Optimización PSO paralelizada para scheduling de flow-shop

Leandro N. Salmieri[1], Javier Iparraguirre[2], Mariano Frutos[3], and Aníbal M. Blanco[1]

[1] Planta Piloto de Ingeniería Química - PLAPIQUI (Universidad Nacional del Sur-CONICET), Bahía Blanca, Argentina
{lsalmieri,ablanco}@plapiqui.edu.ar
[2] Universidad Tecnológica Nacional, Facultad Regional Bahía Blanca, Bahía Blanca, Argentina
j.iparraguirre@computer.org
[3] Departamento de Ingeniería, Universidad Nacional del Sur Bahía Blanca, Argentina
mfrutos@uns.edu.ar

Resumen El problema de scheduling de flow-shop (programación de la producción en una fábrica de flujo continuo) es de tipo NP-Hard, incluso para un número reducido de trabajos y de máquinas. Debido a su gran interés industrial, ha sido estudiado intensamente en las últimas décadas con el objeto de diseñar algoritmos que proporcionen soluciones de buena calidad en tiempos de cómputo aceptables para instancias de interés práctico. En este trabajo se presenta un algoritmo basado en optimización por enjambre de partículas (PSO) para el problema de scheduling de flow-shop. También se implementó una versión paralelizada que hace uso de placas gráficas NVIDIA utilizando la tecnología CUDA para acelerar las ejecuciones.

1. Introducción

El sistema flow-shop requiere que n trabajos $(J_1, ..., J_n)$ pasen secuencialmente por una serie de m máquinas $(M_1, ..., M_m)$. Cada trabajo debe ser procesado en cada máquina solo una vez, en un orden establecido, primero M_1, luego M_2, después M_3, y así hasta completar todas las máquinas. Cada máquina puede procesar un solo trabajo por vez.

Una versión del problema de scheduling de flow-shops (PSFS) consiste en identificar el orden en que deben secuenciarse los trabajos de manera de minimizar el makespan. El makespan se define como el tiempo que transcurre entre que el primer trabajo se inicia en la primera máquina y el último se completa en la última máquina.

Existen dos variantes principales del PSFP, la permutativa y la no permutativa. La primera requiere que todos los trabajos sigan la misma secuencia en todas las máquinas. Por ejemplo si el schedule es $J_1 - J_2$, en todas las maquinas primero se procesa J_1 y a continuación J_2. La versión no permutativa admite que en distintitas maquinas la secuencia de trabajos sea diferente. Por ejemplo en la maquina M_1 el ordenamiento puede ser $J_1 - J_2$ pero en la maquina M_2 puede ejecutarse $J_2 - J_1$. Estos conceptos se ilustran en la Fig. 1 donde la llave indica el makespan en cada caso. Si bien el problema no-permutativo es considerablemente más complejo que el permutativo, es interesante

notar que varios estudios reportan mejoras máximas de entre 1 y 3 % para el makespan de la versión no-permutativa respecto de la permutativa [2].

Ambas versiones del PSFS han sido intensamente estudiadas debido a su importancia práctica, e innumerables algoritmos se han propuesto para resolver instancias realistas de manera rápida y con buenos makespans, en lo posible globalmente óptimos. La programación matemática podría proporcionar en teoría las soluciones óptimas, aunque instancias grandes e incluso medianas se tornan intratables. Por esta razón también se han utilizado una gran variedad de técnicas meta-heurísticas que pueden llegar a proporcionar soluciones de aceptable calidad, aunque no necesariamente globalmente óptimas, para instancias de tamaño considerable. Una revisión de los diferentes enfoques excede el alcance de este trabajo. Se recomienda la lectura de [2] y [6] para revisiones de la literatura de problemas de scheduling de flow-shops.

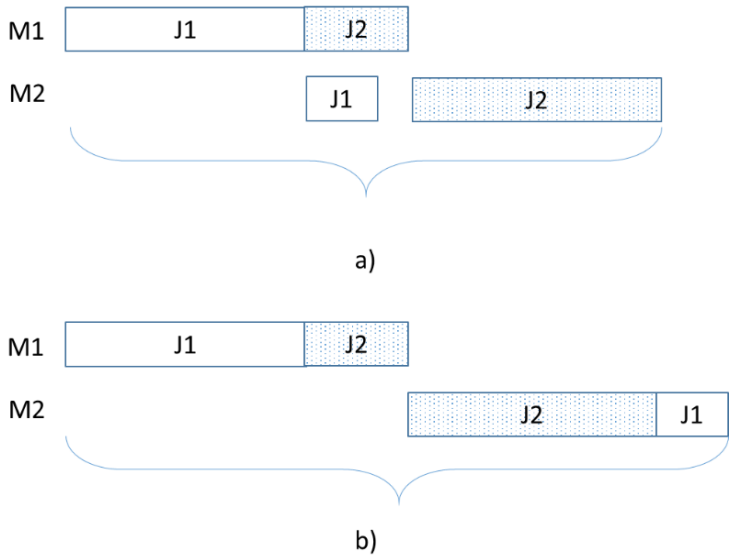

Figura 1: a) Permutativo, b) No-permutativo.

Formalmente la versión permutativa del PSFS posee la formulación expresada en las Ecuaciones 1-7 (adaptada de [2]).

$$minCmax \tag{1}$$

$$x_{mj} + p_{mj} \leq Cmax \veebar j \in [n] \tag{2}$$

$$x_{ij} + p_{ij} \leq x_{i+1,j} \veebar i \in [m-1] \veebar j \in [n] \tag{3}$$

$$x_{ij} + p_{ij} \leq x_{ij'} + M(1 - y_{jj'}) \veebar i \in [m] j \neq j' \in [n] \tag{4}$$

$$y_{jj} + y_{j'j} = 1 \veebar j \neq j' \in [n] \tag{5}$$

$$x_{ij} \geq 0 \veebar i \in [n] \veebar j \in [m] \tag{6}$$

$$y_{jj'} \in [0, 1] \veebar j \neq j' \in [n] \tag{7}$$

En esta formulación la variable binaria $y_{jj'}$ indica si el trabajo j precede al trabajo j'. La variable x_{ij} es el tiempo de inicio del trabajo j en la máquina i y el parámetro p_{ij} representa el tiempo ininterrumpido que demanda ese trabajo en esa máquina. La ecuación (1) representa la función objetivo. La ecuación (2) define el makespan. La ecuación (3) exige que un trabajo finalice en una máquina antes de iniciar el siguiente. La ecuación (4), de tipo big M, garantiza que los trabajos se procesen en el orden definido por las variables $y_{jj'}$. El parámetro M se calcula típicamente como la suma de todos los p_{ij}. La ecuación (5) fuerza que los trabajos se realicen en secuencia. Las ecuaciones (6) y (7) establecen los tipos de variables. Esta formulación define un problema de programación entera que se sabe es NP-Hard para $min(n, m) > 3$ [2].

En este trabajo se propone un algoritmo basado en optimización por enjambre de partículas para abordar el PSFS permutativo y se explora el desempeño de una versión paralelizada sobre una placa gráfica.

2. PSO aplicado a PSFS

La optimización por enjambre de partículas (PSO, por sus siglas en inglés) es una meta-heurística basada en poblaciones inspirada en los principios de búsqueda empleados por los enjambres de insectos en la búsqueda de alimento [5]. Desde su propuesta inicial en 1995, esta técnica ha sido mejorada progresivamente y aplicada en infinidad de problemas debido a su sencillez de programación, flexibilidad y buenas características de rapidez y convergencia.

El algoritmo PSO se rige por dos ecuaciones básicas. La posición (ecuación (8)) de cada individuo de la población $\mathbf{z}^k$, se modifica en cada iteración (k) por medio de un término de velocidad $\mathbf{v}^k$. El periodo de tiempo (t) se asume igual a la unidad. El término de velocidad (ecuación (9)) posee a su vez un componente inercial que depende de la velocidad en el instante anterior, un término individual o cognitivo que dirige a la partícula al mejor lugar identificado por ella misma durante su recorrido hasta el momento ($\mathbf{p}^k$) y un término social que dirige a la partícula a la mejor posición encontrada por todo el enjambre hasta el momento ($\mathbf{g}^k$).

$$\mathbf{z}^{k+1} = \mathbf{z}^k + \mathbf{v}^{k+1} t \tag{8}$$

$$\mathbf{v}^{k+1} = w\mathbf{v}^k + c_1\mathbf{r}_1(\mathbf{p}^k - \mathbf{z}^k) + c_2\mathbf{r}_2(\mathbf{g}^k - \mathbf{z}^k) \tag{9}$$

Las matrices $\mathbf{z}$, $\mathbf{v}$, $\mathbf{p}$ y $\mathbf{g}$ poseen tantas columnas como dimensiones tiene el problema de optimización (D) y tantas filas como partículas tienen la población (Np). Cabe aclarar que la matriz $\mathbf{g}$ posee todas sus filas idénticas e iguales a la posición del mejor individuo del enjambre en su historia y desde el punto de vista de la estructura de datos puede tratarse directamente como un vector.

Las constantes w, c_1 y c_2 son parámetros propios del método y las matrices $\mathbf{r}_1$ y $\mathbf{r}_2$ tienen elementos aleatorios entre 0 y 1 que proporcionan la componente estocástica a la búsqueda. El valor de la función objetivo y el grado de verificación de las restricciones del problema se encuentra de manera implícita en los términos $\mathbf{p}^k$ y $\mathbf{g}^k$.

Para poder adaptar la formulación PSO al PSFS se propone una estructura de datos vectorial donde el conjunto solución, es decir la secuencia de trabajos, es almacenado en un vector de largo n. En la función destinada a evaluar el desempeño de cada partícula, se calcula el makespan asegurando que en todas las máquinas se procesen los trabajos en el mismo orden, y que cada trabajo no puede ser procesado por la maquina i hasta no terminar de ser procesado por la maquina (i-1). La representación propuesta para la solución presenta dos problemas para ser utilizada directamente con PSO: (a) PSO trabaja solo con variables continuas, por lo cual deberíamos incorporar una herramienta que transforme números reales en números enteros; (b) Es necesario incorporar una codificación que evite los individuos no factibles.

Para resolver simultáneamente ambos problemas, proponemos adoptar la bien conocida representación Random Key, la cual codifica un conjunto de números reales aleatorios en un conjunto de números enteros consecutivos sin repeticiones [1, 4]. Esta técnica permite convertir el arreglo de números reales que representa la posición de las partículas en PSO, $\mathbf{z}_p = [z_1, z_2, ..., z_n]$, en un arreglo de números enteros que representa el orden a ser procesados los trabajos $\mathbf{z}_J = [J_1, J_2, ..., J_n]$ de manera que la partícula pueda ser evaluada respecto de su correspondiente makespan. Aquí $\mathbf{z}_p$ representa una fila de la matriz $\mathbf{z}$ de las ecuaciones (8) y (9). Los valores de los elementos de $\mathbf{z}_p$ se generan entre $[0, z_{pmax}]$ donde z_{pmax} es un parámetro de magnitud apropiada seleccionado más o menos arbitrariamente, por ejemplo igual al número de trabajos del flow-shop.

Para esto, $\mathbf{z}_p$ se pasa a un algoritmo de ordenamiento, el cual copia el arreglo original en uno auxiliar ($\mathbf{Aux} = [aux_1, aux_2, ..., aux_n]$), y luego lo ordena de menor a mayor. Al culminar el ordenamiento, se ejecuta una rutina que compara $\mathbf{z}_p$ y $\mathbf{Aux}$ y devuelve el vector de secuencia de trabajos $\mathbf{z}_J$ con el cual se procede al cálculo del makespan. Por ejemplo, para una instancia de 6 trabajos (n = 6) con una posición:

$$\mathbf{z}_p = [0{,}06, 2{,}99, 1{,}86, 3{,}73, 2{,}13, 0{,}67]$$

el vector $\mathbf{Aux}$ correspondiente es:

$$\mathbf{Aux} = [0{,}06, 0{,}67, 1{,}86, 2{,}13, 2{,}99, 3{,}73]$$

y el vector de trabajos ordenados es:

$$\mathbf{z}_J = [1, 6, 3, 5, 2, 4]$$

El algoritmo PSO (ecuaciones (8) y (9)) junto con la técnica Random Key fue implementada en lenguaje de programación C.

3. Paralelización en placa gráfica

El procesamiento paralelo es la forma más intuitiva de acelerar algoritmos que admiten tareas que se pueden ejecutar de forma independiente de manera simultánea. Las meta-heurísticas basadas en poblaciones, en particular el PSO, poseen un paralelismo implícito (cada miembro de la población está sujeto a las mismas operaciones) que se puede explicitar si se cuenta con múltiples núcleos de procesamiento y se lo programa en una plataforma adecuada que explote esos recursos.

Una opción es utilizar los varios núcleos que proporcionan las CPU modernas utilizando por ejemplo el lenguaje OpenMP. Otra alternativa es emplear las placas gráficas de propósito general (GPU, por sus siglas en inglés) de marca NVIDIA utilizando el modelo de programación CUDA. Alternativamente pueden utilizarse GPUs de cualquier marca empleando el lenguaje abierto OpenCL.

En los últimos años ha habido un notable desarrollo en las GPU de propósito general motivado inicialmente por la industria de los video juegos y más recientemente para su empleo en cómputo científico de alto desempeño. Las GPU son dispositivos de relativo bajo costo incluidos en las computadoras de escritorio y notebooks modernas que pueden realizar un gran número de operaciones simples en forma simultánea debido a su arquitectura masivamente paralela. Además de numerosas unidades de aritmética lógica, las GPU cuentan con distintos tipos de memorias que permiten efectuar los cálculos y la comunicación con la CPU.

Desde el punto de vista de la programación, la GPU pone a disposición del usuario una gran cantidad de hilos (threads), organizados por bloques. Una GPU moderna soporta hasta 80 bloques con 1024 hilos cada uno como máximo. Cada hilo puede ejecutar una operación de manera independiente, posibilitando la paralelización de las tareas. La placa permite sincronizar los hilos de manera de mantener un control sobre los segmentos de código que se ejecutan en paralelo.

Desde su aparición en la década del 90, las GPU se han aplicado a infinidad de problemas de interés ingenieril, en particular a programar diversos algoritmos de optimización basados en poblaciones. En [7] puede encontrarse una revisión bastante completa hasta el año 2016, junto a una buena descripción de las principales fortalezas y debilidades de esos desarrollos.

Para estudiar el desempeño del algoritmo para PSFS descripto en la sección anterior, se programó también una versión paralelizada utilizando el lenguaje CUDA sobre placas NVIDIA disponibles en el grupo de investigación.

Existen varias formas de programar el algoritmo PSO para explotar el gran número de hilos que proporciona la GPU. Algunos autores han utilizado un hilo por partícula, lo que obliga a tratar cada dimensión (variable de optimización) en serie dentro del hilo. Esto permite utilizar, en versiones de PSO asincrónicas, poblaciones enormes, tan grandes como el número de hilos que posee la placa (80x1024 por ejemplo), pero es poco eficiente al momento de evaluar las funciones involucradas que es el proceso habitualmente más costoso del algoritmo. Además, se sabe que la calidad de la convergencia del algoritmo se vuelve insensible a partir de poblaciones de cierto tamaño (por ejemplo mas de 50) incluso de manera independiente del número de dimensiones del problema, por lo cual contar con una población excesivamente grande no se traduce necesariamente en una ventaja.

Otra opción es utilizar un hilo por dimensión y alojar toda la población en un solo bloque [3]. Esto permite hacer un uso muy efectivo de memoria local de los bloques pero posibilita abordar problemas de tamaño limitado, restringidos por $D.Np < 1024$. Además, la mayoría de los bloques de la GPU quedan sin uso.

Un tercer enfoque es emplear una partícula por bloque y una variable por hilo. Esto permitiría resolver problemas con Np igual al número de bloques disponibles en la placa, por ejemplo 80, que es un número adecuado para muchas aplicaciones, y con D igual al número de hilos por bloque, por ejemplo 1024, que puede ser un número de variables interesante en ciertas aplicaciones. Hasta hace relativamente poco tiempo el modelo CUDA no poseía una herramienta nativa que permitiera sincronizar bloques, por lo cual la versión de PSO sincrónica (ecuaciones (8) y (9)) no podría implementarse directamente (aunque sí versiones alternativas de PSO asincrónicas). Sin embargo a partir de la versión 10.0 de CUDA (setiembre de 2018) se incorporaron herramientas que permiten la sincronización de grupos de hilos de manera flexible. Se adoptó este último esquema para desarrollar el presente estudio.

La métrica típica para cuantificar el desempeño de las versiones paralizadas de algoritmos es el SpeedUp (SU) definido en la ecuación 10, donde t_{CPU} y t_{GPU} son respectivamente los tiempos correspondientes a las versiones serie y acelerada del algoritmo.

$$SU = \frac{t_{CPU}}{t_{GPU}} \tag{10}$$

4. Resultados

En la Tabla 1 se detallan las seis instancias estudiadas en este trabajo, correspondientes a distintos números de trabajos (n) y de máquinas (m). En todos los casos, los tiempos de procesamiento de cada trabajo en cada máquina fueron generados aleatoriamente a partir de una distribución uniforme: p_{ij}=U[1, 99].

Para dar una idea de la dificultad de los problemas se programó el PSFS (ecuaciones 1-7) en la plataforma GAMS y se resolvieron las instancias propuestas usando el solver CPLEX empleando un tiempo máximo de ejecución de 1.200,00 seg. en un equipo con las siguientes características: Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz. Estos resultados también se presentan en la Tabla 1. Solo en la instancia más pequeña el solver fue capaz de proporcionar la solución óptima. El las instancias intermedias el solver encontró soluciones buenas aunque sub-óptimas. En las instancias más grandes, CPLEX no devolvió solución luego de 1.200,00 seg. de tiempo de CPU.

Cabe mencionar que se empleó CPLEX con su parametrización por defecto para proporcionar simplemente una idea de la dificultad de las instancias abordadas. Es posible que con inicializaciones ad hoc u otras parametrizaciones el solver proporcione desempeños superiores. Un estudio de estas características excede los alcances de este trabajo.

En la Tabla 2 se proporcionan los resultados comparativos para las versiones serie (lenguaje C) y paralelo (lenguaje CUDA) del algoritmo PSO descripto en secciones previas utilizando una parametrización típica ($w = 0{,}75$, $c_1 = c_2 = 1{,}5$). Para todos los experimentos se adoptó una población de 30 partículas. Si bien el número de bloques

Cuadro 1: Definición de instancias y resultados con GAMS/CPLEX.

I	m	n	GAMS/CPLEX		
			Makespan	tCPU(s)	gap
1	4	8	585,24	0,91	0,00
2	4	50	2.808,94	1.200,00	0,81
3	4	100	5.421,61	1.200,00	0,92
4	10	100	6.578,57	1.200,00	0,88
5	50	100	No solution	1.200,00	-
6	50	150	No solution	1.200,00	-

que posee la GPU utilizada (NVIDIA GeForce GTX 1060) permite en teoría emplear muchas más partículas (80), es necesario alojar en la memoria local del procesador las matrices de datos, en particular los tiempos de ejecución de cada trabajo en cada máquina p_{ij}. Para la instancia mayor (I=6), esta matriz es de 50x150 lo cual impone una carga considerable a la memoria del dispositivo e impide el empleo de un mayor número de partículas en el swarm. Si bien las instancias más pequeñas admiten el uso de un mayor número de individuos al ser más pequeña la matriz de tiempos, se prefirió adoptar un único tamaño de swarm para todos los experimentos. Los tiempos de ejecución de los algoritmos corresponden en todos los casos a 10.000 iteraciones.

Cuadro 2: Comparación versiones serie y paralela de PSO.

I	Serie			Paralelo			SU
	Makespan	tCPU (s)	% Error	Makespan	tGPU (s)	% Error	
1	597,53	0,15	2,10	585,24	3,47	0,00	0,04
2	2.928,83	31,13	4,27	2.906,45	21,89	3,47	1,42
3	5.908,95	205,51	8,99	5.907,51	121,01	8,96	1,70
4	6.250,69	208,67	-4,98	6.296,46	121,15	-4,29	1,72
5	10.071,35	212,80	-	10.153,38	121,67	-	1,75
6	13.617,84	647,27	-	13.775,26	444,61	-	1,46

Para dar una idea del desempeño de ambas versiones del PSO se ejecutaron 30 corridas de cada instancia reportándose en la Tabla 2 los valores promediados. En particular, la columna %Error compara el resultado del algoritmo con la versión correspondiente de GAMS/CPLEX. Puede observarse que para las instancias 1 a 3 las soluciones de CPLEX son superiores a las encontradas por los algoritmos PSO. Para la instancia 3, el PSO es superior a la de CPLEX y para las 2 restantes PSO devuelve una solución factible mientras que CPLEX no, aún después de 1.200,00 s de CPU.

En lo que respecta a los tiempos entre la versión serie y la paralelizada, en la instancia más pequeña la versión CPU es superior a la GPU. Para las instancias más grandes se observan reducciones de entre el 42 y 75 % en los tiempos de la versión paralelizada respecto de la serie. Respecto de la calidad de las soluciones se observa que la versión serie presenta ligera mejoras por sobre la paralela para las instancias

más grandes. La discrepancia se atribuye básicamente a la componente estocástica del algoritmo. Igualmente no se descarta que el manejo de números por C en el CPU y por CUDA en la GPU tenga alguna variación que se traslade a las diferencias observadas.

5. Conclusiones

Si bien los Speed Up logrados con la versión GPU no lucen impresionantes, indican el potencial del uso del dispositivo para acelerar problemas de complejidad combinatoria como el PSFS abordado en este trabajo. Se espera que el desempeño de la versión acelerada se incremente en instancias de mayor dimensión. Diversos experimentos (no mostrados) indican que los tiempos de cómputo de la versión serie aumentan significativamente con el número de trabajos y son relativamente insensibles al número de máquinas. Por otra parte, resultados propios y numerosas referencias bibliográficas indican que el Speed Up se incrementa de manera aproximadamente lineal con el número de partículas empleadas en la población. Para instancias mayores que las utilizadas en este trabajo $(m > 50, n > 150)$ se requiere también un mayor número de pobladores $(Np > 30)$ para asegurar una buena exploración del espacio de soluciones. Estas características sugieren un potencial de aceleración si se lograra emplear efectivamente la GPU en estas instancias. Por el momento, la memoria del dispositivo es un cuello de botella para abordar ese tipo de problemas. Para resolver esta limitación, un trabajo en curso y futuro involucraría el uso de GPU de mayores prestaciones, el empleo de múltiples GPU, la programación de PSO multi-swarm y el uso de datos half-precision en los cálculos.

Referencias

1. Bean, J.C.: Genetic algorithms and random keys for sequencing and optimization. ORSA journal on computing **6**(2), 154–160 (1994)
2. Benavides, A.J., Ritt, M.: Two simple and effective heuristics for minimizing the makespan in non-permutation flow shops. Computers & Operations Research **66**, 160–169 (2016)
3. Damiani, L., Diaz, A.I., Iparraguirre, J., Blanco, A.M.: Accelerated particle swarm optimization with explicit consideration of model constraints. Cluster Computing (Apr 2019). https://doi.org/10.1007/s10586-019-02933-1, https://doi.org/10.1007/s10586-019-02933-1
4. Liu, B., Wang, L., Jin, Y.H.: An effective pso-based memetic algorithm for flow shop scheduling. IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics) **37**(1), 18–27 (2007)
5. Marini, F., Walczak, B.: Particle swarm optimization (pso). a tutorial. Chemometrics and Intelligent Laboratory Systems **149**, 153–165 (2015)
6. Rossit, D.A., Tohmé, F., Frutos, M.: The non-permutation flow-shop scheduling problem: a literature review. Omega **77**, 143–153 (2018)
7. Tan, Y., Ding, K.: A survey on gpu-based implementation of swarm intelligence algorithms. IEEE transactions on cybernetics **46**(9), 2028–2041 (2015)