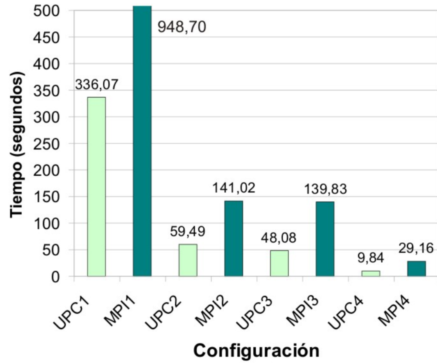

Figura 4: Newton Jacobi por bloques, $n = 10000$

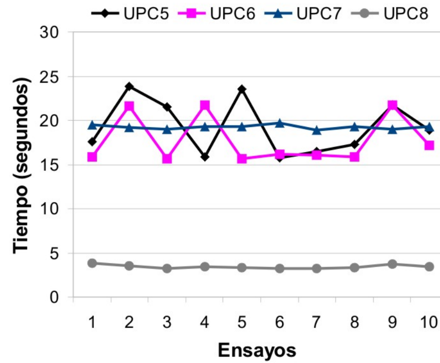

Figura 5: Escalado UPC con Newton Jacobi por bloques, n=10000

Conf.	N. proc.	N. nodos	Proc./nodo
5	12	3	4
6	12	4	3
7	12	6	2
8	63	21	3

Cuadro 3: Configuraciones para prueba de escalado

ración, obteniéndose la media para comparar tiempos. A continuación se muestra la Figura 5 con los diez ensayos realizados y sus tiempos. Una primera observación que se puede realizar es que en las configuraciones 5 y 6 la dispersión de resultados es alta, mientras que en las configuraciones 7 y 8 se reduce sensiblemente, tendiendo los ensayos a ser similares. Esto se debe a la granularidad de las comunicaciones de este algoritmo, que provoca cuellos de botella en el acceso a recursos como la memoria RAM y sufre penalizaciones aleatorias por la propia actividad de los procesos del sistema. Así, para configuraciones que utilizan tres o cuatro procesos por nodo se reproduce este comportamiento, mientras que al reducir el número de procesos a dos por nodo (UPC7) o reducir la cantidad de datos manejada por cada proceso (UPC8) la dispersión decrece.

Queremos hacer notar que a pesar de que comparando las medias de las configuraciones 5, 6 y 7 el mejor tiempo se atribuye a UPC6, analizando los diez ensayos se puede constatar que en UPC5 y UPC6 los tiempos mejoran sustancialmente en un entorno más estable respecto a UPC7. La razón estriba en una mejora del rendimiento distribuyendo 3 o 4 threads por procesador, frente a 2 threads. Si revisamos los tiempos de UPC8, mejoran como era de esperar al distribuirse la carga de cálculo entre los 63 procesadores. Analizando la escalabilidad del programa obtenemos que ésta es alta, dado que si comparamos UPC8 y UPC6 (ambos con 3 threads por nodo) obtenemos: $\frac{\text{N. Proc UPC8}}{\text{N. Proc UPC6}} = \frac{63}{12} = 5,25$ y $\frac{\text{Tiempo UPC6}}{\text{Tiempo UPC8}} = \frac{17,747}{3,443} = 5,154\ s$. Es decir, que se mantiene el factor de escalado también para los tiempos. Por otra parte, atendiendo al indicador de productividad, mientras que en UPC se han empleado 319 LOC, en la versión MPI se dedicaron 394, lo que produce un código un 20 % más reducido.

3.4. Algoritmo del gradiente conjugado no lineal

En este apartado se analiza la implementación paralela del método del gradiente conjugado no lineal (NLCG) [3]. El problema modelo utilizado para realizar los experimentos numéricos, conocido como el problema de Bratu, modela un proceso de reacción térmica en un material rígido (dominio 3D), en el que el proceso depende de un equilibrio entre el calor gene-

rado químicamente y la transferencia de calor por conducción [1]. En concreto tenemos que:

$$\nabla^2 u - \lambda e^u = 0, \qquad (2)$$

donde ∇^2 es el operador laplaciano, u es la temperatura y λ es una constante conocida como el parámetro de Frank-Kamenetskii. En los experimentos hemos considerado un dominio cúbico 3D de longitud uno y $\lambda = 6$. La discretización de la ecuación (2) produce el sistema no lineal $Ax + \Phi(x) = b$, donde $\Phi : \Re^n \rightarrow \Re^n$ es una función no lineal diagonal (es decir, la componente i–ésima Φ_i de Φ es una función que sólo depende de x_i). La matriz jacobiana de $F(x) = Ax + \Phi(x) - b$, es decir $F'(x) = A + \Phi'(x)$, es una matriz dispersa de orden d^3. Para detallar el comportamiento del algoritmo se presentan resultados para sistemas de tamaños $n = 8000$ y $n = 216000$, que corresponden a los mallados $d = 20$ y $d = 60$, respectivamente. Las matrices se han almacenado usando la técnica de almacenamiento CSR (Compressed Sparse Row) y se han utilizado las librerías BLAS para la operaciones entre vectores y matrices.

Los siguientes resultados pretenden analizar el rendimiento en el cluster Picolo, que emplea una red Gigabit Ethernet para comunicar entre nodos. Por tanto los procesos realizan comunicaciones inter-nodos, produciéndose las transferencias de datos a través de la red, descartando cargar más de un proceso por nodo. Tal y como se ilustra en las Figuras 6 y 7, en general, MPI supera en rendimiento a UPC en todas las configuraciones definidas (véase Cuadro 4). La excepción se produce para $d = 20$ y seis procesos, donde debido a la naturaleza del algoritmo no se soporta bien el escalado de procesadores por encima de cuatro, y en este caso UPC iguala los tiempos.

Sin embargo cuando se trabaja con el equipo Bi-quad donde las configuraciones elegidas, tal y como muestra el Cuadro 4, se basan en el despliegue en SMP, de forma que la comunicación se realiza intra-nodo, en memoria, el rendimiento de UPC para este algoritmo es superior al de MPI, como se ilustra en la Figura 8. Las posibles causas de los tiempos tan altos que se reproducen en UPC para el cluster Picolo, en contraste a los obtenidos en la configuración SMP del Bi-quad serían por una parte la naturaleza del algoritmo que está orientado a arquitecturas multicore en SMP, con un escalado pequeño (2-4 procesos) y la red de comunicaciones, Ethernet, que no aprovecha el potencial de rendimiento del entorno de ejecución GASNET. En UPC las transferencias de datos son unilaterales, y esta característica se explota mejor en redes de alto rendimiento que poseen la capacidad RDMA (Remote Direct Memory Access) como por ejemplo Infiniband. Respecto a la productividad, en este caso podemos observar una vez más, menor número de líneas de código para el caso de UPC; en este caso se incrementa la productividad al tratarse de un programa de mayor envergadura. Así, tenemos en UPC 670 LOC, mientras que en MPI se tienen 1189 LOC. Se puede aseverar que UPC supera en este aspecto con creces a la versión en MPI.

Conf.	1	2	3	11	12	13
Proc.	2	4	6	2	4	6
d	20	20	20	60	60	60

Cuadro 4: Configuraciones para NLCG

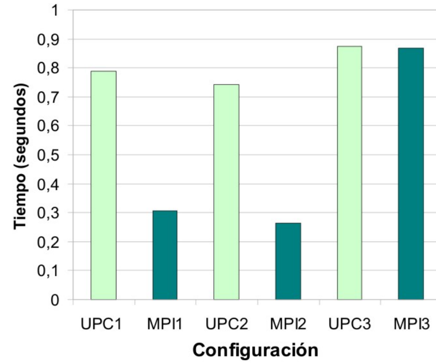

Figura 6: NLCG , $d = 20$, Picolo

4. Conclusiones

En este trabajo se ha realizado una serie de experimentaciones para analizar el comporta-

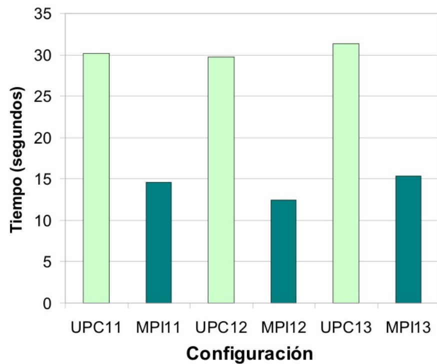

Figura 7: NLCG, $d = 60$, Picolo

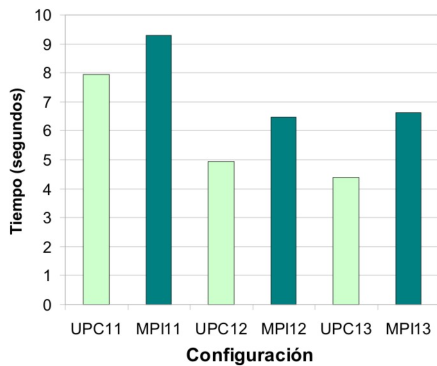

Figura 8: NLCG, $d = 60$, Bi-quad

miento de UPC en distintos entornos paralelos atendiendo a su rendimiento, capacidad de escalabilidad y productividad del código. En resumen podemos decir que la capacidad de escalabilidad de UPC es muy similar a MPI, sin embargo en relación a la productividad del código UPC siempre se muestra mejor necesitando menos líneas de código para implementar el correspondiente algoritmo. Por otra parte, en relación al rendimiento atendiendo al tipo de computador utilizado, se observa que en máquinas SMP se obtienen en general mejores tiempos de ejecución con UPC que con MPI, mientras que en clusters donde se utiliza un proceso por nodo y se tienen comunicaciones basadas en una red Ethernet, en algunos casos los resultados son similares con UPC y MPI y en otros se obtienen prestaciones sensiblemente peores que en MPI, aunque aceptables. Esto nos permite concluir que UPC puede ser una buena alternativa a MPI como lenguaje de implementación de programas paralelos. Su versatilidad para adaptarse a paradigmas de memoria compartida o distribuida, siendo en realidad un híbrido de ellos, lo hace especialmente interesante.

Agradecimientos

El presente trabajo ha sido parcialmente financiado por el Ministerio de Ciencia e Innovación mediante el proyecto TIN2008-06570-C04-04, y por la Universidad de Alicante mediante el proyecto VIGROB-020.

Referencias

[1] Averick, B.M., Carter, R.G., More, J.J., Xue, G., *The MINPACK-2 test problem collection*, Tech. Rep. MCS-P153- 0692, Mathematics and Computer Science Division, Argonne National Laboratory, 1992.

[2] El-Ghazawi, T., Carlson, B., Sterling, T., Yelick, K., *Distributed Shared Memory Programming*, John Wiley & Sons, 2005.

[3] Fletcher, R., Reeves, C., *Function Minimization by Conjugate Gradients*, The Computer Journal, 7:149–154, 1964.

[4] Ortega, J.M., *Introduction to Parallel and Vector Solution of Linear Systems*, Plenum Press, New York, 1988.

[5] Rheinboldt, W.C., *Methods for Solving Systems of Nonlinear Equations*, SIAM, Philadelphia, Pennsylvania, 1974.

[6] *UPC Language Features*, Lawrence Berkeley National Laboratory, http://upc.lbl.gov/lang-overview.shtml.