Implementaciones paralelas para un problema de control de inventarios de productos perecederos

Alejandro G. Alcoba, Eligius M.T. Hendrix, Inmaculada García[1], Gloria Ortega [2]

Resumen—**En este trabajo se analizan y evalúan dos implementaciones de un algoritmo de optimización para un problema de control de inventarios de productos perecederos. Las implementaciones se han llevado a cabo utilizando una arquitectura heterogénea donde cada nodo está compuesto por varios multicores y varias GPUs. Las versiones paralelas que se han desarrollado son: (1) una versión MPI-PTHREADS en la que se extrae el paralelismo tanto a nivel de proceso MPI como a nivel de hilo y (2) una versión multiGPU en la que se obtiene el paralelismo a nivel de proceso MPI y a nivel de cores de GPU. Este algoritmo puede ser descompuesto fácilmente en un conjunto de tareas que no presentan ninguna dependencia entre sí. Sin embargo, la carga computacional asociada a cada una de las tareas es diferente y el problema del reparto de las tareas entre los elementos de proceso se puede modelar como un problema de Bin Packing. Ello implica que la selección del conjunto de tareas asociadas a cada una de las unidades de computación requiere del diseño de heurísticas que sean capaces de balancear la carga eficientemente y de forma estática. En este trabajo hemos analizado y evaluado varias heurísticas. Finalmente, la mejor heurística ha sido la utilizada en la implementación paralela del algoritmo de control de inventarios que ha sido evaluado en la versión MPI-PTHREADS y en la versión multiGPU. Para la implementación MPI-PTHREADS los resultados obtenidos muestran una buena escalabilidad mientras que las versión MultiGPU para el ejemplo que se ha evaluado deja de ser eficiente cuando se usan mas de 2 GPUs.**

Palabras clave—**Multihilo, Multi-GPU, Bin Packing, Monte-Carlo, inventarios, productos perecederos.**

I. Introducción

El objetivo de este trabajo consiste en determinar hasta que punto el uso de una arquitectura heterogénea (multicore-multiGPU) facilita la resolución de un problema de optimización del control de inventarios de productos perecederos. Pretendemos aprovechar la capacidad computacional de estas arquitecturas para obtener soluciones mas exactas, para ejemplos mas pesados desde el punto de vista de la computación, manteniendo tiempos de respuesta aceptables. El problema del control de inventarios queda definido a lo largo de una serie finita de T periodos de tiempo en los que se ha de satisfacer la demanda (estocástica) de un determinado producto perecedero que desde que se produce tiene una vida útil de J periodos. En el modelado de este problema se supone que la distribución se realiza siguiendo la política de distribución FIFO, entregando el producto demandado con mayor antigüedad. Se supone, además, que la demanda que no se satisfaga en un periodo queda perdida, no pudiéndose acumular al periodo siguiente. La solución a este problema consiste en encontrar que cantidades de pedido a lo largo de todos los periodos resulta óptima, en el sentido de minimizar el coste asociado a la producción, distribución, almacenamiento y desecho de los productos que sobrepasen su vida útil.

Actualmente, las arquitecturas de computación de altas prestaciones más extendidas son las plataformas heterogéneas basadas en sistemas de memoria distribuida, donde cada nodo tiene una arquitectura multicore que podría albergar un número distinto de cores [1]. Por lo tanto, las implementaciones paralelas tienen que ser adaptadas para poder ser ejecutadas en dichas arquitecturas heterogéneas. En este contexto, es necesario tener un conocimiento detallado tanto del algoritmo a paralelizar como de los recursos computacionales que se van a utilizar para la implementación [2]. Además, a estas arquitecturas se les pueden incorporar aceleradores, como son FPGAS, GPUs, coprocesadores Intel Xeon Phi, etc. En concreto, en el problema del control de inventario para productos perecederos se ha optado por la combinación de clústeres de Multi-GPUs. De este modo, el uso de plataformas masivamente paralelas (GPUs) permite la aceleración de las tareas computacionalmente más costosas, porque estas unidades tienen mucha potencia de cálculo para los esquemas de computación vectorial. De forma adicional, el uso de plataformas de memoria distribuida permite obtener unos resultados más precisos debido a que el uso de computación paralela permite incrementar el número de simulaciones realizadas para resolver un caso particular sin que el tiempo de ejecución se incremente.

El modelo de computación paralela asociado a este problema se puede describir en términos de un conjunto de conjunto de tareas que no presentan dependencias entre sí. Sin embargo, la carga computacional de cada una de estas tareas es variable y por lo tanto pueden aparecer problemas de desbalanceo de la carga si se hace un reparto de la carga a ciegas. Este problema de asignación de tareas a elementos de procesamiento se conoce en la literatura de complejidad como problema de Bin packing [3]. Dado que es un problema NP-Completo, se han desarrollado varias heurísticas que permiten tener una solución en un tiempo razonable.

El resto del trabajo se organiza de este modo. La

[1]Computer Architecture, Universidad de Málaga. Campus de Excelencia Internacional Andalucía Tech, e-mail: {agutierreza,eligius,igarciaf}@uma.es

[2]Informatics, Univ. of Almería, Agrifood Campus of Int. Excell., ceiA3, e-mail: gloriaortega@ual.es

Sección II estudia y analiza el modelo propuesto para el control de inventario de productos perecederos. La Sección III describe el algoritmo secuencial implementado. En la Sección IV se explican los detalles de las versiones paralelas implementadas. Además, se describe el problema de Bin packing que hay que resolver inicialmente para repartir la carga de trabajo entre las diferentes plataformas. La Sección V ofrece algunos resultados experimentales computacionales al evaluar el modelo utilizando un clúster de Multi-GPUs. Finalmente, la Sección VI expone las conclusiones y las principales líneas de actuación futuras.

II. Descripción del modelo

La base de las implementaciones que se presentan en este trabajo es un algoritmo desarrollado en Matlab para resolver un problema MINLP (Mixed Integer NonLinear Programming). Se trata de planificar, a lo largo de un número finito de periodos T, las cantidades que se deben proveer de cierto producto perecedero para satisfacer la demanda bajo una restricción que establece un nivel de servicio β que necesariamente se debe satisfacer. En concreto, esta restricción establece que para cada periodo (siempre hablando en términos de esperanza matemática) a lo sumo una fracción β de la demanda no pueda ser satisfecha y sea perdida por falta de stock, ya que se supone que esta no puede ser servida en un periodo posterior. Esta condición es equivalente a que al menos una fracción $(1-\beta)$ de la demanda sea cubierta en cada periodo. La duración de cada item producido desde que está disponible para el consumidor hasta que ha de ser retirado es de $J < T$ periodos. Además, se supone que los productos se distribuyen siguiendo la regla FIFO: los productos son expedidos comenzando por los más antiguos.

El problema de optimización que se plantea es el de encontrar la cantidad de producto perecedero que hay que producir en cada periodo de forma que se satisfgan todas las restricciones del problema y que además se minimice una función coste.

A continuación se detallan las principales variables del modelo:

Indices

- t índice del periodo, $t = 1, \dots, T$, siendo T el número total de periodos
- j índice de edad, $j = 1, \dots, J$, siendo J la vida útil de cada unidad

Data

- $\boldsymbol{d}_t$ Demanda en cada periodo con distribución normal dada por su media $\mu_t > 0$ y varianza $(cv \times \mu_t)^2$ dado por un coeficiente de variación cv, idéntico en cada periodo.
- k Coste por periodo en el que se decide realizar un pedido, $k > 0$
- c Coste unitario de producto, $c > 0$
- h Coste por almacenamiento, $h > 0$
- w Coste unitario de desecho, puede ser negativo con la condición, $w > -c$
- β Nivel de servicio, $0 < \beta < 1$

Variables

- $Q_t \geq 0$ Cantidad de producto producido y disponible en el periodo t. Denotamos por Q al vector completo $(Q_1, \dots, Q_T)$
- $Y_t \in \{0, 1\}$ Indica si se produce un pedido en el periodo t. Es 1 si y solo si $Q_t > 0$. Denotamos por Y al vector completo $(Y_1, \dots, Y_T)$
- $\boldsymbol{X}_t$ Ventas perdidas en el periodo t
- $\boldsymbol{I}_{jt}$ Inventario de edad j al final del periodo t, considerando un periodo inicial fijo, $I_{j0} = 0$, $\boldsymbol{I}_{jt} \geq 0$ para $j = 1, \dots, J$.

Además, se usará la notación $(\cdot)^+ = max(\cdot, 0)$.

La función coste que se pretende minimizar depende del vector $Q = (Q_1, \dots, Q_T)$ y se puede definir como:

$$f(Q) = \sum_{t=1}^{T} \left(C(Q_t) + E\left(h \sum_{j=1}^{J-1} \boldsymbol{I}_{jt} + w\boldsymbol{I}_{Jt} \right) \right), \quad (1)$$

siendo

$$C(x) = k + cx, \ \text{ if } \ x > 0, \ \text{and } \ C(0) = 0. \quad (2)$$

El nivel de inventario para cada periodo $t = 1, \dots, T$ y cada edad j siguiendo la regla FIFO puede calcularse como sigue:

$$\boldsymbol{I}_{jt} = \begin{cases} \left(Q_t - (\boldsymbol{d}_t - \sum_{j=1}^{J-1} \boldsymbol{I}_{j,t-1})^+\right)^+ & j = 1, \\ (\boldsymbol{I}_{J-1,t-1} - \boldsymbol{d}_t)^+ & j = J, \\ \left(\boldsymbol{I}_{j-1,t-1} - (\boldsymbol{d}_t - \sum_{i=j}^{J-1} \boldsymbol{I}_{i,t-1})^+\right)^+ & otro\ j \end{cases} \quad (3)$$

Por otra parte, la restricción del nivel de servicio puede expresarse como:

$$E(\boldsymbol{X}_t) \leq (1-\beta)\mu_t, \ t = 1, \dots, T. \quad (4)$$

Para controlar el cumplimiento de esta restricción es necesario calcular las ventas perdidas que se producen en cada periodo t, lo cual viene dado por:

$$\boldsymbol{X}_t = \left(\boldsymbol{d}_t - \sum_{j=1}^{J-1} \boldsymbol{I}_{j,t-1} - Q_t \right)^+. \quad (5)$$

El valor esperado de las ventas perdidas es una función conocida como *loss-function* que en general no admite una expresión en términos elementales. Algunas aproximaciones factibles pueden verse en [4], [5], [6], [7]. Para nuestro modelo hemos decidido utilizar la simulación Monte-Carlo para obtener una estimación de la *loss-function*. Con las condiciones impuestas, el problema de encontrar las cantidades de producto perecedero que se deben producir en cada periodo y que minimizan la función coste $f(Q)$ dada en (1), y con ello la política $Y \in \{0, 1\}^T$ de periodos de pedido óptima, es un problema MINLP (Mixed Integer NonLinear Programming). Su resolución, mediante técnicas de programación dinámica, puede consultarse en [8]. Como veremos, la técnica usada presenta características adecuadas para su implementación en computadores de alto rendimiento.

III. Algoritmo secuencial

En esta sección se detalla el algoritmo que resuelve el problema MINLP mediante programación dinámica, tal como se ha planteado en la sección anterior. El Algoritmo 1 presenta el nivel más general de la resolución del problema. En primer lugar, se generan en forma de vectores todas las políticas de pedido válidas $\{0,1\}^T$ para unos valores T y J dados. Una política de pedido $Y \in \{0,1\}^T$ es considerada no válida si contiene J o más ceros consecutivos, ya que se considera que en ningún periodo la demanda puede ser determinista e igual a 0 y, por lo tanto, no se cumplirían los requerimientos de nivel de servicio. Para una determinada política de pedidos Y, el Algoritmo 2 calcula, para cada periodo, cuales son las cantidades de producto a producir ($Q(Y)$) que minimizan la función coste (1). El Algoritmo 2 solo se ejecuta para los casos en los que la función $LB(Y) <$ mincost, siendo $LB(\cdot)$ un limite inferior de la estimación de la función coste (1) (en [9] puede encontrarse información sobre la estimación de $LB(.)$). Por lo tanto, el Algoritmo 1 realiza una evaluación exhaustiva (salvando una cantidad relativa de casos con la función LB), para encontrar tanto la política óptima Y^* como el vector de producción óptimo Q^* y su coste asociado $f(Q^*)$.

Algorithm 1 *AllY*(): Cálculo de la política de pedidos óptima (Y^*) a partir de la evaluación de todas las politicas factibles Y

```
1:  Generate all feasible Y
2:  mincost=∞
3:  for all Y do
4:      if LB(Y) < mincost then
5:          Q_Y=MINQ(Y); # Algorithm 2
6:          Determine f(Q_Y)
7:          if f(Q_Y) < mincost then
8:              mincost=f(Q_Y)
9:              Y* = Y; Q* = Q_Y
10:         end if
11:     end if
12: end for
13: return Y*; Q*; f(Q*) = mincost
```

Algorithm 2 $MinQ(Y)$: Optimización de $Q(Y)$

```
1:  Generar para Y los vectores A y R
2:  for i = 1 to lenght(A) do
3:      if A_i = 1 o A_i − A_{i−1} = J then # No Invent.
4:          Q_i = Q_{R_i,A_i};
5:          floss(Q_i, A_i, A_i + R_i − 1) #Alg 4
6:      else
7:          Q_i = OrdVal(A_i, A_i + R_i) # Alg 3
8:      end if
9:  end for
10: return Q;
```

Para cada periodo t en el que $Y_t = 1$, el Algoritmo 2 calcula la cantidad de producto producido que permita cubrir la demanda de un ciclo, entendiendo por ciclo los periodos comprendidos entre dos elementos consecutivos del vector Y que valen 1. El vector A almacena el índice del inicio de cada ciclo, mientras que R guarda su duración en periodos. Para cada ciclo, la cantidad de producto necesaria es aquella cantidad que ajusta el valor de las ventas perdidas al límite inferior permitido por (4).

La forma de calcular $Q_i(Y)$ para los ciclos en los que el inventario es nulo es diferente de la usada para los ciclos en los que existe algún inventario.

En caso de no existir inventario anterior el valor esperado de las ventas perdidas para este problema concreto puede ser calculado resolviendo una sencilla ecuación [8].

Para los ciclos en los que existe algún inventario es necesario recurrir a la simulación reiterada del problema, haciendo uso de (3) y (5) para obtener aproximaciones del valor de ventas perdidas que permitan encontrar el valor óptimo de $Q(Y)$. De eso se encarga la función $OrdVal(t_1, t_2)$, descrita en el Algoritmo 3, que, por el método de la secante, calcula la cantidad óptima para el ciclo. La función *floss*(q, t_1, t_2) (Algoritmo 4 realiza la simulación y devuelve la cantidad de ventas perdidas en el último periodo del ciclo, t_2, suponiendo una llegada de q unidades al comienzo del ciclo (periodo t_1).

Algorithm 3 $OrdVal(t_1, t_2)$: Calculo de la cantidad de pedido mínima que satisface (4) para un ciclo entre t_1 hasta t_2

1: $K = (1-\beta) * \mu_{t_2}$;
2: $q_1 = \sum_{i=t_1}^{t_2} \mu_i$;
3: $q_2 = q_1(1 + cv\hat{x})$;
4: $f_1 \leftarrow floss(q_1, t_1, t_2)$; #Alg 4
5: $f_2 \leftarrow floss(q_2, t_1, t_2)$; #Alg 4
6: **repeat**
7: $\quad q \leftarrow q_1 + \frac{(K-f_1)(q_2-q_1)}{f_2-f_1}$; #secant method
8: $\quad f_1 \leftarrow f_2$;
9: $\quad f_2 \leftarrow floss(q, t_1, t_2)$; #Alg 4
10: $\quad q_1 \leftarrow q_2$;
11: $\quad q_2 \leftarrow q$;
12: **until** $|\frac{f_2}{\mu_{t_2}}| - (1-\beta) < \epsilon$
13: **return** q

Algorithm 4 $floss(q, t_1, t_2)$: Método Monte-Carlo para aproximar $E(\boldsymbol{X})$

1: **for** $n = 1$ **to** N **do**
2: $\quad$ **for** $t = t_1$ **to** t_2 **do**
3: $\quad\quad$ **for** $j = 1$ **to** $j = J$ **do**
4: $\quad\quad\quad$ Update $I_{j,t,n}$ using (3)
5: $\quad\quad$ **end for**
6: $\quad$ **end for**
7: **end for**
8: $X = \frac{1}{N}\sum_{n=1}^{N}\left(d_{t_2,n} - \sum_{j=1}^{J-1} I_{j,t_2-1,n} - q\right)^+$;
9: **return** X;

IV. Implementaciones paralelas

El orden de complejidad del problema, partiendo del Algoritmo 1, está relacionado con el número de políticas factibles Y, lo cual depende de los valores de J y T. Independientemente del valor de J, el número de casos posibles a tratar aumenta de forma exponencial con el valor de T, es decir, $O(e^T)$. Más aún, para cada Y tratado por el Algoritmo 2, su complejidad depende del número de veces que es necesario utilizar el Algoritmo 3, limitada a T en cada caso. Por último, cada ejecución del Algoritmo 3 requiere el cómputo de la función *floss* que a su vez ejecuta N simulaciones de Monte-Carlo en las que se tiene que calcular el inventario y las ventas perdidas mediante (3) y (5). Por tanto, el orden de complejidad para el método completo, es decir, hallar el vector de pedidos Y óptimo y las cantidades óptimas de pedido, es aproximadamente del orden de $O(N \cdot T \cdot e^T)$.

En la sección anterior se ha puesto de manifiesto la necesidad de realizar simulaciones para obtener aproximaciones de la función *floss*. Esta función es la que consume la mayor parte del tiempo computacional de la ejecución del problema de inventarios. Es importante destacar la necesidad de realizar un elevado número de simulaciones para que las aproximaciones que lleva a cabo la función *floss* sean suficientemente exactas. Como ejemplo, en la Figura 1 se muestra el histograma de 20000 ejecuciones de la función *floss*, realizando en cada una de ellas $N = 30000$ simulaciones de Monte-Carlo, para un caso en el que analíticamente, su valor exacto es 0.05. Las aproximaciones de la función *floss* siguen una distribución normal (pasa un test chi-cuadrado (χ^2) al 5% de nivel de significación) y, aunque la media se ajusta a la realidad, 0.05, la dispersión es alta. Por tanto, para realizar aproximaciones relativamente precisas del valor de las ventas perdidas (X), es necesario realizar un número N de simulaciones del problema suficientemente alto, lo que constituye la verdadera carga computacional del problema. Como caso de estudio, se ha considerado un ejemplo del problema de control de inventarios en el que $T = 12$ y $J = 3$, que es bastante realista. Consideraremos siempre $N = 50000$ simulaciones de Monte-Carlo.

Revisando el Algoritmo 2 se observa que los ciclos en los que existe inventario previo son aquellos en los que hay que proceder a la simulación de Monte-Carlo de forma reiterada mediante el Algoritmo 3 (función $OrdVal$), que es lo que representa la mayor carga computacional de cada optimización de un vector Y. Podemos hacer una estimación de la carga computacional asociada con cada vector de pedidos (Y) (Algoritmo 2) como la suma de los valores de los elementos del vector R para los casos en los que se ejecuta el Algoritmo 3 (función $OrdVal$). Para el caso que vamos a analizar, con $T = 12$ y $J = 3$, el número total de vectores Y (diferentes politicas de pedidos) que se generan es 927, y la carga computacional estimada asociada a cada vector Y valorada entre 0 y 11 presenta una distribución como la mostrada en

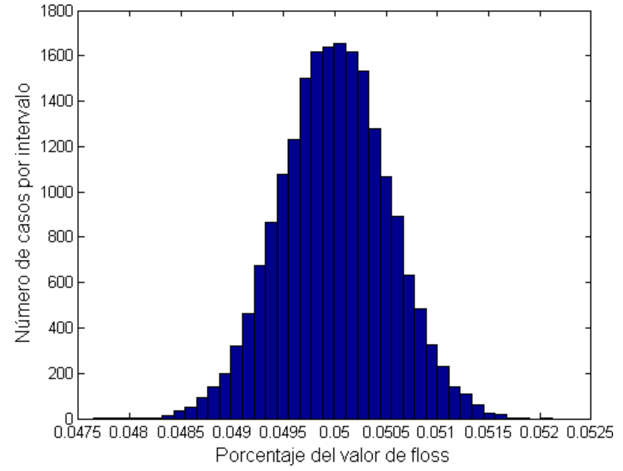


Fig. 1
Histograma de 20000 aproximaciones de *floss* mediante el método Monte-Carlo con $N = 30000$ simulaciones en cada caso. El valor exacto, calculado analíticamente para ese caso, es 0.05

la Tabla I, donde, por ejemplo, existen 24 configuraciones diferentes del vector Y que tendrían una carga estimada de 5 unidades. Experimentalmente, se ha podido comprobar que esta estimación de la carga para cada vector Y está directamente relacionada con el tiempo de ejecución del Algoritmo 2.

Una vez analizada la estructura algorítmica del problema, pasamos a describir los detalles de las implementaciones que hemos llevado a cabo sobre una arquitectura heterogénea formada por un clúster de Multi-GPUs (multicores y dispositivos GPUs). El hecho de explotar una plataforma heterogénea de un clúster tiene dos ventajas fundamentales: poder abordar la resolución de problemas de mayor tamaño y reducir el tiempo de ejecución de un caso concreto. Las implementaciones consideradas en este trabajo han sido:

- MPI-PTHREADS: Esta implementación obtiene el paralelismo de los procesadores multicore y de los nodos disponibles en el clúster. Para ello, se utiliza programación basada en hebras [10] y MPI [11].
- Multi-GPU: Esta implementación está basada en el uso de GPUs para realizar las simulaciones de Monte-Carlo, las cuales son la parte computacionalmente más costosa del problema a resolver. Para ello, la interfaz de programación que se utiliza es CUDA [12].

En las siguientes subsecciones se describen ambas implementaciones, así como el reparto inicial de la carga entre las diferentes unidades de proceso, en base a la carga estimada asociada a cada vector Y tal y como se ha descrito en la Tabla I, y que se puede modelar como un problema de Bin packing [3].

A. *Implementación MPI-PTHREADS*

Centrando nuestra atención en la implementación MPI-PTHREADS, se ha explotado el paralelismo en

TABLA I

NÚMERO DE VECTORES Y (CASOS POSIBLES) EN FUNCIÓN DE LA CARGA COMPUTACIONAL ESTIMADA, PARA $T = 12$ Y $J = 3$.

Carga estimada	0	1	2	3	4	5	6	7	8	9	10	11
Casos posibles	1	1	2	6	14	24	50	86	120	185	260	178

dos niveles: a nivel de nodo (memoria distribuida) y a nivel de multicore (memoria compartida). Por un lado, existen múltiples formas de paralelizar rutinas en modelos de memoria compartida, aunque la librería estándar es Pthreads (POSIX threads). Pthreads provee un conjunto unificado de rutinas en una librería de C cuyo principal objetivo es facilitar la implementación de threads o hilos en el programa. Por otro lado, debido a su portabilidad, MPI ha sido el interfaz considerado para explotar el paralelismo a nivel de nodo.

Partiendo del algoritmo de optimización del problema de inventarios, se ha realizado una paralelización híbrida (MPI y Pthreads), en la cual el conjunto de vectores Y que se van a evaluar en el problema de optimización son repartidos entre los procesadores de acuerdo a las heurísticas de balanceo de la carga que se describen en la Sección IV-B. La evaluación de esta implementación se ha realizado en un clúster Bullx y los resultados se describen en la Sección V.

B. Heurísticas para el problema de Bin packing

El reparto inicial de la carga de trabajo entre los procesadores disponibles puede considerarse como un problema de Bin packing con algunas restricciones. El problema de Bin packing se enmarca dentro de la optimización combinatoria (NP-completo), y en nuestro caso se puede modelar de la siguiente forma: Dado un conjunto de E ejecuciones independientes del Algoritmo 2 (items), cada una de ellas con una carga computacional $0 < w_i < B$ y dado un conjunto de P procesadores (Bins), repartir las ejecuciones del algoritmo entre los procesadores de forma que la carga computacional máxima asignada a un procesador sea mínima (ver [3] para una formulación general del problema de Bin packing).

Debido a la dificultad de encontrar soluciones óptimas para este tipo de problemas, habitualmente se utilizan técnicas heurísticas y metaheurísticas, que son capaces de encontrar una solución aceptable en un tiempo razonable. Algunas de estas heurísticas están inspiradas en computación evolutiva [13]. Para resolver el problema de balanceo de la carga que se ha modelado como un problema de tipo Bin packing, proponemos tres algoritmos heurísticos (H1, H2 y H3) para repartir la carga de trabajo (en nuestro caso, los posibles vectores Y) entre todos los elementos de procesamiento disponibles de forma que se minimice el tiempo de ejecución del problema de optimización.

1. (H1): Heurística basada en Round Robin: Ordenando previamente, de mayor a menor, el peso de las tareas a asignar, estas se reparten entre los P procesadores siguiendo el patrón $(1, \ldots, P, P, P-1, \ldots, 1, 1, \ldots)$
2. (H2): Heurística basada en asignar sucesivamente los items w_i al procesador que menos carga de trabajo haya acumulado.
3. (H3): Similar a la heurística H2, pero previamente ordenando los items de mayor a menor carga.

Para valorar las heurísticas se han utilizado tres instancias del problema denominadas γ_1, γ_2 y U, en las que la carga computacional estimada que se asocia a cada vector Y es diferente. (γ_1) y (γ_2) están basadas en distribuciones gamma con parámetros de forma y escala (10,4) y (1,25), respectivamente, y (U) sigue una distribución uniforme con valores entre 0 y 100. De cada una se han tomado 4000 muestras distintas, entendiendo que cada una de ellas es una tarea cuya carga computacional está asociada a su valor. En la Figura 2 aparecen representadas las muestras de las distribuciones gamma(10,4), gamma(1,25) y la uniforme.

Para medir el grado de balanceo de la carga asignada a cada procesador, se ha utilizado el coeficiente de Gini (G), ampliamente utilizado en el campo de la economía para medir el grado de desigualdad de la distribución de la riqueza en poblaciones [14], [15], [16]. Este índice varía entre 0 (equidad absoluta) y 1 (un solo individuo (procesador) posea toda la riqueza de la población (carga computacional)). G se define como la media de la diferencia entre cada posible par de procesadores, divididos por su carga media. Para un número de ejecuciones E asignadas a P elementos de proceso, siendo w_i la carga computacional asignada al procesador i, ordenadas de forma ascendente, G se calcula como sigue:

$$G = \frac{2 \sum_{i=1}^{P} i \cdot w_i}{P \sum_{i=1}^{P} w_i} - \frac{P+1}{P} \quad (6)$$

Gráficamente, G representa el ratio entre la diferencia del área rodeada por la línea de uniformidad y la curva de Lorenz de la distribución, y el área triangular que hay debajo de la línea de uniformidad. G toma valores entre un mínimo de 0, cuando todos los procesadores tienen la misma carga, a un máximo de 1, cuando todos los procesadores (excepto uno) tienen una carga de cero. Por lo tanto, cuando G se acerca a 0 la carga está bien balanceada, y cuando se acerca a 1 está desbalanceada.

En la Tabla II se resume el comportamiento de las diferentes heurísticas a través del valor del coeficiente de Gini (G) para los ejemplos planteados (comparándolos con un reparto a ciegas (columna

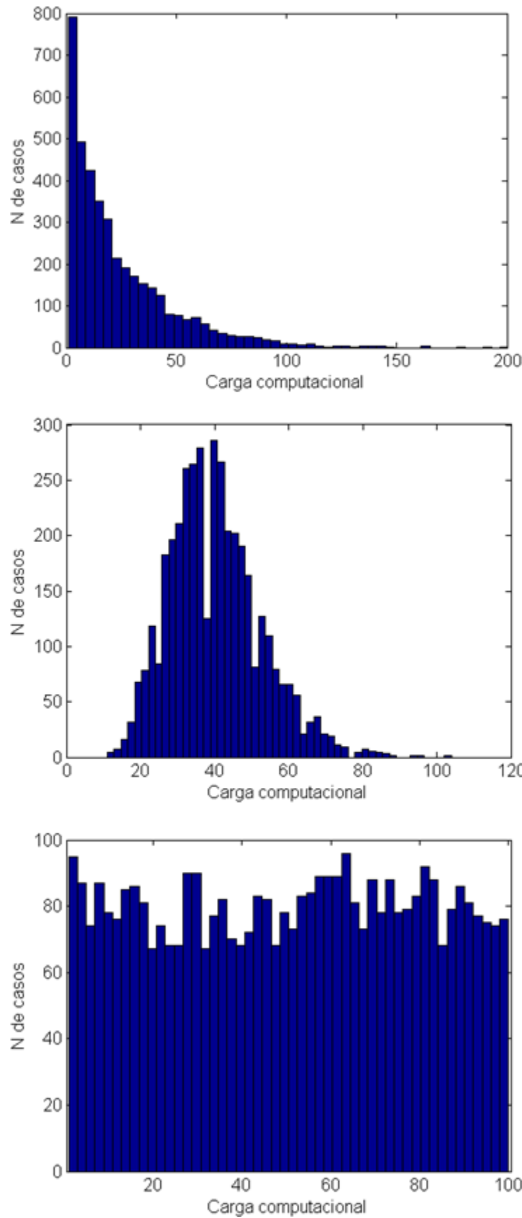


Fig. 2
Distribuciones γ_1: Gamma(10,4) (arriba), γ_2: Gamma(1,25) (centro) y U Uniforme[0,100] (abajo) de una muestra de 4000 casos.

TABLA II
Comportamiento de las heurísticas H1, H2 y H3 y su comparativa con un reparto aleatorio de tareas HR, para $P = 8, 16, 32, 64$ y las distribuciones γ_1, γ_2 y U. El comportamiento se mide por el coeficiente de Gini G.

γ_1

P	H1	H2	H3	HR
8	$1,3\ 10^{-4}$	$4,8\ 10^{-4}$	$1,2\ 10^{-5}$	$3,7\ 10^{-3}$
16	$2,5\ 10^{-4}$	$1,2\ 10^{-3}$	$5,2\ 10^{-5}$	$7,6\ 10^{-3}$
32	$5,8\ 10^{-4}$	$2,0\ 10^{-3}$	$1,4\ 10^{-4}$	$1,3\ 10^{-2}$
64	$3,2\ 10^{-3}$	$3,9\ 10^{-3}$	$1,5\ 10^{-3}$	$2,1\ 10^{-2}$

γ_2

P	H1	H2	H3	HR
8	$6,9\ 10^{-4}$	$1,0\ 10^{-3}$	$1,9\ 10^{-5}$	$2,3\ 10^{-2}$
16	$1,2\ 10^{-3}$	$2,1\ 10^{-3}$	$1,9\ 10^{-5}$	$3,9\ 10^{-2}$
32	$3,3\ 10^{-3}$	$4,1\ 10^{-3}$	$7,8\ 10^{-5}$	$6,1\ 10^{-2}$
64	$8,0\ 10^{-3}$	$7,6\ 10^{-3}$	$1,0\ 10^{-4}$	$7,9\ 10^{-2}$

U

P	H1	H2	H3	HR
8	$1,3\ 10^{-4}$	$7,5\ 10^{-4}$	$7,4\ 10^{-6}$	$1,2\ 10^{-2}$
16	$1,8\ 10^{-4}$	$1,4\ 10^{-3}$	$8,6\ 10^{-6}$	$1,8\ 10^{-2}$
32	$2,2\ 10^{-4}$	$2,4\ 10^{-3}$	$3,9\ 10^{-5}$	$2,6\ 10^{-2}$
64	$3,7\ 10^{-4}$	$4,1\ 10^{-3}$	$5,4\ 10^{-5}$	$4,0\ 10^{-2}$

(HR)). Claramente se demuestra en esta tabla que la heurística H3 es, al menos, un orden de magnitud mejor que las heurísticas H1 y H2 y que el reparto aleatorio de las tareas entre los procesadores (HR) es al menos dos ordenes de magnitud peor que H3 y un orden de magnitud peor que H1 y H2. De los datos de la Tabla II, se concluye que la heurística H3 es la que presenta mejores resultados, consiguiendo balancear la carga de forma casi exacta, por lo tanto esta es la heurística que produce mejores tiempos de ejecución en la evaluación de las implementaciones paralelas del problema del control de inventarios que se muestra en la Sección V.

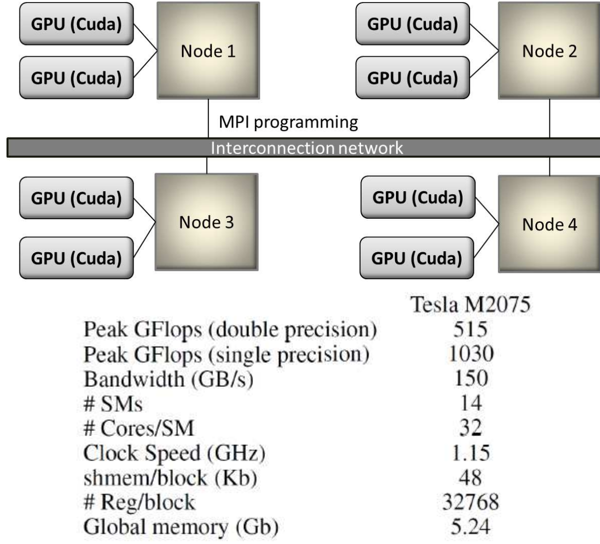
C. *Implementación Multi-GPU*

La versión Multi-GPU se ha basado en la explotación de diversas GPUs para la paralelización de las simulaciones del método de Monte-Carlo, realizadas por la función *flossGPU* (ver Algoritmo 5). Por una parte, cada una de las N simulaciones son independientes entre sí. Al mismo tiempo, para el cálculo de todo el inventario de cada una de las posibles edades (3), este solo depende del inventario del periodo anterior. Por tanto, separando por periodos, un kernel de CUDA puede realizar en paralelo el cálculo de las N simulaciones y, al mismo tiempo, la actualización del inventario de J edades diferentes. Por tanto, la computación que se realiza con la GPU es la simulación de Monte-Carlo. Al mismo tiempo, el modelo se ha implementado de forma que cada proceso MPI abre uno o dos hilos, que a su vez abren una o dos GPUs del nodo en el que se encuentra. Las tareas se reparten usando la heurística (H3) entre las GPUs que se vayan a utilizar en cada caso.

El Algoritmo 5 resume el cambio realizado en el Algoritmo 4 para adaptarlo a su ejecución en una o varias GPU. Tal y como se ha mencionado, el bucle que recorre los periodos se sitúa en el primer nivel. En la siguiente sección se muestran los resultados experimentales de la ejecución de las implementaciones MPI-PTHREADS y Multi-GPU.

V. Resultados

Para la evaluación de las implementaciones paralelas hemos utilizado un clúster compuesto de ocho nodos Bullx R424-E3 Intel Xeon E5 2650 (cada uno con 16 cores), interconectados por un puerto Infini-



	Tesla M2075
Peak GFlops (double precision)	515
Peak GFlops (single precision)	1030
Bandwidth (GB/s)	150
# SMs	14
# Cores/SM	32
Clock Speed (GHz)	1.15
shmem/block (Kb)	48
# Reg/block	32768
Global memory (Gb)	5.24

Fig. 3
(a) Cuatro nodos del cluster Multi-GPU utilizado para la evaluación del problema de control de inventarios y (b) características de las GPUs.

Algorithm 5 $flossGPU(q, t_1, t_2)$: Método Monte-Carlo para aproximar $E(\boldsymbol{X})$

1: **for** $t = t_1$ **to** t_2 **do**
2:     **for** $n = 1$ **to** N **do** #GPU (líneas 2-6)
3:         **for** $j = 1$ **to** $j = J$ **do**
4:             Update $I_{j,t,n}$ using (3)
5:         **end for**
6:     **end for**
7: **end for**
8: $X = \frac{1}{N} \sum_{n=1}^{N} \left(d_{t_2,n} - \sum_{j=1}^{J-1} I_{j,t_2-1,n} - q \right)^{+}$;
9: **return** X;

Band QDR/FDR embebido en la placa madre, 8-GB RAM y 16-GB SSD) con ocho GPUs TeslaM2075 (de los ocho nodos, cuatro de ellos tienen dos GPUs por nodo). El driver de CUDA que se ha utilizado es CUDA 6.5 [17]. La arquitectura Multi-GPU y las características de las GPUs se muestran en la Figura 3.

Las Tablas III, IV y V muestran los tiempos de ejecución, en segundos, del ejemplo del problema de control de inventarios de productos perecederos descrito anteriormente, en el que $T = 12$ y $J = 3$. Adicionalmente, se ha medido el tiempo tomado por cada heurística para asignar las tareas a los procesadores, siendo en el peor de los casos de aproximadamente mil microsegundos, por lo que todos ellos cumplen su función de balancear la carga en un tiempo que resulte insignificante para el problema. Cada proceso MPI se ejecuta en un nodo diferente del clúster (hemos evaluado con hasta ocho nodos), mientras que en cada uno de ellos se ha probado con hasta dieciséis hilos. La multiplicación del número de procesos MPI por el número de hilos, da como resultado el número de cores entre los que las heurísticas distribuiran la carga computacional asociada a los vectores Y del problema. En la tabla VI se recogen los tiempos cuando se aplica un reparto aleatorio de la carga entre los cores (HR).

TABLA III
Tiempos de ejecución del Algoritmo 1 usando la heurística H1 (en segundos).

MPI\threads	1	2	4	8	16
1	63.10	37.42	18.79	8.07	4.96
2	31.75	18.82	9.56	4.74	2.61
4	16.00	9.60	4.96	2.56	1.42
8	8.17	4.98	2.63	1.43	0.88

TABLA IV
Tiempos de ejecución del Algoritmo 1 usando la heurística H2 (en segundos).

MPI\threads	1	2	4	8	16
1	63.18	37.35	18.38	8.04	4.94
2	31.76	18.77	9.51	4.83	2.62
4	16.04	9.53	4.86	2.55	1.48
8	8.14	4.89	2.57	1.40	0.92

TABLA V
Tiempos de ejecución del Algoritmo 1 usando la heurística H3 (en segundos).

MPI\threads	1	2	4	8	16
1	63.14	37.22	18.57	8.04	4.80
2	31.72	18.76	9.51	4.77	2.59
4	16.01	9.49	4.87	2.21	1.42
8	8.14	4.89	2.54	1.40	0.87

TABLA VI
Tiempos de ejecución del Algoritmo 1 usando la heurística HR (en segundos).

MPI\threads	1	2	4	8	16
1	63.08	38.03	19.49	8.34	4.98
2	32.19	19.51	9.94	4.94	2.83
4	16.49	9.95	5.05	2.60	1.36
8	8.43	5.06	2.64	1.43	0.96

En cuanto a la paralelización MPI-PTHREADS, se puede apreciar en las tablas un buen nivel de speed-up para todas las heurísticas consideradas. Aunque en diversas pruebas, simulando el comportamiento de las heurísticas con problemas con una gran disparidad de carga computacional de las tareas (distribuciones γ_1, γ_2 y uniforme) se pueden apreciar diferencias entre ellas, para el problema de inventarios los tiempos de ejecución son similares, ligeramente inferiores para la heurística H3. En términos relativos, también puede apreciarse una diferencia en tiempos de ejecución con respecto al reparto a ciegas (HR).

En la Tabla VII se recogen los tiempos de ejecución del problema completo, tomando hasta las 8

GPUs existentes en el clúster para la paralelización del método Monte-Carlo en CUDA. En dicha tabla se aprecia como se pueden conseguir buenos resultados paralelizando una escasa porción del código. En cambio, al usar múltiples GPUs, la escalabilidad está penalizada por el tiempo de inicialización de las GPUs (aproximadamente 5 segundos), lo cual hace que no se obtenga un buen rendimiento con más de 2 GPUs para este ejemplo concreto. Para el ejemplo de inventarios considerado, el uso de los dieciséis cores de un solo nodo resulta más beneficioso que el uso de las dos GPUs disponibles.

TABLA VII
Tiempo en segundos para la paralelización del método Monte-Carlo para GPUs.

Number of GPUs	Tiempo
1	13.54
2	9.53
4	7.44
6	6.80
8	6.39

VI. Conclusiones

En este trabajo se ha analizado la implementación paralela del problema basado en el control de inventarios de productos perecederos. Dicha implementación se basa en tres ideas principales:

1. La explotación de plataformas multicore, basándonos en el uso de Pthreads y MPI.
2. El uso de la computación GPU para acelerar la parte computacionalmente más costosa.
3. El uso de computación MPI para distribuir la carga entre los distintos procesadores y poder utilizar GPUs ubicadas en diferentes nodos.

La paralelización MPI-PTHREADS con un balanceo estático de la carga basado en heurísticas presenta una buena escalabilidad. La rapidez con la que las heurísticas se ejecutan las hace apropiadas incluso para problemas en los que el desbalanceo no es muy acusado, mejorando un reparto aleatorio de las tareas. Para casos en los que el desbalanceo es mucho mayor (ejemplos con las distribuciones gamma y uniforme) el beneficio es mucho mayor y se observa que la heurística (H3) presenta mejores resultados. Con respecto a la paralelización del método Monte-Carlo en CUDA, la escalabilidad con el uso de múltiples GPUs se ve limitada por el tiempo de inicialización requerido, aunque podría ser beneficiosa en comparación con la paralelización MPI-PTHREADS si el problema a tratar requiere más precisión, siendo necesarias más simulaciones del método de Monte-Carlo, de forma que el tiempo de inicialización de las GPUs se hiciera comparativamente irrelevante.

Agradecimientos

Alejandro G. Alcoba es becario del programa FPI. Este trabajo ha sido financiado por Universidad de Málaga, Campus de Excelencia Internacional Andalucía Techel, el Ministerio de Ciencia (TIN2012-37483) y la Junta de Andalucía (P11-TIC7176), parcialmente financiados por el Fondo Europeo de Desarrollo Regional (FEDER).

Referencias

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 2011.

[2] A.L. Lastovetsky, “Special issue on heterogeneity in parallel and distributed computing,” *Journal of Parallel and Distributed Computing.*, vol. 73, no. 12, 2013.

[3] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman & Co., New York, NY, USA, 1979.

[4] A.A. Kurawarwala and H. Matsuo, “Forecasting and inventory management of short life-cycle products,” *Operations Research*, vol. 44, pp. 131–150, 1996.

[5] R. Rossi, S. Armagan Tarim, S. Prestwich, and B. Hnich, “Piecewise linear lower and upper bounds for the standard normal first order loss function,” *Applied Mathematics and Computation*, vol. 231, pp. 489–502, 2014.

[6] S. K. De Schrijver, El-H. Aghezzaf, and H. Vanmaele, “Double precision rational approximation algorithm for the inverse standard normal first order loss function,” *Applied Mathematics and Computation*, vol. 219, no. 3, pp. 1375 – 1382, 2012.

[7] G. R. Waissi and D. F. Rossin, “A sigmoid approximation of the standard normal integral,” *Applied Mathematics and Computation*, vol. 77, no. 1, pp. 91 – 95, 1996.

[8] A.G. Alcoba, E.M.T. Hendrix, I. Garcia, K.G.J. PaulsWorm, and R. Haijema, “On computing order quantities for perishable inventory control with nonstationary demand,” in *Proceedings of MAGO 2014*, 2014, pp. 5–8.

[9] A.G. Alcoba, E.M.T. Hendrix, I Garcia, G. Ortega, K. G. J. Pauls-Worm, and R. Haijema, “On computing order quantities for perishable inventory control with non-stationary demand,” in *Proceedings of ICCSA 2015.* 2015, vol. En prensa of *Lecture Notes in Computer Science*, Springer.

[10] D.R. Butenhof, *Programming with POSIX Threads*, Professional Computing Series. Addison-Wesley, 1997.

[11] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI-The Complete Reference, Volume 1: The MPI Core*, MIT Press, Cambridge, MA, USA, 1998.

[12] “CUDA C Best Practices Guide,” `http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/`, accessed 23 February 2015.

[13] C. Blum and A. Roli, “Metaheuristics in combinatorial optimization: Overview and conceptual comparison,” *ACM Comput. Surv.*, vol. 35, no. 3, pp. 268–308, Sept. 2003.

[14] Dror G. Feitelson, “Workload modeling for performance evaluation,” in *Performance Evaluation of Complex Systems: Techniques and Tools, Performance 2002, Tutorial Lectures*, London, UK, UK, 2002, pp. 114–141, Springer-Verlag.

[15] Zujie Ren, Jian Wan, Weisong Shi, Xianghua Xu, and Min Zhou, “Workload analysis, implications, and optimization on a production hadoop cluster: A case study on taobao,” *Services Computing, IEEE Transactions on*, vol. 7, no. 2, pp. 307–321, April 2014.

[16] Andrew Burkimsher, Iain Bate, and LeandroSoares Indrusiak, “Scheduling hpc workflows for responsiveness and fairness with networking delays and inaccurate estimates of execution times,” in *Euro-Par 2013 Parallel Processing*, Felix Wolf, Bernd Mohr, and Dieter an Mey, Eds., vol. 8097 of *Lecture Notes in Computer Science*, pp. 126–137. Springer Berlin Heidelberg, 2013.

[17] “NVIDIA CUDA TOOLKIT V6.5 (RN-06722-001_v6.5),” `http://docs.nvidia.com/cuda/pdf/CUDA_Toolkit_Release_Notes.pdf`, August 2014.