PONTIFICIA UNIVERSIDAD CATÓLICA DE VALPARAÍSO
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA INFORMÁTICA

Diseño e Implementación de Un Algoritmo Paralelo Basado en Tabu Search Para La Resolución del Problema de Manufacturing Cell Design

Félix Fuentes González

TESIS DE GRADO
MAGÍSTER EN INGENIERÍA EN INFORMÁTICA

Octubre 2013

PONTIFICIA UNIVERSIDAD CATÓLICA DE VALPARAÍSO
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA INFORMÁTICA

Diseño e Implementación de Un Algoritmo Paralelo Basado en Tabu Search Para La Resolución del Problema de Manufacturing Cell Design

Félix Fuentes González

Profesor Guía: **Broderick Crawford Labrín**

Programa: **Magíster en Ingeniería Informática**

Octubre 2013

Abstract

This work addresses the Manufacturing Cell Design problem through the use of the Tabu Search metaheuristic, the resolution method presented is designed in a parallel manner, and targeted towards an implementation in graphics processing units (GPU), using the now standard framework OpenCL, making use of the capacity of making general purpose processing or GPGPU in these devices. The aim is to produce a solver whose results can be later compared to previous works, that have made use of metaheuristics for the resolution of the Manufacturing Cell Design problem.

Keywords: Tabu search metaheuristics, Parallelization strategies, Comparison, Manufacturing Cell Design.

Resumen

En este trabajo se trata el problema de Manufacturing Cell Design mediante la metaheurística de Tabu Search, el método de resolución presentado está diseñado de una manera paralela pensado para su implementación en unidades de procesamiento de gráficos o GPU, utilizando el framework estándar OpenCL para realizar procesamiento de propósito general en GPU. El objetivo es producir un solver cuyos resultados puedan ser luego comparados con trabajos anteriores que hayan empleado metaheurísticas para la resolución del problema de Manufacturing Cell Design.

Palabras Clave: Tabu search metaheurísticas, Estrategias de Paralelización, Comparación, Manufacturing Cell Design.

Índice

Lista de Figuras

Lista de Tablas

1. Introducción

En el problema de diseño de manufactura en celdas o «Manufacturing Cell Design» se trabaja con un sistema conformado de máquinas y partes, en el cuál se busca minimizar los movimientos que deben realizar las partes entre las distintas máquinas que son requeridas para la finalización de un producto en específico y de esta manera optimizar los costos de producción [4]. Para su resolución, se han empleado distintas técnicas metaheurísticas [9, 33, 18] (incompletas) y otras técnicas completas como la presentada por [30] .

Para este trabajo se ha elegido la metaheurística de Tabu Search o Búsqueda Tabú, presentada originalmente en [21, 22] , que ha sido usada extensivamente [8, 19, 13] y generalmente entregando buenos resultados [25]. Tabu Search, el que además tiene un potencial importante en cuanto a paralelización, como se muestra en [10] y [11] , esto significa que los elementos principales de esta metaheurística (ver 3.4) permiten descomponer una instancia del algoritmo en varias partes, o sub instancias, independientes que pueden ser ejecutadas en hardware multi procesador, y luego reunidas para entregar los resultados finales de la ejecución.

Recientemente la tecnología GPU (Graphics Processing Unit) ha ganado popularidad como hardware para cálculos de propósito general, al haber evolucionado a un nivel de sofisticación programática que permite realizar operaciones desde y hacia la memoria principal, en vez de solamente hacia la memoria de video, y, debido a que dedican la mayoría de sus transistores a unidades de proceso aritmético-lógicas, pueden realizar una cantidad mucho mayor de operaciones de este tipo por ciclo de reloj en comparación a una CPU [31].

Al unir la capacidad de paralelización de Tabu Search y la capcidad de ejecutar código en forma paralela de la computación por GPU se puede lograr un aprovechamiento mayor de los recursos hardware en una arquitectura PC tradicional y obtener rendimiento mucho más alto. El presente trabajo se divide en la presentación del problema, una revisión de la literatura existente en relación a los métodos de resolución más comunes usados en el Manufacturing Cell Design, las metaheurísticas, los modelos de computación paralela, su uso más específicamente con Tabu Search, y su implementación en arquitecturas GPU. Luego se presenta el Diseño realizado, la implementación y pruebas con respecto a una implementación secuencial de referencia, el análisis de los resultados obtenidos y por último conclusiones y comentarios finales.

2. Objetivos y metodología

2.1. Objetivo General

- Diseñar e implementar un algoritmo paralelo basado en la metaheurística de Tabu Search para la resolución del Manufacturing Cell Design.

2.2. Objetivos específicos

- Comprender el problema de Manufacturing Cell Design.
- Entender las distintas técnicas de paralelización en algoritmos de optimización.
- Aplicar técnicas de paralelización en un diseño estándar de Tabu search.
- Implementar un solver siguiendo el diseño establecido.
- Realizar pruebas contra resultados obtenidos por otros trabajos en metaheurísticas aplicadas al problema.

2.3. Análisis de objetivos

Para poder lograr el objetivo principal que es una implementación exitosa de un solver para el problema de Manufacturing Cell Design (MCD) usando búsqueda tabú sobre una arquitectura paralela, es necesario descomponer la tarea en varias subtareas que permitan secuencialmente ir desde el entendimiento del problema y los métodos de solución hasta la optimización de la implementación sobre el hardware.

Para comprender el problema es necesario estudiar la literatura existente y obtener un modelado matemático. Luego es necesario comprender la metaheurística de Tabu Search, también acudiendo a la literatura existente haciendo énfasis a trabajos anteriores que traten el mismo problema. Para poder pasar desde el diseño de Tabu Search canónico (algoritmo secuencial) se debe investigar los distintos enfoques de paralelización que se han desarrollado para esta y otras metaheurísticas.

Con el objetivo de imeplementar el diseño creado luego de trabajar en los puntos anteriores se debe hacer una investigación suficientemente amplia de las técnicas de optimización para la implementación del solver en la arquitectura elegida y de esta manera poder tener el mayor rendimiento posible

sobre el hardware seleccionado.

Para poder contextualizar los resultados obtenidos respecto de trabajos anteriores dedicados a la resolución del MCD se deben realizar pruebas con instancias conocidas del problema y se espera obtener resultados al menos al nivel de trabajos anteriores.

2.4. Metodología

Como metodología para el presente trabajo se ha optado por un enfoque iterativo, otorgándole inicialmente mayor énfasis a las etapas de investigación y diseño para luego volcarse gradualmente a la implementación, a medida que el diseño pasa a completarse de acuerdo a la investigación realizada. Se espera que el diseño del algoritmo sufra cambios a lo largo de la mayor parte del proceso, finalizando éstos a medida que se ejecutan pruebas preliminares con la implementación, con el objetivo de optimizar su desempeño o sencillamente corregir algún comportamiento inesperado que pudiera surgir del diseño establecido previamente. Los resultados obtenidos se reporta en las secciones de resultados y análisis de los mismos, en los que se comunican los resultados obtenidos y la comparación con resultados conocidos de otros trabajos.

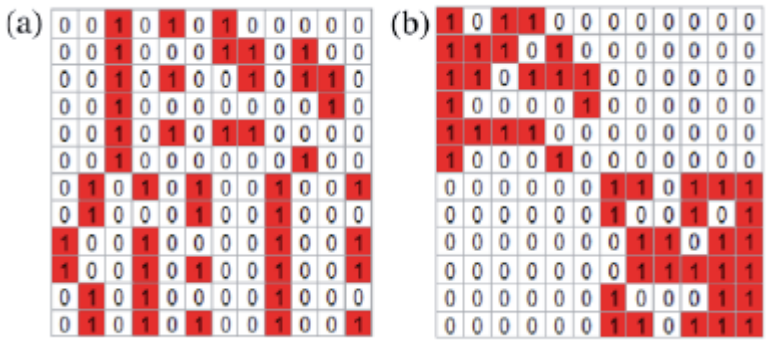

Figura 1: Matriz de incidencia inicial (a) y reagrupada (b)

3. Revisión de la literatura

3.1. Problemática

El objetivo de Manufacturing Cell Design es descomponer una línea de ensamblaje muy compleja en varios grupos de máquinas (celdas), cada una siendo dedicada a procesar una familia de partes. En la Figura 1 se muestra gráficamente una matriz de agrupaciones (matriz de incidencia) inicial y otra ordenada, en donde se ve claramente cómo se han formado los grupos de manufactura. Debido a esto, cada tipo de parte es idealmente producida por una sola celda. Luego el flujo de material se simplifica y la calendarización de las tareas se hace mucho mas fácil. Manufacturing Cell Design es un acercamiento organizacional basado en la tecnología de grupo [18] (group technology, GT).

Para maximizar la eficiencia del sistema, un problema de formación de celdas es resuelto para particionar el sistema en subsistemas que son tán autónomos como es posible, en el sentido que las interacciones de las máquinas y las partes dentro de un subsistema están maximizadas, y que las interacciones entre máquinas y partes de otros subsistemas son reducidas tanto como es posible. Esto da paso a la resolución de un problema de formación de celdas.

El problema de formación de celdas es un problema de optimización de clase NP [17]. Por esta razón, varios métodos metaheurísticos han sido desarrollados a lo largo de los últimos cuarenta años para generar buenas soluciones en un tiempo computacional razonable. Aunque, sólo algunas formulaciones de programación entera [26, 3] han sido propuestas para resolver pro-

blemas más pequeños de manera de poder evaluar la calidad de las soluciones propuestas por los métodos metaheurísticos. Este proceso de identificación requiere una formulación efectiva para formar familias de partes de manera que la similaridad dentro de una familia de partes pueda ser maximizada. En general el método más usado para el Manufacturing Cell Design es el análisis de agrupamiento [4] (clustering analysis). Sin embargo, debido a que éste problema es de clase NP [17] , existe aún el desafío de crear un método de agrupado eficiente.

3.2. Métodos de resolución

Aquí se presenta una algoritmo paralelo basado en el método de la búsqueda tabú presentado originalmente por F. Glover [21, 22].
Se discutirán distintas estrategias de paralelización para esta metaheurística, además de comparar sus distintas ventajas y desventajas frente a la implementación en una arquitectura SIMD (Single Instruction, Multiple Data) como son las GPU actuales. Luego se presentará el algoritmo propuesto en concordancia con los puntos anteriormente detallados.

3.2.1. Manufacturing cell design y metaheurísticas

El Manufacturing Cell Design (MCD) pertenece a la categoría de problemas de dificultad NP [2], lo que quiere decir que no puede ser resuelto en un tiempo polinomial para instancias no triviales.

Existen muchas consideraciones a tomar en el diseño y planificación de un problema de MCD, como por ejemplo el considerar el problema de disposición dentro del problema de MCD [28], planificación de concurrencia de producción en MCD, además, la planificación simultánea de MCD [25],[24] , etc. Los elementos excepcionales son definidos como partes, las cuáles deben ser procesadas en distintas celdas y por ende tienen movimientos inter-celda. [5] desarrolló un modelo que introduce distintos estados para elementos excepcionales considerando movimientos inter-celda e intra-celda, duplicación de máquinas y costos de subcontratación. [3] propuso un acercamiento integrado para rediseñar MCD considerando énfasis en los aspectos de rediseño. Su acercamiento usó simulación basada en módulo de planificación.

3.3. Técnicas de optimización combinatoriales

Nuestra habilidad para resolver grandes problemas combinatoriales de optimización ha mejorado dramáticamente durante la década pasada y en lo

que va de ésta. La disponibilidad de software confiable, hardware extremadamente rápido y barato y lenguajes de alto nivel que hacen el modelamiento de problemas complejos mucho mas rápido, han llevado a una mayor demanda por herramientas de optimización. En el pasado grandes proyectos de investigación requerían una gran recolección de datos, mainframes extremadamente caros y muchas horas-hombre de analistas. Ahora, podemos resolver problemas mucho más grandes en computadores personales, muchos de los datos necesarios se recolecta rutinariamente y existen herramientas para acelerar tanto el modelamiento como el análisis de optimalidad posterior.

Los intentos de resolver los problemas de agrupamiento de máquinas (Manufacturing Cell Design) pueden ser clasificados en dos grupos [18]. El primer grupo consiste en algoritmos que intentan determinar la solución óptima, y el segundo consiste en métodos heurísticos o métodos incompletos, que trabajan sobre una parte de todas las soluciones posibles y no garantizan encontrar el óptimo global.

3.3.1. Técnicas completas

Consisten en algoritmos que intentan determinar la solución óptima. Podemos mencionar trabajos como [30] donde se utilizan técnicas de constraint programming para dar solución al Manufacturing Cell Design Problem. También es importante notar que no cualquier instancia del problema es apta para resolverse con una técnica completa, especialmente las instancias más grandes en donde la complejidad aumenta abruptamente debido a la cantidad inmensa de combinaciones a probar, que no son reducibles con, por ejemplo, los algoritmos de constraint programming, a un nivel que sea abordable por la capacidad hardware de los computadores (PC) actuales.

3.3.2. Técnicas incompletas

Consiste mayormente en (meta)heurísticas, en las cuales generalmente se explora solamente una parte del espacio total de búsqueda, de manera de poder alcanzar un tiempo razonable de cálculo para problemas muy grandes, a la vez que no se garantiza encontrar la solución óptima, pero sí una factible en muchos casos.

En años recientes, diferentes métodos metaheurísticos han sido usados para resolver el problema de Manufacturing Cell Design [7, 33]; recientemente, [35] usaron Simulated Annealing, [25] presenta un algoritmo basado en Tabu Search, [32] usa un método basado en ACO y algoritmos genéticos,

[18] usa el método de PSO (particle swarm optimization) y [24] presenta una técnica basada en electromagnetismo.

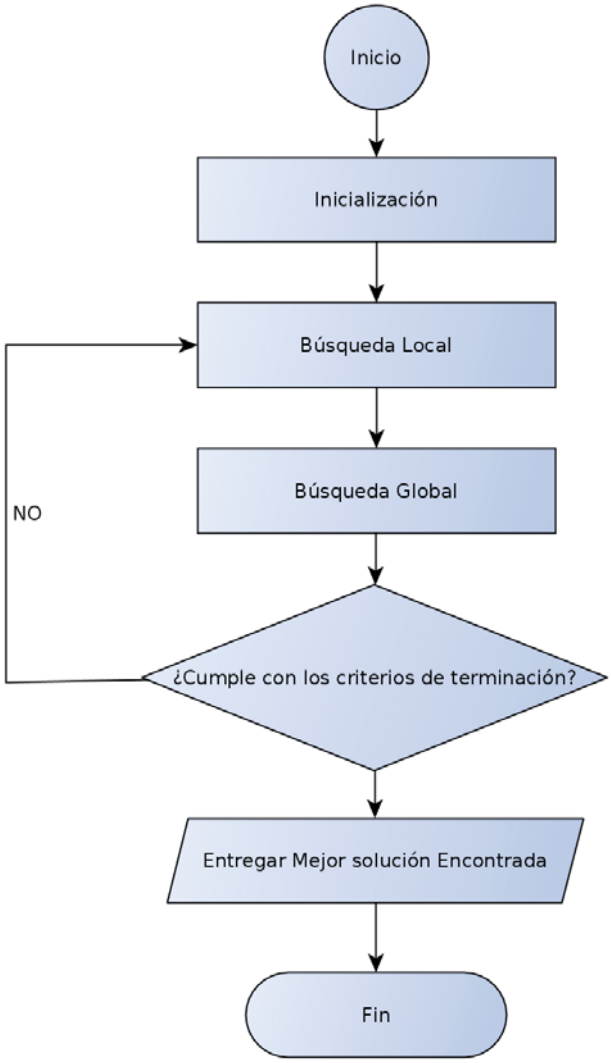

Figura 2: Procedimiento general de búsqueda tabú

3.4. La metaheurística de búsqueda tabú

Aquí se presentan brevemente los componentes de la búsqueda tabú. [21, 22, 6, 23]

Un procedimiento de Tabu Search esquemático para la resolución de un problema de optimización, según [11] puede ser modelado de la siguiente manera:

$$(P)\ Minimizar\ f(x)\ sujeto\ a\ x \in X \subseteq R^n$$

Puede ser visto como una combinación de tres pasos principales: búsqueda local, intensificación de la búsqueda de una subregión seleccionada, mover la búsqueda a una región previamente inexplorada. Mientras se explora el dominio de acuerdo a las reglas de uno de estos procedimientos, se recolecta, se guarda y procesa conocimiento para poder ganar un entendimiento del problema y de su dominio, para extraer una imagen de una buena solución, para identificar regiones donde estas buenas soluciones se pueden encontrar, y para guiar la búsqueda. Típicamente la búsqueda local es llevada a cabo evaluando movimientos de una solución actual $\bar{x} \in X$. Estos pasos se presentan a gran escala en la Figura 2.

Un movimiento es cualquier procedimiento que permita pasar de una solución (P) a una solución distinta de (P) (a estas soluciones puede permitírseles ser infactibles). Todas las soluciones pueden ser alcanzadas desde $\bar{x}$ desde el vecindario $N(\bar{x})$ a $\bar{x}$.

La búsqueda local puede entonces ser ejecutada sobre el vecindario completo, o solamente un subconjunto identificado como un listado de candidatos en la iteración k, $C(\bar{x}, k) \subseteq N(\bar{x})$. El mejor movimiento-candidato, con respecto a un criterio específico (usualmente basado en el valor objetivo), en la lista de candidatos es seleccionado e implementado.

3.4.1. Componentes

Método de búsqueda tabú:

Una representación conveniente de búsqueda en el entorno identifica, para cada solución $\bar{x} \in X$, o un conjunto asociado de vecinos, $N(\bar{x}) \subset X$, llamado entorno de $\bar{x}$. En búsqueda tabú, los entornos normalmente se asumen simétricos, es e decir, $\bar{x}$ es un vecino de $\bar{x}$ si y sólo si $\bar{x}$ es un vecino de $\bar{x}$. Esto se conoce también como búsqueda local y se describe a continuación.

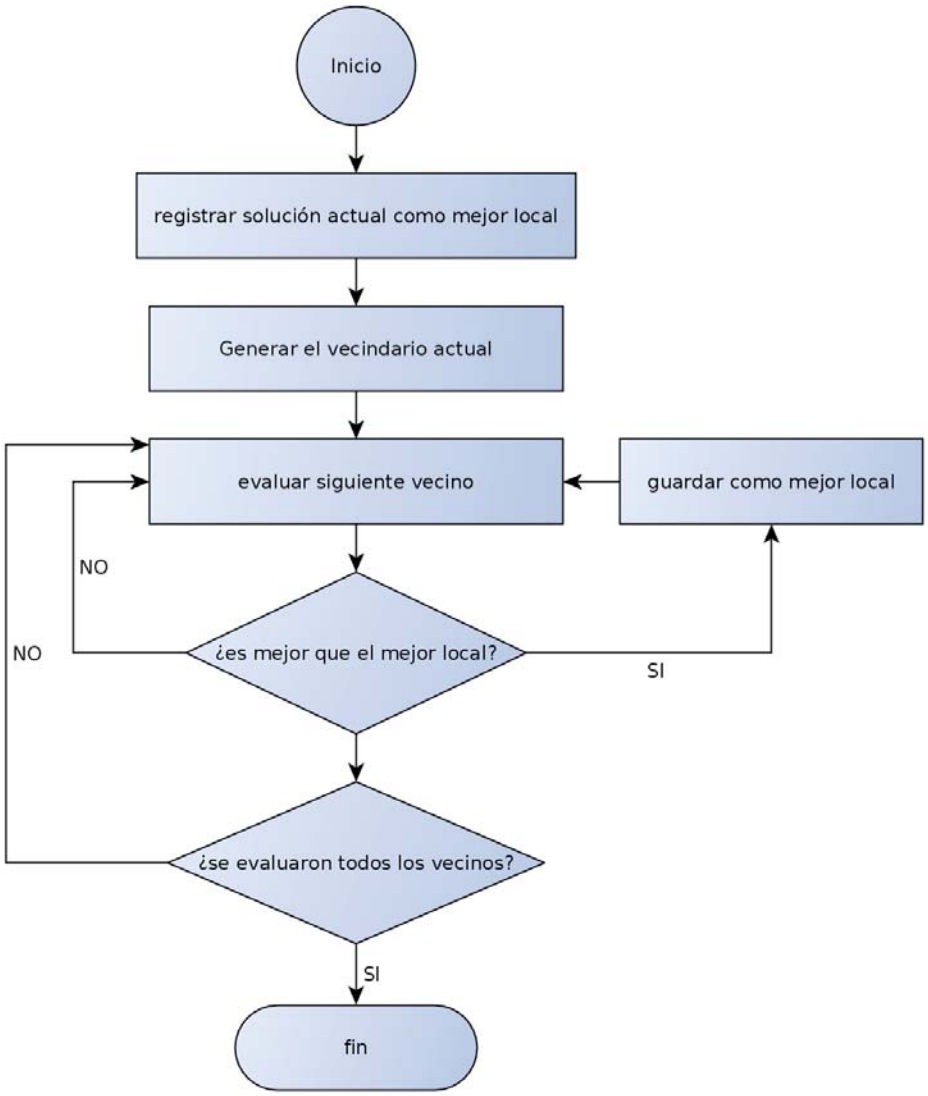

Figura 3: Procedimiento de búsqueda local

Método de búsqueda local:

Evaluación de costo de elementos vecinos a la solución actual, esta evaluación puede ser exhaustiva o selectiva, podemos ver su representación esquemática en la Figura 3

- **Paso 1** (Inicialización).
  - Seleccionar una solución o de arranque $\bar{x}$ Actual $\in$ X. y empezar con la lista de la historia H vacío.
- **Paso 2** (Elección y finalización).
  - Determinar Candidato N ($\bar{x}$ Actual) como un subconjunto de N (H, $\bar{x}$ Actual).
  - Seleccionar $\bar{x}$ Siguiente de Candidato N ($\bar{x}$ Actual) para minimizar c(H, $\bar{x}$) sobre este conjunto
    ($\bar{x}$ Siguiente es llamado elemento de evaluación o mayor de Candidato N ($\bar{x}$Actual)).
  - Terminar mediante un criterio de parada seleccionado.
- **Paso 3** (Actualización).
  - Ejecutar la actualización o por el método de búsqueda global, y actualizar la lista de la historia H.

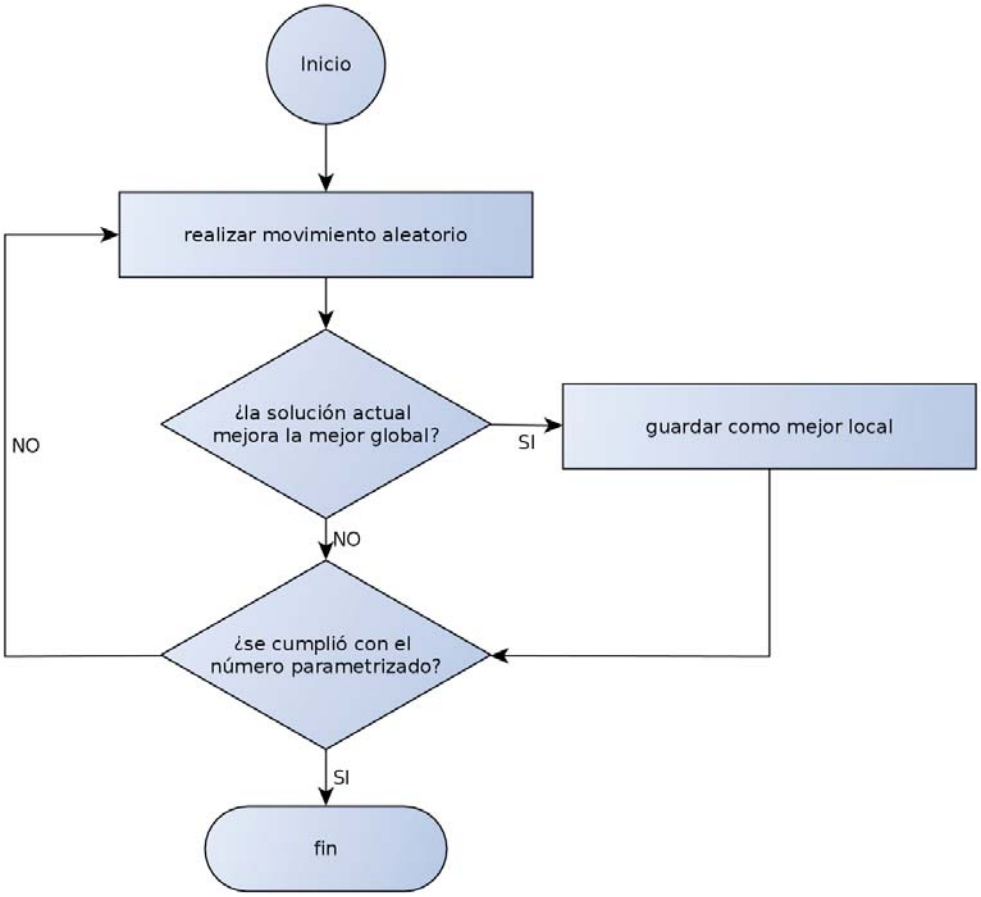

Figura 4: Procedimiento de búsqueda global

Método de búsqueda global:

En este paso se cambia de vecindad mediante una transformación N sobre la solución $\bar{x}$ actual. También se conoce como proceso de diversificación, porque permite salir de óptimos locales al aceptar costos de soluciónes mayores al mejor actual, podemos ver una versión esquemática en la Figura 4.

- **Paso 1** (Inicialización).
  - Seleccionar una solución o de arranque $\bar{x}$ Actual $\in$ X.
  - Almacenar la mejor solución o actual conocida haciendo $\bar{x}$ Mejor $= \bar{x}$ Actual y definiendo MejorCoste = c($\bar{x}$ Mejor).

- **Paso 2** (Elección o y finalización).
    - Elegir una solución o $\bar{x}$ Siguiente $\in$ N($\bar{x}$ Actual). Si los criterios de elección o empleados no pueden ser satisfechos por ningún u miembro de N ($\bar{x}$ Actual), o si se aplican otros criterios de parada, entonces el método para.
- **Paso 3** (Actualización).
    - Rehacer $\bar{x}$ Actual = $\bar{x}$ Siguiente, y si c($\bar{x}$ Actual) $<$ MejorCoste, ejecutar el paso 1
    - Volver al paso 2.

Elementos de la búsqueda tabú

Asociado a cada intercambio hay un valor de movimiento, que representa el cambio sobre un valor de la función objetivo como resultado del intercambio propuesto. Los valores de los movimientos generalmente proporcionan una base fundamental para evaluar la calidad de los mismos, aunque también pueden ser importantes otros criterios. Un mecanismo principal para explotar la memoria en la búsqueda tabú es clasificar un subconjunto de movimientos en un entorno como prohibidos (o tabú). La clasificación depende de la historia de la búsqueda, determinada mediante lo reciente o frecuente que ciertos movimientos o componentes de soluciones, llamados atributos, han participado en la generación de soluciones pasadas.

- Movimiento
- Vecindario
- Criterio de Aspiración

Tipos de memoria

Una característica distintiva de este procedimiento es el uso de memoria adaptativa. Las estructuras de memoria de la búsqueda tabú funcionan mediante referencia a cuatro dimensiones principales, consistentes en la propiedad de ser reciente, en frecuencia, en calidad, y en influencia.

El concepto de influencia, mide el grado de cambio inducido en la estructura de la solución o o factibilidad. (la influencia a menudo se asocia a la idea de distancias de movimiento, es decir, donde se concibe que un movimiento de mayor distancia tiene mayor influencia).

La calidad se refiere al relativo valor de las soluciones que se van encontrando, en el contexto del vecindario o globalmente.

Basada en lo reciente

Un mecanismo principal para explotar la memoria en la búsqueda tabú es clasificar un subconjunto de movimientos en un entorno como prohibidos (o tabú). La clasificación depende de la historia de la búsqueda, determinada mediante lo reciente o frecuente que ciertos movimientos o componentes de soluciones, llamados atributos, han participado en la generación de soluciones pasadas. Por ejemplo, un atributo de un movimiento es la identidad del par de elementos que cambian posiciones

Basada en frecuencia

Para nuestros propósitos presentes, concebimos medidas de frecuencia como proporciones, cuyos numeradores representan cuentas del número de ocurrencias de un evento particular (por ejemplo, el número de veces que un atributo particular pertenece a una solución movimiento) y cuyos denominadores generalmente representan uno de cuatro tipos de cantidades: (1) el número total de ocurrencias de u todos los eventos representados por los numeradores (tal como el número de iteraciones asociadas), (2) la suma de los numeradores, (3) el máximo valor del numerador, y (4) la media del a valor del numerador. Los denominadores (3) y (4) dan lugar a lo que se puede llamar frecuencias relativas. En los casos en los que los numeradores representan cuentas ponderadas, algunas de las cuales pueden ser negativas, los denominadores (3) y (4) se expresan como valores absolutos y el denominador (2) se expresa como una suma de valores absolutos.

3.5. Computación paralela

La computación paralela/distribuida aplicada a la solución de problemas significa que varios procesos trabajan simultáneamente en varios procesadores con el fin común de resolver una instancia de un problema determinada. Uno, entonces debe determinar cómo el proceso de resolución global es controlado y qué información se intercambia entre los distintos procesos. Más de un método de solución puede ser usado para realizar la misma tarea y entonces una taxonomía debe especificar esta característica [11].

3.5.1. Algoritmos paralelos

La mayoría de los algoritmos de hoy en día son secuenciales, esto es, especifican una secuencia de pasos en los que cada uno consiste de un sola operación. Estos algoritmos funcionan bien en los computadores tradicionalmente disponibles que basicamente realizan operaciones de una manera secuencial.

Aunque la velocidad en la que los computadores secuenciales ha venido incrementándose exponencialmente por muchos años, este incremento llegó a un punto de inflexión, en cuanto a la relación costo - ganancia. Como consecuencia, los investigadores han visto una oportunidad de mejora en la construcción de computadores «paralelos» a menores precios, computadores que realizan múltiples operaciones en un solo paso.

Para resolver un problema eficientemente en una máquina paralela, es generalmente necesario diseñar un algoritmo que especifique múltiples operaciones en cada paso, i.e. un algoritmo paralelo. El paralelismo en un algoritmo puede mejorar el desempeño en varios tipos de computadores. Por ejemplo, en un computador paralelo, las operaciones en un algoritmo paralelo se pueden llevar a cabo simultáneamente por diferentes procesadores. Más aun, incluso en un computador de un solo procesador, el paralelismo en un algoritmo puede ser usado por varias unidades funcionales, unidades funcionales en línea de secuencia (pipeline), o sistemas de memoria en línea de secuencia.

Entonces, se puede entender que es importante hacer una distinción entre el paralelismo de un algoritmo y la habilidad de tal o cual computador, para desempeñar múltiples operaciones en paralelo.

Desde luego, para lograr que un algoritmo paralelo funciones eficientemente en cualquier tipo de computador, el algoritmo debe al menos contener tanto paralelismo como el computador, ya que cualquier recurso extra quedaría sin uso. Desafortunadamente, no todas las máquinas paralelas podrán correr eficientemente todos los algoritmos paralelos, ya que este desempeño depende en gan medida de la arquitectura hardware, y de la forma que se aborda el paralelismo en el algoritmo.

3.5.2. Paralelización de metaheurísticas

Dentro de las metaheurísticas, los límites de lo que puede resolverse dentro de un tiempo computacional «razonable» aún está llegando a sus límites

muy rápido para muchas configuraciones de distintos problemas [12], al menos demasiado rápido para las crecientes necesidades de la investigación y la industria por igual.

Las heurísticas en general no garantizan optimalidad, mas aún, el desempeño generalmente depende del problema en particular, de sus características, y de las características de la instancia a resolver. En consecuencia, un problema mayor en el diseño de las metaheurísticas y en su calibración no está solamente en cómo construirlos para máximo desempeño, pero también para hacerlos robustos, en el sentido de poder ofrecer un nivel de desempeño consistentemente alto en una amplia variedad de configuraciones de problemas y características.

Las metaheurísticas paralelas apuntan a resolver ambos problemas. Por supuesto, el primer objetivo es resolver instancias grandes de problemas en un tiempo computacional razonable. Con características apropiadas, tales como las estrategias cooperativas multi-búsqueda, las metaheurísticas paralelas prueban también ser mucho más robustas que sus versiones secuenciales en tratar con diferencias en los tipos y características de los problemas. También requieren menos esfuerzos de calibración.

En este trabajo en particular se expone una descripción centrada en la búsqueda tabú, aunque la literatura en el campo de las metaheurísticas paralelas es basta, y es posible encontrar múltiples enfoques paralelos para las metaheurísticas más conocidas como [15] que aplica Tabu Search en una arquitectura GPU, [27] presenta un algoritmo ACO paralelizado, [29] demuestra un algoritmo genético con un diseño paralelo.

3.6. Paralelización de la búsqueda tabú

La metaheurística de búsqueda tabú cuenta con gran cantidad de trabajos que aplican distintos grados de paralelismo

3.6.1. Taxonomía de Paralelización y sus dimensiones

Cualquier estrategia de paralelización implica una descomposición del dominio o de los pasos y tareas básicos del algoritmo, o de ambos. En consecuencia, no toda la información está disponible necesariamente en cualquier momento durante la resolución paralela del una instancia del problema, y, debido a esto, la manera como se intercambia el conocimiento recolectado

Cardinalidad de Control	1-control p-control
Tipo de control y comunicación	Sincronización Rídigda Sincronización de Conocimiento Collegial Collegial con Conocimiento
Diferenciación de Búsqueda	SPSS SPDS MPSS MPDS

Cuadro 1: Dimensiones de Clasificación

durante la exploración paralela del dominio es intercambiado y combinado entre los procesos es tan importante como la forma en que el dominio es dividido, o como la tareas son distribuidas entre los distintos procesos.

La taxonomía aquí utilizada, y propuesta por [11], está construida de acuerdo a tres dimensiones que capturan los factores mencionandos anteriormente. Los primeros dos representan las formas de paralelización que dicen relación al control de la trayectoria de búsqueda y al enfoque de procesamiento y comunicación de información, mientras que el tercero se relaciona con las estrategias usadas para particionar el dominio y para especificar los parámetros para cada búsqueda. Las tres dimensiones están sumarizadas en el Cuadro 1

Cardinalidad de control de búsqueda

El control de la búsqueda tabú puede residir en un solo procesador, generalmente llamado maestro o procesador principal, o distribuído entre varios procesadores. Se pueden definir dos categorías. En el primer caso llamado 1-control, trivialmente corresponde al caso secuencial. En un contexto paralelo, representa el enfoque en el cual un procesador esencialmente ejecuta el algoritmo, pero delega algo de trabajo a otros procesadores. El proceso maestro recolecta y reincorpora la información, distribuye las tareas a ser ejecutadas por los otros procesadores, y determina cuando la búsqueda debe detenerse. Las tareas que son delegadas pueden consistir solamente de cálculos numéricos, de la exploración paralela del vecindario, o de la construcción y evaluación de una lista de candidatos.

En el segundo caso, el control de la búsqueda está compartido entre $p, p > 1$, procesadores, por lo tanto se lo puede identificar como p-control, el arreglo clásico de procesos multihilos o *Collegial* cae en esta categoría. Cada proceso está a cargo de su propia búsqueda, tanto como de establecer comunicaciones con los otros procesos. La búsqueda global termina una vez que cada búsqueda individual termina. La coordinación de los intercambios de información y los intentos de asegurarse de que información adecuada esté disponible cuando sea requerida están entre los problemas principales en este contexto, y juegan también un rol importante en la definición del tipo de control que es ejercido.

Tipo de control y comunicación

La segunda dimensión de la taxonomía está basada en el tipo y ejercicio del control : toma en cuenta la organización de la comunicación, sincronización y jerarquía, también como la manera en que la información es procesada y compartida entre procesos. La dimensión de tipo de control está construida por hasta cuatro etapas o grados, que combinan los dos niveles de cardinalidad de control que definen las estrategias de paralelización relativas al manejos de procesos e información. El primer grado corresponde a la sincronización rígida de procesos. Un modo de operación síncrono usualmente indica que todos los procesos deben detenerse y realizar alguna forma de comunicación e intercambio de información en algunos puntos de su proceso (número de interaciones, intervalos de tiempo, etapas del algoritmo específicas, etc.) determinados exógenamente: ya sea codificados dentro de los procedimientos o determinado por un proceso de control.

Clasificación

Se califican estas organizaciones como rígidas, cuando existe un intercambio muy limitado de información entre los procesos que están dedicados a la ejecución del mismo nivel de tareas. En particular, la sincronización rígida idealmente complementa el enfoque de 1-control. Este es el caso clásico de maestro-esclavo, donde el maestro ejecuta lo que se acerca a una búsqueda tabú secuencial usando otros procesadores para la ejecución de tareas de procesamiento intensivo. No existe comunicación entre los procesos esclavos, y la información se mantiene y maneja exclusivamente por el maestro, que

también inicia todas las fases de comunicación. La extención al p-control es el caso de la estrategia de paralelización donde búsquedas independientes son realizadas simultáneamente.

Cada búsqueda puede empezar por una solución inicial distinta, o puede estar usando un juego de parámetros distinto o ambos. En síntesis, no existe comunicación entre los procesos durante la búsqueda y cada proceso termina cuando su propio criterio de término es cumplido. La mejor solución es seleccionada una vez que todos los procesos se han detenido. La siguiente etapa está caracterizada también por un método de operar síncrono, per un nivel de comunicación mayor permite construir un intercambio de conocimiento. De ahí que, se puede identificar como sincronización de conocimiento. Cuando se opera dentro del marco de 1-control, el maestro continua manejando la información, para sincronizar los procesos, y para despachar trabajo a los esclavos, pero delega una gran parte del trabajo. Los procesos esclavos aún no se comunican entre ellos.

Tareas de los procesos

Sus tareas, sin embargo, son más complejas que en la sincronización rígida, y podrían implicar que estructuras de memoria local están presentes. Por ejemplo, un proceso esclavo puede ejecutar una secuencia limitada de pasos de la búsqueda tabú en un subconjunto del vecindario (e.g. la intensificación de los candidatos prometedores). Pero a petición del maestro (cuando se sincroniza, por ejemplo) el proceso esclavo retorna el problema y los resultados, y espera una nueva tarea. Cuando se adopta una estrategia de p-control, la sincronización de conocimiento corresponde a varias trayectorias de búsqueda que terminan todas en un momento determinado (e.g. el número de iteraciones), el mismo para todos los procesos. En ese momento la fase de comunicación intensiva empieza entre todos los procesos de control. Esto podría ser visto como un acercamiento híbrido entre la sincronización rígida y el collegial independiente.

Para sumarizar, en modo síncrono la estrategia de 1-control implica canales de comunicación maestro-esclavo verticales exclusivamente, mientras que solamente existe comunicación orizontal entre procesos procesos en una estrategia p-control. La diferencia entre sincronización rígida y traspaso de conocimiento no está siempre a la vista en el contexto de 1-control, ya que está mayormente basado en cuánto trabajo asigna el maestro a cada escla-

vo. Esta diferencia es mucho más significativa para las estrategias p-control, ya que corresponde a la ausencia o presencia de comunicación inter-proceso e intercambios de conocimiento. El tercero y cuarto grados de dimensiones de estrategias de control hace uso de modos de comunicación síncrona. En este contexto, cada proceso guarda y trata su propia información, iniciando comunicación con alguno o todos los demás procesos de acuerdo su lógica interna y status.

Se definen los dos grados de acuerdo a la cantidad, calidad y tratamiento de la información intercambiada. De acuerdo al rol que la comunicación toma parte en reconstruir una patrón de búsqueda global cuando varios hilos independientes de búsqueda exploran el dominio de búsquda. En la tercera etapa, que se llama Collegial, cada proceso ejecuta una búsqueda tabú eventualmente distinta en todo o parte del dominio.

Comunicación entre procesos

Cuando un proceso necesita una solución mejor (local o globalmente, de acuerdo a la estrategia elegida), lo transmite (junto, posiblemente con su contexto e historia) a todos o algunos de los procesos de búsqueda. Puede también depositarlo en una memoria central, y solamente transmitir (si es que) la mejor solución que se ha encontrado. En todos los casos, las comunicaciones son simples, en sentido de que los mensajes enviados corresponden al mensaje recibido, este no es necesariamente el caso, en la cuarta etapa de Collegial con conocimiento, aquí los contenidos de las comunicaciones son analizados para inferir información adicional con respecto a las trayectoria de búsqueda global y a las características globales de buenas soluciones.

Las memorias globales (e.g. la frecuencia del cambio de estado de algunas variables) y las listas tabú que reflejan la dinámica de la exploración síncrona paralela del dominio, pueden ser entonces construidas, mientras nuevas soluciones pueden ser construidas en base a las soluciones y los contenidos en memoria enviados por las búsquedas individuales. De esta manera el mensaje recibido por un proceso es generalmente más completo, que y distinto al mensaje enviado inicialmente por otro proceso.

Estrategia de diferenciación de búsqueda

En la clasificación propuesta por Voß [34], el único criterio considerado para referirse al número de estrategias de soluciones iniciales distintas (ajuste de parámetros, políticas de manejo de la lista tabú, etc.) usadas por una implementación en particular. Esto corresponde a la tercera dimensión anteriormente descrita, la que se identifica como estrategia de diferenciación de búsqueda. Se usa el término «Estrategia de búsqueda» en el sentido más general que include las distintas definiciones de vecindarios, ajuste de parámetros, reglas de manejo de memoria, formas de diversificación, etc.

Se puede identificar los siguientes cuatro casos: SPSS, (mismo punto/población inicial, misma estrategia de búsqueda, Same initial point/population, Same search strategy); SPDS (mismo punto/población inicial, diferentes estrategias de búsqueda, same initial point/population, different search strategies); MPSS, (puntos/poblaciones iniciales distintos, misma estrategia de búsqueda); MPDS (puntos/poblaciones iniciales distintos, distinta estrategia de búsqueda).

3.6.2. Estrategias de paralelización

TS es particularmente propicio para ser implementado de forma paralela, y el desempeño es generalmente escalable mediante el uso de varios procesadores, el tiempo real (de reloj) puede ser

substancialmente reducido comparado con un programa secuencial equivalente. Hay varias estrategias disponibles y éstas son descritas en [14, 13]. Combinando estrategias de paralelización puede ser efectivo, como es demostrado en [16]. Algunas de las más comunes pueden ser sumarizadas como :

1. Evaluación paralela de vecinos: Este método es una implementación del modelo maestro-esclavo y funciona con la premisa que la parte más caras computacionalmente hablando de la búsqueda tabú es la evaluación de los vecinos (correspondiente a al paralelismo de granularidad gruesa). En esta aproximación, el vecindario es dividido en partes iguales entre los procesadores. Cada procesador desempeña la transición junto con la evaluación de la función de costo y las restricciones y envía su mejor vecino de vuelta al procesador maestro. El procesador maestro determina el mejor vecino entre todos los que recibió (siguiendo las reglas de la lista tabú) y anuncia el vecindario para que cada esclavo pueda actualizar su copia local de la solución. Esta aproximación

puede requerir un tiempo considerable de comunicación debido a la dependencia maestro-esclavo y la naturaleza iterativa del algoritmo, pero puede ser fácilmente aplicado a un rango de COP's. ver [22, 13] como ejemplos de la implementación de este método.

2. Búsquedas paralelas independientes: En esta aproximación, un número de búsquedas tabú secuenciales son ejecutadas simultáneamente en todos los procesadores para un problema en particular. Cada búsqueda es distinta, ya que parámetros clave tales como la semilla aleatoria, la solución inicial o el largo de la lista tabú son cambiados. Este método es particularmente apto para arquitecturas paralelas en las cuales cada nodo se comporta como un sistema independiente, tales como los procesadores multi núcleo actuales. Debido a esta independencia de las búsquedas, no se requiere comunicación entre los procesadores.

3. Interacción de Búsquedas Tabú paralelas: Este enfoque es similar al método 2, excepto que a intervalos dados en la búsqueda, ocurre una interacción entre las búsquedas. Esto consiste en determinar qué búsqueda ha sido la más exitosa y transferir esa solución a los otros procesos de búsqueda. Cada búsqueda entonces continua con una lista tabú vacía. Esta aproximación puede tener una sobrecarga de comunicación bastante grande debido a la necesidad de comunicar estructuras de solución enteras.

4. División del espacio de búsqueda: A cada procesador se le asigna una parte del espacio de búsqueda. Una búsqueda Tabú explora entonces su subsección y envía de vuelta la solución parcial al proceso maestro luego de haber terminado [13]. Estas soluciones parciales son combinadas en una solución final. Mientras que este método tiene pocos costos de comunicación, el proceso de dividir el espacio de búsqueda es bastante específico al problema y podría no ser posible para todos los problemas.

Según la clasificación propuesta por [12], las estrategias paralelas que explotan el potencial de la descomposición de tareas dentro de los ciclos internos de las metaheurísticas son llamados de «bajo nivel», ya que no modifican ni la lógica del algoritmo ni el espacio de búsqueda. Éstos apuntan solamente a acelerar la búsqueda y generalmente no modifican el comportamiento de una metaheurística secuencial. Típicamente, la exploración se inicializa desde una sola solución inicial o población, y la búsqueda procede de acuerdo a una sola estrategia de búsqueda, solamente los cálculos de los ciclos internos son descompuestos y simultáneamente ejecutados por varios

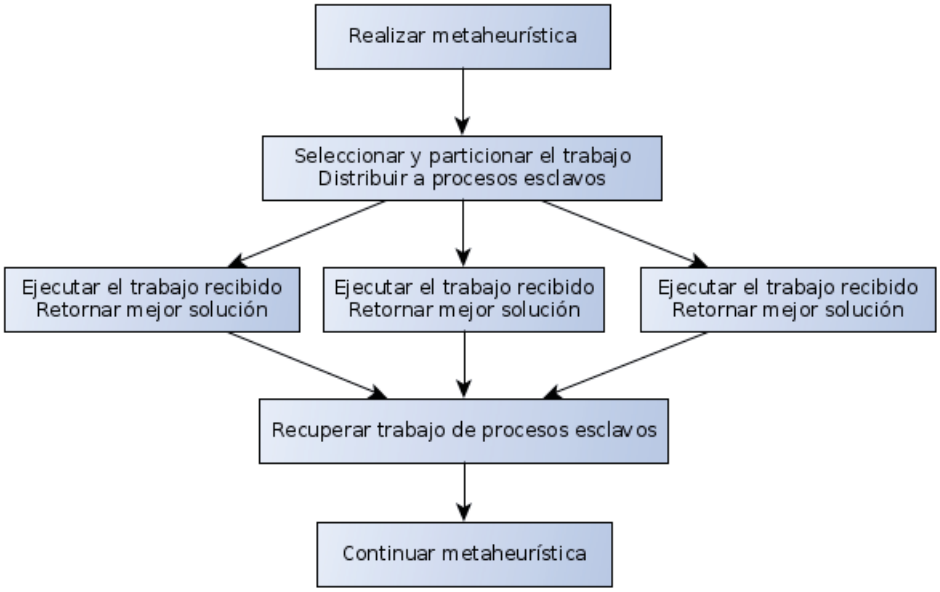

Figura 5: Estrategia de descomposición de bajo nivel

procesadores.

En este caso en particular, la estrategia elegida es la de evaluación paralela de vecinos, que a su vez se puede clasificar como un estrategia de descomposición de bajo nivel, y que además resulta ser la más intuitiva y simple de implementar en una arquitectura SIMD tal como son las GPU's, debido a que realiza la misma operación con distintos datos que son cada vecino en particular.

3.7. Arquitecturas hardware paralelas

Las operaciones concurrentes o paralelas toman muchas formas dentro de un sistema de cómputo [20]. Usando un modelo basado en los distintos flujos usados en el proceso computacional, representamos algunos de los distintos tipos de paralelización disponible. Un flujo es una secuencia de objetos tales como datos, o acciones tales como instrucciones. Cada flujo es independiente de los demás, y cada elemento del flujo puede consistir en uno o más objetos o acciones. Entonces tenemos cuatro combinaciones que describen las arquitecturas paralelas más familiares :

1. SISD : single instruction, single data stream (instrucción simple, flujo de datos simple). Es el uni-procesador tradicional

2. SIMD: single instruction, multiple data stream (instrucción simple, flujo de datos múltiple). Incluye procesadores vectoriales tanto como procesadores masivamente paralelos.

3. MISD: multiple instruction, single data stream (instrucción múltiple, flujo de datos simple). Típicamente encontrado en arreglos sistólicos (systolic arrays), en donde un conjunto de procesadores dispuestos en forma de matriz procesan datos y los guardan de manera independiente.

4. MIMD: multiple instruction, multiple data stream (instrucción múltiple, fujo de datos múltiple). Incluye multiprocesadores tradicionales, tanto como las redes de nodos interconectados.

Cada una de estas combinaciones caracteriza una clase de arquitecturas y sus correspondientes tipos de paralelismo, esto se encuentra resumido en la Figura 6.

3.7.1. GPGPU

Las Unidades Procesadoras de Gráficos (GPU's) de hoy en día están compuestas de cientos o miles de unidades procesadoras [31] corriendo a frecuencias bajas o moderadas y están diseñadas para realizar grandes cantidades de cálculos de punto flotante y que a la vez son poco sensibles a la latencia. Para esconder la latencia de la memoria global, las GPU's contienen relativamente pequeños caches en-chip que hacen uso extensivo de la capacidad multi-hilo del hardware, ejecutando decenas de miles de hilos concurrentemente a través del conjunto de unidades de proceso.

Arquitectura

Las unidades de proceso GPU están típicamente organizadas en agrupamientos SIMD (single instruction, multiple data) y son controlados por un decodificador de instrucciones único con acceso compartido a caches en chip muy rápidos y también a memorias compartidas. Los agrupamientos SIMD ejecutan instrucciones en un ciclo, la divergencia de ramas es manejada ejecutando ambos caminos y enmascarano los resultados de las unidades de proceso inactivas según sea necesario. El uso de arquitecturas SIMD y

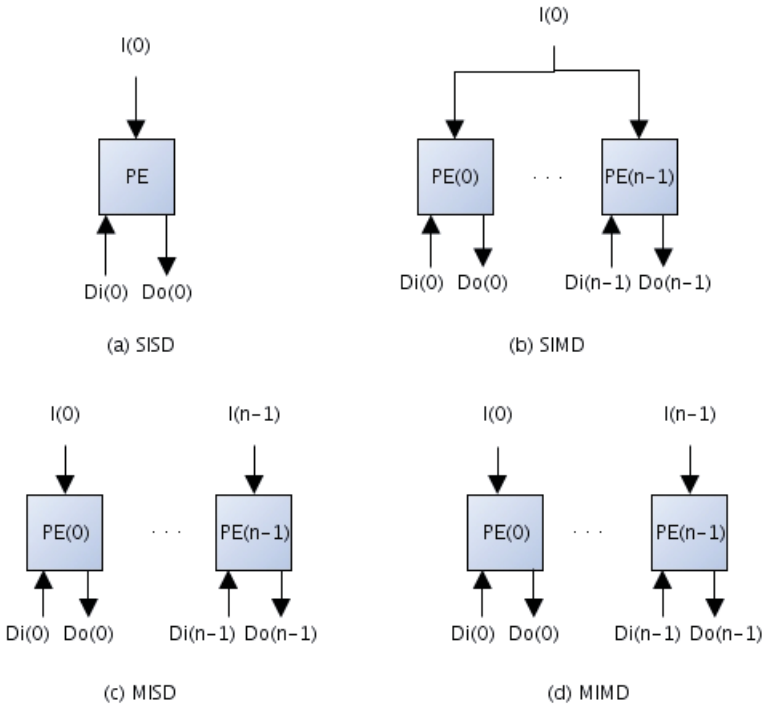

Figura 6: El modelo de flujos en multiprocesadores

ejecución de instrucciones en-orden permite a las GPU contener muchas más unidades aritméticas en la misma área que las CPU tradicionales. AMD y NVidia han lanzado implementaciones de OpenCL que soportan una arquitectua de procesadores escalable para PE (unidades de proceso) individuales expuestas por OpenCL, haciéndolos altamente eficientes en la mayoría de los tipos de datos OpenCL.

Las GPU de AMD usan una arquitectura vectorial, y típicamente alcanzan mejor desempeño cuando los Work-Items de OpenCL operan con tipos de datos vectoriales de 4 elementos (como float4).

En muchos casos un kernel OpenCL vectorizado puede desempeñarse bien en CPU's x86 y en GPU's AMD y NVidia, pero el kernel resultante podría resultar ser menos legible que su equivalente escalar.Las diferencias en la arquitectura de bajo nivel de las GPU incluyen variaciones en qué memoria se pasa a cache y qué patrones de acceso a memoria crean conflictos de

bancos, que afectan la optimalidad del kernel. La literatura de optimización provista por cada fabricante generalmente contiene guías de optimización de bajo nivel.

NVidia y CUDA

La introducción de la Compute Unified Device Arquitecture (CUDA) por parte de NVidia, causó un incremento rápido en el interés de la comunidad científica y la industria, la computación de propósito general usando hardware gráfico (GPGPU). Dos factores contribuyeron en mayor medida al éxito de CUDA.

Primero, es que durante los últimos años, la capacidad computacional de las GPU, está creciendo mucho mas rápido que las capacidades de las CPU. Esto fue mayormente debido al gran incremento de los requierimientos de hardware de los juegos modernos. Luego NVidia liberó una linea completa de GPU's llamadas Tesla, diseñadas específicamente para computación de alto desempeño, en vez de para juegos. El hardware GPU programable estuvo disponible incluso antes de CUDA. GPGPU empezó con unidades vectoriales programables (shaders de vértices y pixeles), aunque las posibilidades eran bastante limitadas, debido al pequeño conjunto de instrucciones y a la extención limitada del código.

Como resultado el desarrollo GPGPU era muy complicado y los beneficios no eran impresionantes, esto resultó en un interés relativamente bajo en GPGPU. Por otro lado CUDA llega en una forma de extensiones a los lenguajes C/C++, permitiendo una programación simplificada en GPU's al esconder todos los detalles relacionados al hardware GPU. Por supuesto, para maximizar el desempeño de las aplicaciones CUDA, todavía es necesario gran conocimiento de las características del hardware, aunque muchas optimizaciones son hechas en forma automática.

La investigación más reciente está enfocada en cómo utilizar las enormes capacidades computacionales de las GPU modernas para acelerar la ejecución de problemas que demandan gran capacidad de proceso. Muchos algoritmos existentes, e incluso librerías son paralelizadas e implementadas para correr en GPU, resultando en un incremento substancial en su desempeño. Relativamente poca investigación se ha hecho hasta ahora en la implementación con CUDA de meta heurísticas de optimización discreta. Algunos resultados en esta área fueron publicados por e.g. Janiak et al. [12], quienes proponen una implementación GPU de tabu search para el problema del vendedor via-

jero (TSP).

3.7.2. OpenCL

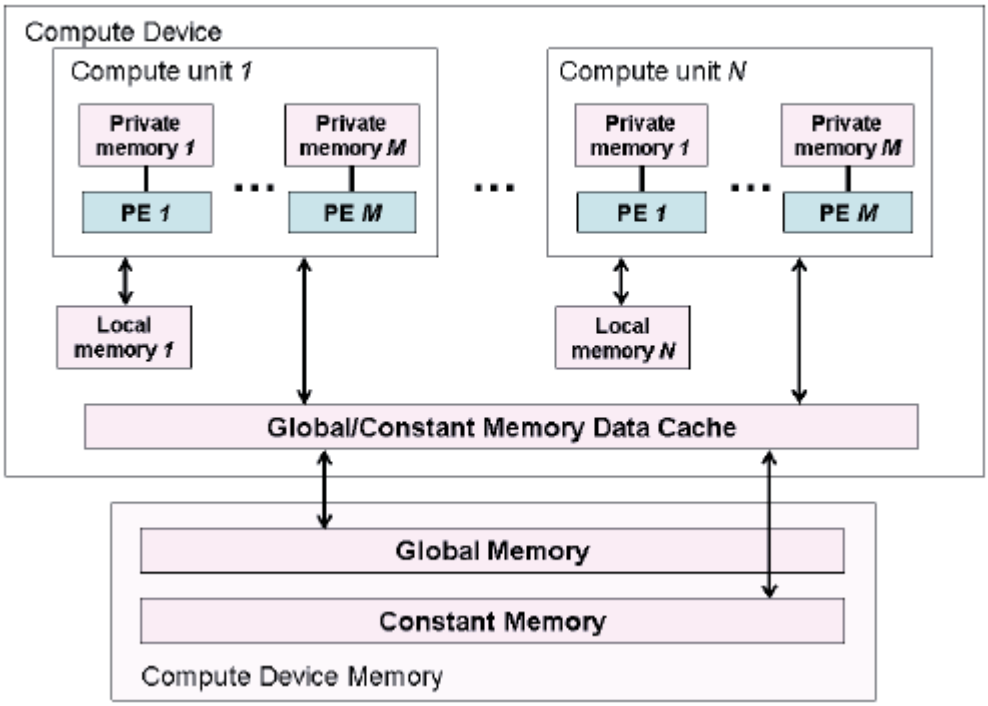

Figura 7: Arquitectura de dispositivo OpenCL conceptual, con elmentos de proceso (PE), unidades de cómputo y dispositivos. No se muestra el Host

Heterogeneidad del hardware

La fuerte necesidad de incrementar la capacidad computacional en ciencia e ingeniería ha llevado al uso de computación heterogénea, con CPU's y otros aceleradores actuando como coprocesadores para actividad intensiva en computación aritmética. OpenCL es un nuevo estándar de la industria para computación heterogénea de tarea-paralela y data-paralela en una variedad de CPU's GPU's DSP's y otros tipos de diseños de microprocesadores modernos.

Esta tendencia a la computación heterogénea y a las arquitecturas altamente paralelas ha creado una fuerte nececidad de infraestructura de desarrollo de software en la forma de programación paralela, lenguajes y librerías de subrutinas que soporten computación heterogénea en hardware de distintos fabricantes. Para cumplir con esto, los desarrolladores adaptaron varias aplicaciones de ciencia e ingeniería existentes para aprovechar las CPU multi núcleo y las GPU masivamente paralelas usando toolkits como los Threading Building Blocks (TBB), OpenMP, y CUDA. Los toolkits de programación existentes, sin embargo, erean ya sea limitados a una sola familia de microprocesadores o no soportaban la computación heterogénea. OpenCL [1, 31] provee abstracciones fáciles de usar y una amplia gama de APIs de programación basadas en éxitos anteriores con CUDA y el toolkit AIP.

OpenCL define la funcionalidad principal que todos los dispositivos soportan, además de funcionalidad opcional para dispositivos de más alta capacidad; también incluye un mecanismo de extensión que permite a los fabricantes exponer características de hardware únicas e interfaces de programación experimentales para el beneficio de desarrolladores de aplicaciones. Aunque OpenCL no puede enmascarar diferencias significativas en la arquitectura del hardware, si garantiza portabilidad y correctitud. Esto hace mucho más fácil para los desarrolladores partir con un programa OpenCL ajustado para una arquitectura y producir un programa que funcione correctamente en otra arquitectura.

El modelo de programación de OpenCL

En OpenCL, un programa es ejecutado en un dispositivo computacional que puede ser CPU, GPU, u otro acelerador. Los dispositivos contienen una o más unidades de cómputo (cores de procesador). Estas unidades están a su vez compuestas por uno o más elementos de proceso (PE) de una instrucción y data múltiple (SIMD) que ejecutan instrucciones en un ciclo. Estas características están representadas en la Figura 8.

Al proveer un lenguaje e interfaz de programación común y abstracciones de hardware, OpenCL ha llevado a los desarrolladores a acelerar aplicaciones con computaciones de tarea paralela o data paralela en ambientes de computación heterogéneos consistentes de la CPU del host y cualquier número de dispositivos OpenCL acoplados. Tales dispositivos pueden o no compartir memoria con la CPU del host, y tipicamente tienen un conjunto de instrucciones máquina distintas. Las interfaces de programación de OpenCL asumen

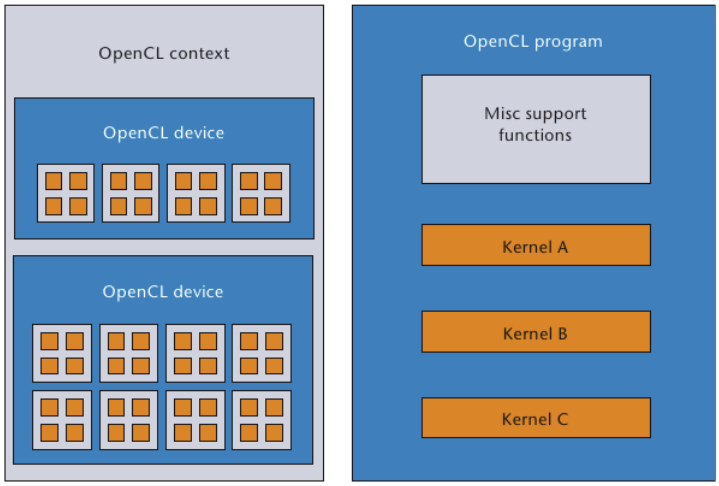

Figura 8: Jerarquía de componentes en arquitectura OpenCL

heterogeneidad entre el host y sus dispositivos acoplados.

Las intefaces de programación claves de OpenCL proveen funcioes para

- enumerar los dispositivos disponibles en el host (CPU, GPU y acelaradores varios);
- administrar los contextos de los dispositivos del objetivo;
- administrar las reservas de memoria;
- realizar transferencias entre memoria del host y dispositivos;
- compilar los programas OpenCL y las funciones de kernel que esos dispositivos van a ejecutar;
- lanzar kernels en los dispositivos objetivo;
- consultar el progreso de la ejecución;
- verificar errores.

4. Solución propuesta

4.1. Modelamiento del problema

Se usa la nomenclatura presentada en [18], que cuenta con los siguientes elementos :

- M el número de máqinas
- P, el número de partes
- C, el número de celdas
- i, el índice de la máquina (i=1,...,M),
- j, el índice de la parte (j=1,...,P),
- k, el índice de la celda (k=1,...,C),
- $A = [a_{ij}]$, la matriz de incidencia binaria máquina-parte M x P,
- M_{max}, el número máximo de máquinas por celda.
- $y_{ik} = \begin{cases} 1 & si\, la\, máquina\, i \in celda\, k; \\ 0 & de\, otra\, manera; \end{cases}$
- $z_{jk} = \begin{cases} 1 & si\, la\, parte\, j \in familia\, k; \\ 0 & de\, otra\, manera; \end{cases}$

Minimizar :

$$\sum_{k=1}^{C}\sum_{i=1}^{M}\sum_{j=1}^{P} a_{ij} z_{jk}(1 - y_{ik}) \tag{1}$$

Sujeto a :

$$\sum_{k=1}^{C} y_{ik} = 1 \quad \forall i \tag{2}$$

$$\sum_{k=1}^{C} z_{jk} = 1 \quad \forall j, \tag{3}$$

$$\sum_{i=1}^{M} y_{ik} \leq M_{max} \quad \forall k \tag{4}$$

4.2. Elementos del algoritmo

La búsqueda tabú requiere que se definan ciertos elementos para poder operar, particularmente para este diseño se han definido los siguiente componentes:

- Movimiento : reordenamiento de una fila o columna en la matriz de incidencia.
- Vecino : solución que dista un movimiento de otra.
- Vecindario : todos los vecinos de una solución dada.
- Se penaliza costo de soluciones infactibles

4.3. Algoritmo propuesto

El algoritmo propuesto cuenta con una parte secuencial, y otra parte que es ejecutada en paralelo. La parte secuencial es el proceso maestro que se encarga de distribuír el trabajo de los procesos paralelos que actúan como esclavos, y luego recolectar e implementar sus resultados.

Proceso maestro secuencial

1. Entradas :
   a) Matriz de incidencia inicial (M x P)
2. Parámetros :
   a) Numero máximo de iteraciones
   b) C: cantidad de celdas
   c) Mmax: cantidad máxima de máquinas por celda
   d) d : cantidad de permutaciones aleatorias que sufrirá la solución actual en la etapa de diversificación.

e) t : cantidad de turnos en los que una determinada solución permanecerá en la lista tabú.

- Inicialización
  - Inicialización de variables
  - Se obtiene una solución inicial aleatoria
  - Se guarda la solución obtenida como mejor global
- Mientras (no se cumpla con el número de iteraciones especificado)
  - Búsqueda Local
    - Enumerar vecindario : generar todas las soluciones que componen el vecindario
    - Cada solución vecina es evaluada por una función de costos paralela (proceso esclavo)
    - Se compara con mejor global (aplica criterio de aspiración)
    - Se comparan los costos obtenidos entre los procesos esclavos
    - Se elige la mejor del vecindario que no sea tabú
    - Se agrega la solución elegida a la lista tabú
  - Diversificación
    - Se usa la mejor solución encontrada en la iteración anterior
    - Se permutan d elementos al azar
    - Se evalúa la solución obtenida
    - Se continúa con otra etapa de búsqueda local
- Fin Mientras
- Se Comunica la mejor solución encontrada
- Fin.

Adicionalmente se ha optado por implementar una versión secuencial del algoritmo de cálculo de costos a manera de referencia, la que permitirá realizar comparaciones con la versión paralela para juzgar el desempeño relativo de cada una.

4.4. Implementación

Para implementar el algoritmo de Búsqueda Tabú en OpenCL se debe tener en cuenta que la paralelización de la carga de trabajo no se realiza como en los sistemas multiprocesadores estándar, donde existe un número de procesadores idénticos en cuanto a capacidad. En OpenCL, y particuarmente en GPU existen unidades de proceso, que en comparación con la CPU son muy limitadas en cuanto a la velocidad con que ejecutan operaciones de control y memoria, pero debido a su gran número, y a que están diseñadas para ser eficientes en operaciones aritmetico-lógicas pueden lograr un número muy superior de operaciones por ciclo en comparación a un sistema multiprocesador estándar usando la misma cantidad de transistores.

4.4.1. Representación

Matriz de Incidencia

La matriz de incidencia será representada por una matriz de enteros sin signo, a pesar de que la matriz de incidencia es una matriz binaria. Dado que las GPU's actuales funcionan nativamente con operandos de 32 bits (en algunos casos soportan 64 bits también), mayor volumen de proceso puede ser logrado representando todos los valores como enteros de 32 bits, es decir, de tipo entero.

Solución

La solución entregada por el algoritmo corresponde a un vector de enteros en donde cada índice representa cada máquina en la matriz de incidencia (i : 1,...,M) y el valor de cada elemento representa el índice de la celda (k) a la que pertenece esta máquina (valores entre 1 y C), de ésta manera se asegura de que la misma máquina no esté asignada a más de una celda.
A continuación la clase que representa la solución en la implementación realizada.

```
class Solution {
public:
  Solution(unsigned int n_machines);
  virtual ~Solution();
  // vector solución
  int *cell_vector;
  // inicialización de la solución
  void init();
  // movimiento de intercambio
  int exchange(unsigned int i,unsigned int j);
  // clonar soluciónsu misma naturaleza
  Solution *clone();
private:
  // número de máquinas
  unsigned int n_machines;
};
```

Estructura tabú

La estructura tabú está representada por un vector de estructuras *TabuList* (en la implementación C++) en donde cada objeto guarda el número de turnos tabú restantes, cuyo valor incial corresponde al número de turnos tabú máximos, y una copia de la solución que se guarda en ese elemento de la lista tabú.
A continuación la implementación de la estructura tabú.

```
typedef struct tabu_item {
  Solution *solution;
  int turns;
} Tabu_item;

std::list<Tabu_item> lista_tabu;
```

Particionamientio del vecindario entre los procesos esclavos

En una primera instancia el proceso maestro (corriendo en CPU, secuencialmente) será quién genere matrices de incidencia para cada elemento del vecindario, para luego repartir particiones de éstos a cada proceso esclavo, para optimizar esto bajo la arquitectura OpenCL se deberá tener en cuenta el tipo de paralelismo que maneje el hardware subyacente así como la canti-

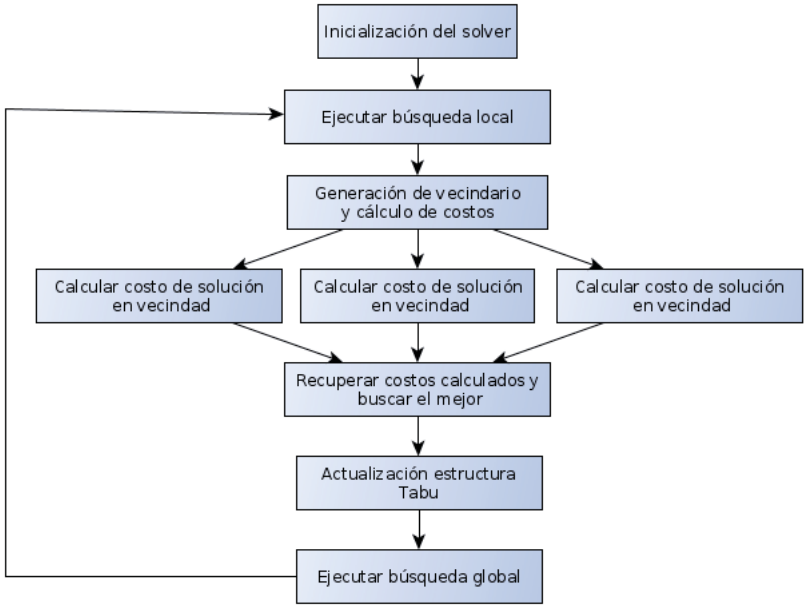

Figura 9: Vista de alto nivel de la implementación

dad de memoria local que posea cada grupo de ejecución, para así minimizar los accesos a memoria global.

Tabu search para MCD en OpenCL

Los detalles de la implementación del algoritmo propuesto en la arquitectura OpenCL propuesta requieren de una parte de programación C/C++ estándar sobre el procesador Host (secuencial), la que contendrá el proceso maestro descrito en la sección anterior, este proceso maestro se encarga de coordinar a los procesos esclavos, destinándoles trabajo (una parte del vecindario a evaluar) y luego coordinando y comparando los resultados entregados por cada esclavo. La parte de los procesos esclavos estará implementada como Kernels de OpenCL, optimizados para correr sobre una arquitectura SIMD; ya sea sobre GPU (caso ideal) o sobre CPU (una o varias) que cuenten con extensiones SIMD (e.g. SSE3).

Cada iteración del algoritmo de Búsqueda Tabú consiste en cuatro partes principales: generación del vecindario, evaluación de vecinos, elegir la mejor solución dentro del vecindario y actualizar la lista tabú, y finalmente moverse

a la nueva solución. Este flujo puede verse de una manera simplificada en la Figura 9.

Debido a la latencia de llamar al API de OpenCL para realizar tareas en el dispositivo se ha optado por mover la mayor cantidad de funcionalidad posible dentro de éste. Se definieron 4 kernels, que pueden ser vistos como una funciones que se ejecutan en el dispositivo dentro del contexto de OpenCL, los kernels son descritos a continuación :

- kernel búsqueda local : genera todas las soluciones del vecindario.
- kernel cálculo de costos : calcula el costo real de cada solución del vecindario.
- kernel cálculo de penalizaciones : calcula la penalización de costo para soluciones infactibles.
- kernel mínimo costo : elige la solución con el menor costo del vecindario.

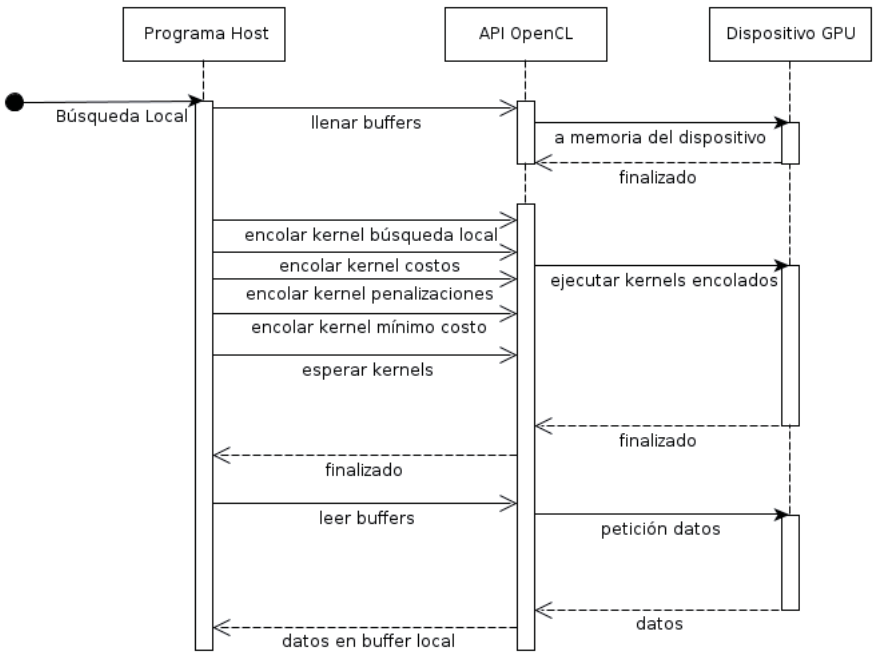

Figura 10: Secuencia de búsqueda local en OpenCL

En la Figura 10 se muestra la secuencia de ejecución que tiene la implementación al pasar por el API OpenCL y luego al dispositivo GPU, cabe destacar las llamadas asíncronas realizadas contra el API y la mayor latencia que se genera al tener que esperar la finalización de las tareas paralelizadas y luego la lectura de resultados por medio de copia de buffers.

4.5. Benchmarks

Se ha optado por utilizar los resultados publicados por Boctor [7] como referencia para realizar las pruebas de efectividad del algoritmo propuesto. Utilizando los 10 problemas propuestos, ya que existen los los óptimos globales y los resultados de una metaheurística (Simulated Annealing), lo que provee datos suficientes para realizar una comparación efectiva del método Tabu Search paralelo aquí descrito.

4.5.1. Resultados obtenidos

A continuación se presenta una tabla con los dos conjuntos de parámetros usados

Parámetro	Conjunto 1	Conjunto 2
Número de celdas (C)	2	3
Máximo de máquinas por celda (M)	8,9,10,11,12	6,7,8,9
Iteraciones máximas	5000	5000
Turnos tabú	100	100
Parámetro de diversificación	2	2

Cuadro 2: Parámetros usados por el solver en las pruebas realizadas

La motivación para optar por este juego de parámetros han sido pruebas preliminares que en general han demostrado un estancamiento del solver cuando se ha iterado sobre aproximadamente 500 veces, además de mayor efectividad al usar parámetros de diversificación pequeños (i.e. < 5), esto significa que en cada etapa de diversificación se cambian aleatoriamente un número de elementos (asignación de celdas a máquinas) según este parámetro. Esto tiene un papel fundamental en la política de movimiento por distintos vecindarios.

A continuación se presenta una tabla con los resultados obtenidos a partir de 30 ejecuciones del algoritmo para las instancias de los problemas 1 al 10 utilizando los parámetros de 2.

	C=2					C=3			
Instancia	M_{max}=8	9	10	11	12	M_{max}=6	7	8	9
1	11	11	11	11	11	27	18	11	11
2	7	6	4	3	3	7	6	6	6
3	4	4	4	3	1	9	4	4	4
4	14	13	13	13	13	27	18	14	13
5	9	6	6	5	4	11	8	8	6
6	5	3	3	3	2	6	4	4	3
7	7	4	4	4	4	11	5	5	4
8	13	10	8	5	5	14	11	11	10
9	8	8	8	5	5	12	12	8	8
10	8	5	5	5	5	10	8	8	5

Cuadro 3: Valores óptimos

	C=2					C=3			
Instancia	M_{max}=8	9	10	11	12	M_{max}=6	7	8	9
1	19	15	15	15	15	30	21	17	15
2	8	7	7	4	4	10	8	8	7
3	9	4	4	3	1	9	4	4	4
4	24	19	19	18	13	30	23	21	19
5	13	9	9	8	7	16	14	13	9
6	5	4	4	3	2	7	4	4	4
7	11	9	8	8	7	13	9	9	9
8	18	16	13	12	12	20	19	18	16
9	13	8	8	5	5	16	12	9	8
10	9	6	6	6	6	13	10	9	6

Cuadro 4: Mejores resultados obtenidos

	C=2					C=3			
Instancia	M_{max}=8	9	10	11	12	M_{max}=6	7	8	9
1	11	11	11	11	11	28	18	11	11
2	7	6	10	4	3	7	6	7	6
3	5	4	4	4	4	12	8	8	5
4	14	13	13	13	13	27	18	14	13
5	9	6	6	7	4	11	9	9	8
6	5	3	5	3	3	8	5	5	4
7	7	4	4	4	4	11	5	5	5
8	13	20	15	11	7	14	11	11	10
9	13	8	8	8	8	12	12	13	8
10	8	5	5	5	5	12	14	8	8

Cuadro 5: Mejores resultados Boctor

Los resultados presentados en la tabla anterior demuestran que si bien se pudo encontrar soluciones a algunas de las instancias elegidas, no todas pudieron ser resueltas dentro de los parámetros establecidos (número iteraciones, etc).

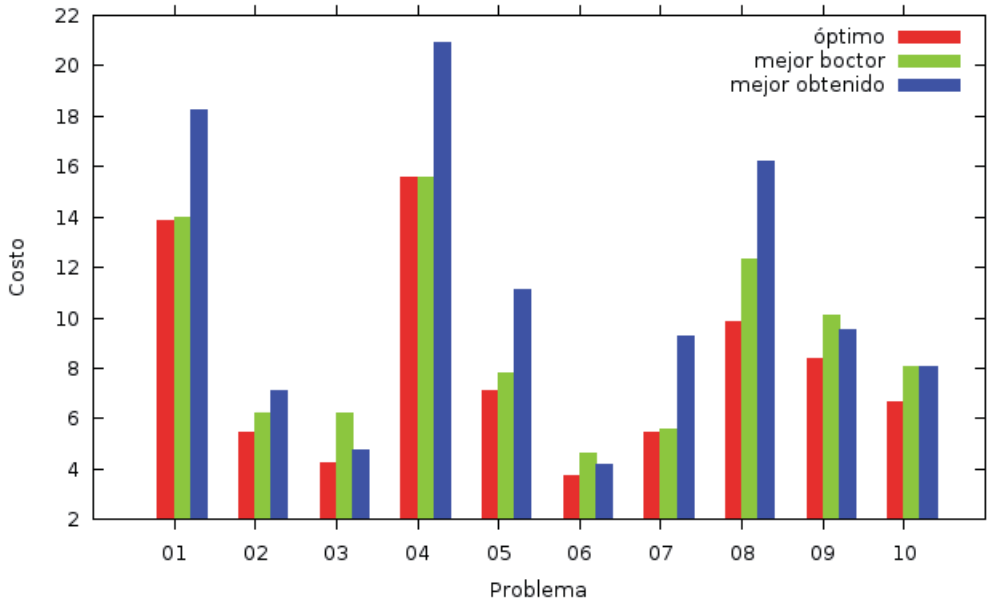

Figura 11: Mejores obtenidos vs óptimos y mejores de Boctor

4.5.2. Implementación CPU vs OpenCL

Mediante la implementación de la búsqueda local tanto en CPU (de modo secuencial) como en OpenCL (implementación paralela), que se encuentra optimizado para su uso en GPU's. Se pudo realizar una comparación del desempeño del solver implementado corriendo la función de costos, tanto en CPU (secuencial) como en OpenCL (paralelo).
Estas pruebas se llevaron a cabo utilizando un equipo de escritorio con un procesador AMD A10-5800K APU (3.8GHZ) que cuenta con un chip gráfico Radeon HD 7660D, y 8 GB de ram, donde 512MB son utilizados como memoria de video.

A continuación se presenta una tabla con los resultados obtenidos en la resolución del problema 1 utilizando 1000 iteraciones.

Item	CPU	OpenCL (GPU)
Total Ejecución Solver	1924.75 ms	4636.16 ms
Total ejecución Búsqueda Local	1837.84 ms	4100.41 ms
Uso CPU	~97 %	~45 %
Overhead OpenCL búsqueda local	0 ms	2104.55 ms
Tiempo cálculo de costos	1380.81 ms	1995.86 ms
Cuello de Botella	std::vector (~36 %)	Overhead OpenCL (~50 %)

Cuadro 6: Comparación entre OpenCL y CPU; tiempo ejecución solver

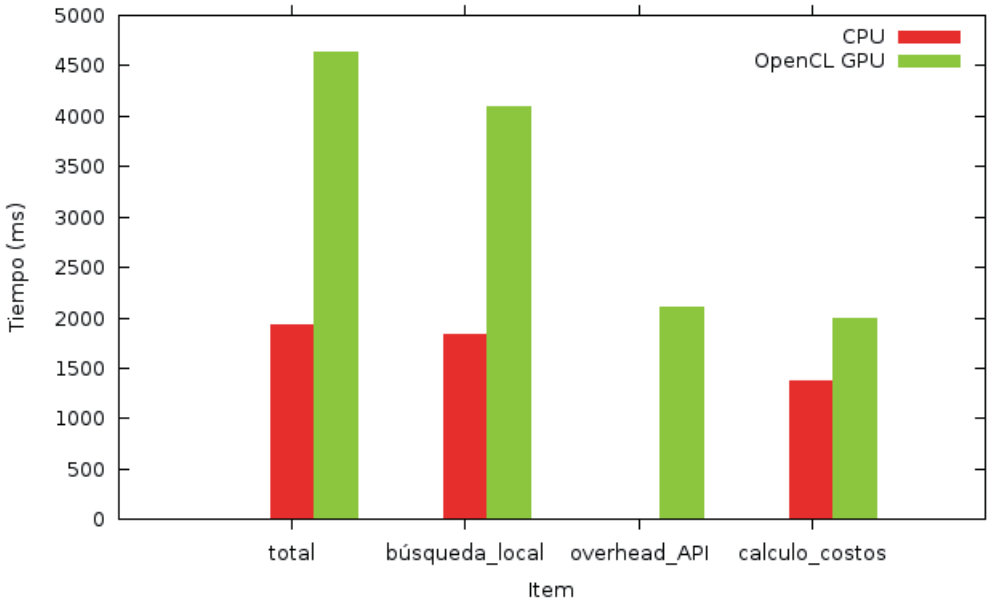

Figura 12: Tiempos CPU vs OpenCL GPU

Los resultados mostrados en la tabla de comparación OpenCL vs CPU fueron obtenidos al realizar un perfilamiento del solver, corriendo una instancia con 1000 iteraciones, mediante la herramienta *perf* que muestra el porcentaje de tiempo utilizado en cada función o componente del programa (respecto del tiempo total de ejecución). Adicionalmente para la parte OpenCL se utilizó la herramienta de perfilamiento OpenCL que provee AMD (AMDAPPProfiler) , la que recopila los tiempos de ejecución detallados de los distintos componentes del runtime OpenCL, tal como tiempos de escritura en buffers y ejecuciones de kernels.

Item	Tiempo utilizado (ms)	Porcentaje
Total Ejecución Solver	4636.16	100
Esperar eventos OpenCL	4331.93	93,43
Total ejecución kernels	1195.97	25,79
Total kernel costo	1179.45	98,61 (kernels)
Total transferencias memoria	799.88	17,25
Compilación kernels	85.83	1,85
Total kernel penalizaciones	12.36	1,03 (kernels)
Total kernel vecindario	2.19	0,183 (kernels)
Total kernel mejor local	1.95	0,163 (kernels)

Cuadro 7: Detalle tiempos de ejecución OpenCL

En la tabla de detalles de tiempos OpenCL se presentan los tiempos de los principales componentes de la implementación OpenCL, fundamentalmente relacionados con los componentes particulares de esta tecnología, como son el tiempo utilizado por los kernels y las transferencias de memoria.

4.6. Análisis de resultados

Se puede apreciar de la tabla anterior que el tiempo de ejecución en la implementación OpenCL es significativamente mayor al de la implementación secuencial. Esto se puede explicar por la significativa cantidad de overhead que introducen las invocaciones a los kernels y a la parte del API OpenCL que espera a que retornen los kernels para poder retomar el control via CPU. Idealmente se busca ocultar estas latencias realizando otras tareas útiles por parte de la CPU y realizando una espera asíncrona del proceso GPU, pero en el caso del problema que se está resolviendo esto se vuelve muy difícil, ya que al momento de realizar la búsqueda local (proceso paralelizado) el algoritmo depende completamente del resultado de ésta para poder continuar, lo que imposibilita la utilización efectiva de esta técnica de optimización.

Detalle tiempos OpenCL

Adicionalmente, se ve en los resultados que la gran mayoría del tiempo es ocupado por la ejecución del cálculo del óptimo local, que es lo que se busca paralelizar, evidentemente en este caso toma mucho más tiempo que su versión secuencial en CPU. Pero mirando más de cerca se encontró que este tiempo excesivo es explicado en su mayor parte por el método clWaitForEvents, el que corresponde a una parte del API de OpenCL, que se encarga de

esperar asincrónicamente las llamadas a kernels y otros métodos del API, en este caso, las llamadas a los kernels que calculan el vecindario y el costo para la búsqueda local están contenidos dentro de este tiempo, y si nos fijamos solamente en el tiempo de ejecución de los kernels, veremos que constituye alrededor del 25 % del tiempo total de la instancia del solver, junto con un 17 % que corresponde a transferencias de memoria constituyen un tiempo comparable al de su símil secuencial, si se tiene solamente en cuenta la resolución de la búsqueda local. Las llamadas del clWaitForEvents se encuentran entre los eventos de escritura de buffers y las llamadas a los kernels y luego entre éstas y las lecturas de los buffers, teniendo en cuenta esto se puede apreciar que no es posible ejecutar acciones mientras se está esperando a estas llamadas, ya que existe dependencia entre las partes.

Implementación de la búsqueda local paralela

El algoritmo que constituye la búsqueda local tiene varios componentes que poseen dependencia de datos y por tanto hacen más difícil su implementación en una manera paralela. El problema más recurrente tiene que ver con la función de costos, la que requiere de varios puntos de sincronización para poder calcular el costo de cada solución de manera correcta.

La forma en que se maneja la granularidad del paralelismo en la implementación realizada, esto es, a nivel de descomposiciones de ciclos, propicia este problema, al estar basada en la forma secuencial del algoritmo, la que está pensada para evaluar los distintos componentes del vector solución secuencialmente.

También, al problema anterior debe sumársele los tiempos de latencia con la transferencia de datos que representan aproximadamente el 50 % del total de la ejecución de la instancia, tiempos que no existen en la implementación secuencial, que no necesita realizar transferencias de memoria de un dispositivo a otro ni coordinar ejecuciones de procesos paralelos externos.

5. Conclusiones y comentarios finales

A partir de la investigación de un problema con una preponderancia cada vez mayor como es el Manufacturing Cell Design, y aplicando una metaheurística conocida como es Tabu Search se ha podido obtener una mejor perspectiva respecto de otros trabajos que tratan con el mismo problema, a esto sumándole la implementación de un solver utilizando la gran capacidad de proceso que proveen las GPU actuales proporciona enormes posibilidades para desarrollar técnicas y enfoques que antes no eran posibles y prácticos debido a limitaciones en el hardware que ahora han desaparecido.

Se eligió el estándar para procesamiento de propósito general en GPU OpenCL, el cuál que ofrece mayor portabilidad entre hardware que su competencia directa CUDA, tecnología privativa de NVidia. Esto significa que la implementación realizada puede funcionar en virtualmente cualquier procesador (CPU o GPU) moderno, independiente del fabricante (siempre que se cuente con el runtime OpenCL adecuado), incluso es posible correr el solver obtenido puramente en CPU, con y sin múltiples núcleos.

La tecnología OpenCL, presenta una capa de abstracción sobre las características específicas del hardware, pero de tal manera que incorpora el diseño arquitectónico paralelo dentro de su modelo lógico. Utilizando las herramientas específicas de la tecnología elegida y lenguaje C/C++ con extensiones particulares fue posible obtener un modelo de programación y un conjunto de tecnologías que ofrecen la mayor independencia al hardware posible pero manteniendo muchas optimizaciones que se implementan de manera automática. De esta manera el trabajo realizado, incluyendo el código fuente es portable entre distintos dispositivos hardware, lo que beneficia de gran manera la posibilidad de hacer comparaciones de desempeño.

La implementación realizada en cuanto al algoritmo secuencial ha dado resultados que se consideran buenos, ya que están en cercanía con otro trabajo conocido y en una instancia incluso pueden encontrar el óptimo global y en otras se acercan bastante. Por otro lado en cuanto a la implementación paralela el desempeño logrado fue muy inferior a lo que se esperaba, incluso muy inferior a la implementación secuencial de referencia. De esto podemos concluir que debido a la naturaleza del tipo de computación necesario para implementar un algoritmo de búsqueda tabú se necesita mayormente operaciones de control a diferencia de las operaciones de cálculo puramente, esto sumado a la arquitectura de memoria no unificada entre CPU y GPU la

eficiencia de la implementación realizada ha distado de lo previsto, no tanto debido ajustes de parámetros incorrectos sino a decisiones de diseño y a la misma naturaleza de los requerimientos computacionales.

Es necesario notar además que la arquitectura hardware GPU actual está constantemente en cambio, con nuevos modelos apareciendo cada 6 meses, con capacidades nuevas, esto tiene repercuciones claras en un tipo de investigación como la presente, que está firmemente ligada a la arquitectura hardware, ya que nuevos avances en tecnología GPU podrían hacer desaparecer limitancias como la del traspaso de datos entre la memoria de video y la memoria principal, por ejemplo, aumentando la eficiencia del solver aquí presentado.

Como resultado del diseño e implementación se obtuvo un solver que puede competir, en cuanto a optimización de costos, con otros trabajos, aunque no supera, al menos los resultados de referencia que se utilizaron. En cuanto a rapidez se ha usado como referencia una versión secuencial del algoritmo que se comparó con la versión paralelizada implementada utilizando OpenCL, los tiempos de ejecución resultaron ser sostenidamente más largos en la implementación paralela, exactamente lo opuesto de lo planteado inicialmente, esto debido al gran uso de instrucciones de control, como ciclos y condicionales. Aunque se mitigó esto último convirtiendo algunos condicionales en operaciones aritméticas, no fue suficiente para igualar siquiera la implementación secuencial debido en este caso a la gran sobrecarga del API OpenCL versus el no poder aprovechar al 100 % la capacidad de paralelismo del hardware debido a la naturaleza del diseño del algoritmo.

De ninguna manera se descarta que la capacidad de procesamiento paralelo de las GPU actuales sea utilizable para resolver problemas como el Manufacturing Cell Design utilizando la metaheurística de Tabu Search. Como trabajo futuro se propone un estudio y pruebas a más bajo nivel del diseño básico propuesto teniendo en cuenta las capacidades específicas de la plataforma hardware / software a utilizar y buscar distintas maneras de optimización en la implementación a realizar.

Referencias

[1] *OpenCL 1.1 Specification*, 2010.

[2] G. K. Adil, D. Rajamani, and D. Strong. A mathematical model for cell formation considering investment and operational costs. *European Journal of Operational Research*, 69(3):330 – 341, 1993.

[3] S. Ahkioon, A. A. Bulgak, and T. Bektas. A comprehensive mathematical modeling approach to the design of cellular manufacturing systems integrating production planning and system reconfiguration. *European Journal of Operational Research, 192*, pages 414–428, 2009.

[4] P. K. Arora, A. Haleem, and M. K. Singh. Cell formation techniques – A study. *International Journal of Engineering Science and Technology*, pages 1178–1181, 2011.

[5] M. A. Bajestani, M. Rabbani, A. R. Rahimi-Vahed, and G. B. Khoshkhou. A multi-objective scatter search for a dynamic cell formation problem. *Computers and Operations Research*, 36(3):777–794, 2009.

[6] R. Battiti and G. Tecchiolli. The reactive tabu search. *INFORMS Journal on Computing*, 6(2):126–140, 1994.

[7] F. F. Boctor. A linear formulation of the machine-part cell formation problem. *International Journal of Production Research*, 29(2):343–356, 1991.

[8] J. Chakrapani and J. Skorin-Kapov. Massively parallel tabu search for the quadratic assignment problem. *Annals of Operations Research*, forthcoming, 1992.

[9] W.-H. Chen and B. Srivastava. Simulated annealing procedures for forming machine cells in group technology. *European Journal of Operational Research*, 75(1):100 – 111, 1994.

[10] M. T. T. G. Crainic and M. Gendreau. Towards a taxonomy of parallel tabu search algorithms. Technical report, Centre de Recherche sur les Transports, 1993.

[11] M. T. T. G. Crainic and M. Gendreau. Towards a taxonomy of parallel tabu search algorithms. pages 61–72, 1997.

[12] T. Crainic and M. Toulouse. Parallel metaheuristics. In *Proc. First meeting of the PAREO working group on Parallel Processing in Operations Research PAREO '98 Conference*, 1998.

[13] T. G. Crainic. Parallel asynchronous tabu search for multicommodity location-allocation with balancing requirements. *Annals of Operations Research*, pages 277–299, 1996.

[14] T. G. Crainic, P. Dejax, and L. Delorme. Models for multimode multicommodity location problems with interdepot balancing requirements. *Annals of Operations Research*, 18:277–302, 1989.

[15] M. Czapinski and S. Barnes. Tabu search with two approaches to parallel flowshop evaluation on CUDA platform. *Journal of Parallel and Distributed Computing*, 71(6):802–811, 2011.

[16] J. Dabrowski. Parallelization techniques for tabu search. In *Applied Parallel Computing. State of the Art in Scientific Computing (8th PARA'06)*, volume 4699, pages 1126–1135. 2007.

[17] C. Dimopoulos and A. M. S. Zalzala. Recent developments in evolutionary computation for manufacturing optimization: problems, solutions, and comparisons. *IEEE Trans. Evolutionary Computation*, 4(2):93–113, 2000.

[18] O. Duran, N. Rodríguez, and L. Airton. Collaborative particle swarm optimization with a data mining technique for manufacturing cell design. *Expert Systems with Applications*, 37(2):1563–1567, 2010.

[19] C.-N. Fiechter. A parallel tabu search algorithm for large traveling salesman problems. *Discrete Applied Mathematics*, 51(3):243–267, 1994.

[20] M. Flynn. Very high speed computing. *Proceedings of the IEEE*, pages 1901–1909, 1966.

[21] F. Glover. Tabu search - Part 1. *ORSA Journal on Computing*, 1(3):190–206, 1989.

[22] F. Glover. Tabu search - Part 2. *ORSA Journal on Computing*, 2(1):4–32, 1989.

[23] F. Glover and B. Melián-Batista. Búsqueda tabú. *Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial*, 7(19):29–48, 2003.

[24] F. Jolai, R. Tavakkoli-Moghaddam, A. Golmohammadi, and B. Javadi. An electromagnetism-like algorithm for cell formation and layout problem. *Expert Systems with Applications*, 39(2):2172–2182, 2012.

[25] C. Lin, C. Chang, and F. Li. An efficient tabu search approach to determine cell formation problem with consideration of cell layout. In *IEEM*, pages 1441–1445, 2011.

[26] I. Mahdavi, B. Javadi, K. Fallah-Alipour, and J. Slomp. Designing a new mathematical model for cellular manufacturing system based on cell utilization. *Applied Mathematics and Computation*, 190(1):662 – 670, 2007.

[27] M. J. Meena, K. R. Chandran, A. Karthik, and A. V. Samuel. A parallel ACO algorithm to select terms to categorise longer documents. *International Journal of Computational Science and Engineering*, 6(4):238–248, 2011.

[28] N. Ngampak and B. Phruksaphanrat. Cellular manufacturing layout design and selection: A case study of electronic manufacturing service plant. In *Proceedings of the International MultiConference of Engineers and Computer Scientists 2011 Vol II, IMECS*, pages 1182–1187, 2011.

[29] V. P. Reddy, G. Michael, and M. Umamaheshwari. Coarse-grained parallel genetic algorithm to solve the shortest path routing problem using genetic operators. *Indian Journal of Computer Science and Engineering*, pages 39–42, 2011.

[30] R. Soto, H. Kjellerstrand, O. Durán, B. Crawford, E. Monfroy, and F. Paredes. Cell formation in group technology using constraint programming and boolean satisfiability. *Expert Systems with Applications*, 39(13):11423–11427, 2012.

[31] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering*, 12(3):66–73, 2010.

[32] M. Taghavifard. Scheduling cellular manufacturing systems using ACO and GA. *International Journal of Applied Metaheuristic Computing*, 3(1):48–64, 2012.

[33] L. T. Thanh, J. A. Ferland, N. D. Thuc, and V. H. Nguyen. Simulated annealing method with different neighborhoods for solving the cell for-

mation problem. In *International Joint Conference on Computational Intelligence (ECTA-FCTA)*, pages 525–533, 2011.

[34] S. Voß. Meta-heuristics: The state of the art. In *Local Search for Planning and Scheduling*, volume 2148 of *Lecture Notes in Computer Science*, pages 1–23. 2001.

[35] T.-H. Wu, C.-C. Chang, and S.-H. Chung. A simulated annealing algorithm for manufacturing cell formation problems. *Expert Systems with Applications*, 34(3):1609–1617, 2008.