Paralelización de un algoritmo genético para el problema del agente viajero

Moisés Emmanuel Ramírez Guzmán*, Erik Germán Ramos Pérez**

Resumen

En el presente trabajo se muestra el uso de software para la implementación secuencial, paralela y distribuida de un algoritmo genético para tratar el problema del agente viajero. La implementación está hecha usando software libre. Como resultados se muestra el uso de diferentes operadores genéticos y la generación de archivos de trazas para monitorear gráficamente la ejecución del algoritmo.

1. Introducción

La formulación y análisis del problema del agente viajero (TSP) tiene cerca de dos siglos de antigüedad. Diversas aplicaciones como la cristalografía de rayos X, el reacondicionamiento de motores de turbina de gas o la perforación de placas de circuitos impresos pueden ser modeladas y solucionadas como un TSP.

Un método que brinda una solución exacta del TSP, consiste en probar cada una de las posibles permutaciones para encontrar el camino de menor longitud, cuyo inconveniente es un alto costo computacional. El trabajo presenta una implementación paralela y distribuida de algoritmos genéticos (AG) para generar una buena aproximación al óptimo con un menor costo en tiempo y computacional.

El desarrollo del AG contempla la representación del problema, los operadores genéticos usados y la función objetivo. La implementación está hecha en C, usando en primer lugar la biblioteca abierta de multiprocesamiento (OpenMP) para paralelizar el cómputo en cada nodo, y por otro lado, la interfaz de paso de mensajes (MPI) para permitir la comunicación entre éstos.

La metodología usada es: implementación secuencial, después de un análisis de ejecución usando una herramienta de perfil de ejecución (gprof) se detectaron las partes que ocupan mayor cantidad de procesamiento. Se analizaron estas regiones y se procedió a paralelizarlas y distribuirlas en los diferentes nodos.

Para hacer un análisis de rendimiento y comunicación entre los nodos en tiempo de ejecución, se usa una herramienta de instrumentación para aplicaciones paralelas y distribuidas de Cómputo de Alto Rendimiento (HPC). Al final del documento, se presenta un análisis y una comparativa de mejora de velocidad entre las implementaciones secuencial, paralela y paralela-distribuida.

* merg@mixteco.utm.mx. Universidad Tecnológica de la Mixteca.
** erik@mixteco.utm.mx. Universidad Tecnológica de la Mixteca.
Colaboración de Gil Hernández Ledesma, hernandezlg1f@gmail.com. Universidad Tecnológica de la Mixteca.

2. Marco teórico

El origen de los AG está en el análisis de la naturaleza (genética y evolución) para ayudar a resolver problemas de optimización. Estos algoritmos utilizan 3 operadores: cruza, mutación y selección para modificar poblaciones iteradamente [1].

Un AG, en general contiene los siguientes pasos:

- Generar una población con N individuos.
- Evaluar función de aptitud para cada elemento de la población.
- Repetir para un número de generaciones determinado o hasta cumplir alguna regla de terminación:
  a. Aplicar un criterio de selección sobre los individuos de la población.
  b. Modificar los individuos seleccionados utilizando operadores de cruza y mutación.
  c. Crear una nueva generación a partir de elementos nuevos y/o anteriores.
  d. Evaluar la función de aptitud de los individuos nuevos o alterados.
  e. Mostrar los mejores individuos.

2.1 Representación del problema y Operadores genéticos

La **representación del problema** es uno de los aspectos más importantes y es crucial para la solución del mismo. El problema abordado en este documento es representado utilizando valores enteros. El algoritmo inicia generando una permutación que representa una solución factible, que es tratada en el AG con los siguientes operadores:

Operador de selección: Selecciona individuos de la población basándose en la aptitud. Uno de los más utilizados es el de ruleta (ver Figura 1), donde cada individuo tiene una probabilidad de ser seleccionado de acuerdo a su aptitud. Otros operadores de selección importantes son: torneo y ranking.

Operador de cruza: Combina a dos individuos de la población mediante uno o más puntos de cruza, los cuales dividen al individuo en partes para que se formen nuevos, tras combinarlas (ver Figura 2). En esta implementación después de generar los descendientes, se deben detectar e intercambiar los elementos repetidos en los descendientes de tal forma que cumplan con el criterio del ciclo hamiltoniano.

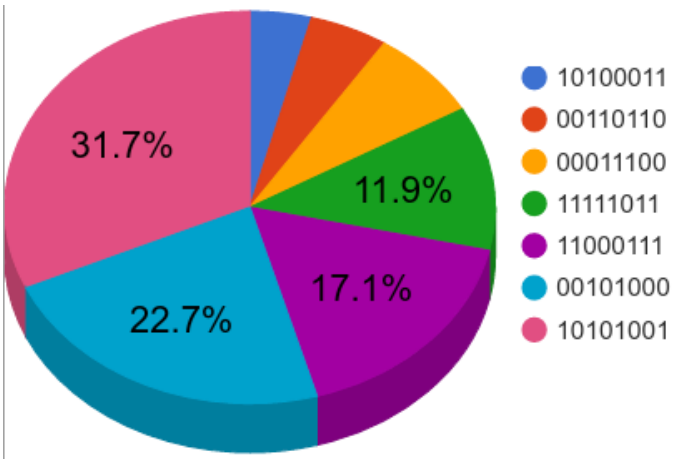

Figura 1: Selección por rueda de la ruleta

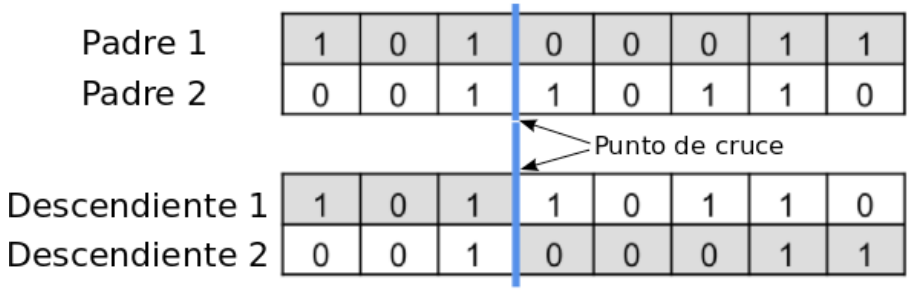

Figura 2: Cruza en un punto

Operador de mutación: Modifica el gen de un individuo, la mutación variará de acuerdo al problema y su representación (ver Figura 3). En problemas con representación binaria, es común invertir un bit seleccionado aleatoriamente. En esta implementación para el TSP se intercambian dos valores de posiciones seleccionadas aleatoriamente.

Figura 3: Mutación en un bit

En [2] se pueden ver otras implementaciones del operador de cruza y mutación.

2.2 Problema del agente viajero

El TSP consiste en determinar el orden óptimo de visita de un conjunto prescrito de ciudades de tal forma que la distancia recorrida sea mínima, cada ciudad se visite exactamente una vez, se inicie y termine en una misma ciudad [3] (ver Figura 4). La importancia del problema es que el espacio de búsqueda es (n-1)!/2, volviéndolo un problema NP-hard. Para problemas con un número grande de nodos es más práctico obtener una respuesta aproximada, siendo un nicho para la aplicación de metaheurísticas como los AG [1].

Otros problemas importantes tratados con AG son: la planificación de tareas (task scheduling problem) [4], el problema de empaquetado (bin packing problem) [1] y el problema de planificación de rutas (vehicle routing problem) [1].

2.3 Programación paralela y el HPC

Las necesidades de HPC, se hacen notorias debido al incremento de operaciones que deben ser llevadas a cabo para resolver problemas modernos, además de mayores requerimientos de almacenamiento de datos, tanto de manera permanente como durante el tiempo en que se procesan para generar información.

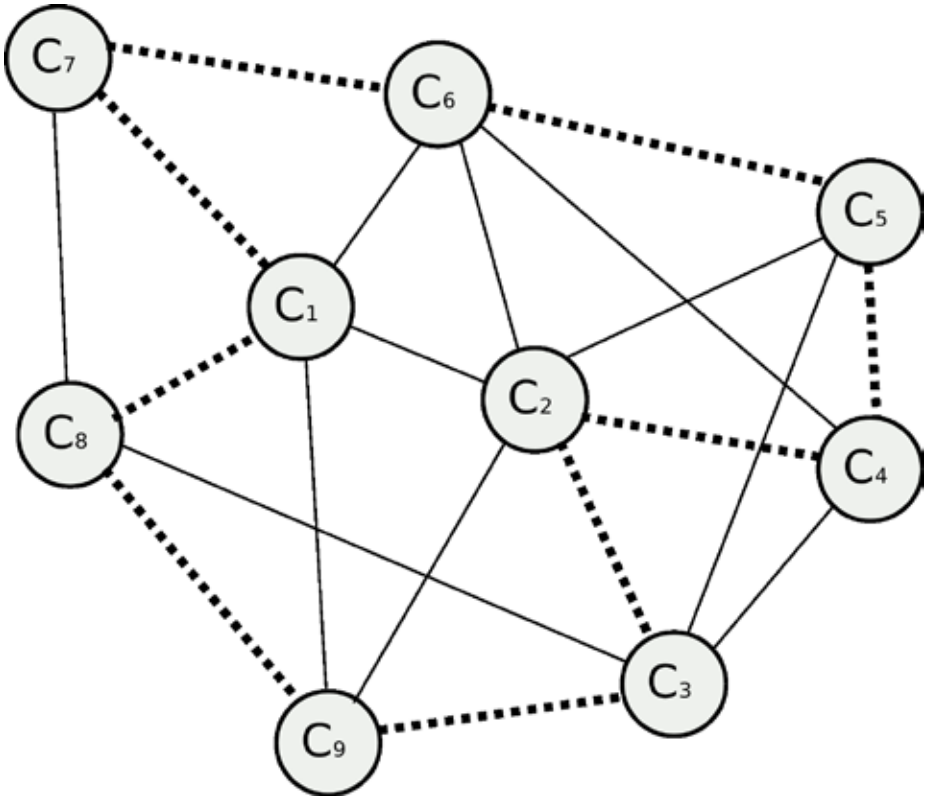

Figura 4: Representación gráfica del problema del TSP, las líneas punteadas muestran un ciclo hamiltoniano, que visita todas las ciudades una sola vez.

El procesamiento de datos usando equipos de cómputo tuvo en sus inicios su mayor auge de desarrollo durante la Segunda Guerra Mundial. A partir de ese momento se empezaron a utilizar computadoras con diferentes tecnologías debido a las necesidades de procesamiento que fueron surgiendo. Después de la guerra, la industria y el gobierno, en convenio con diferentes instituciones académicas alrededor de todo el mundo siguieron haciendo uso y promoviendo el desarrollo de estas tecnologías, debido principalmente al incremento de sus necesidades de procesamiento para la solución de problemas complejos [5]. Algunos problemas tratados con HPC son: modelado del clima [6], procesos biológicos [7], químicos y físicos [8, 9], defensa y armamento, visualización de datos, modelos matemáticos [11], inteligencia artificial [12], entre otros.

OpenMP es una interfaz de programación para C, C++ y Fortran que permite la ejecución de múltiples procesos usando la arquitectura de memoria compartida a nivel de cada nodo o computadora. Usa el modelo de ejecución fork-join, que consiste en tener un nodo maestro que va ejecutando el flujo principal del programa, al tener instrucciones que pueden ser ejecutadas en paralelo se usa una directiva (#pragma) que crea hilos de ejecución (fork). Al terminar la ejecución de los procesos paralelos se devuelve el control al nodo maestro (join). La figura 5 ilustra gráficamente este modelo.

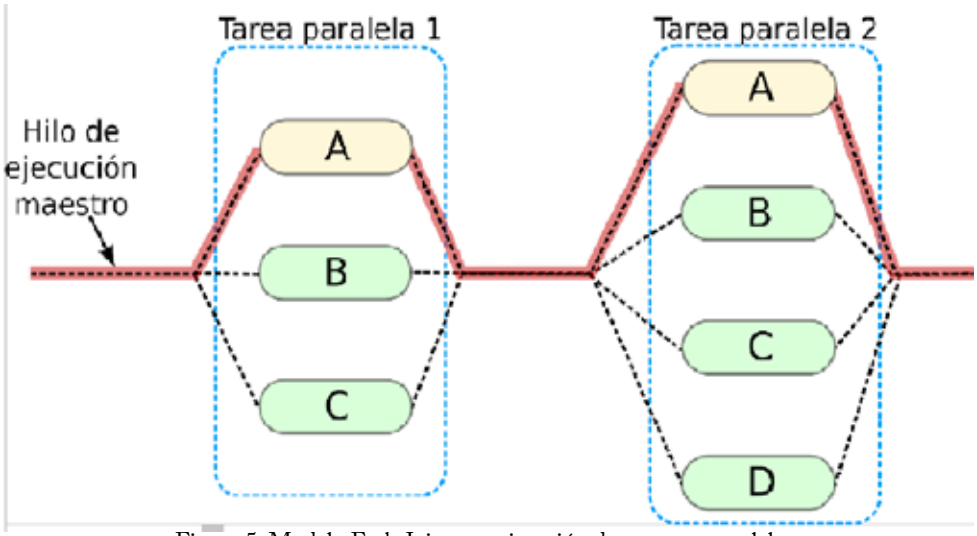

Figura 5: Modelo Fork-Join para ejecución de procesos paralelos.

La Interfaz de Paso de Mensajes es la primera biblioteca de paso de mensajes que ofreció escalabilidad, portabilidad, rendimiento, estandarización y una implementación eficiente para el desarrollo de aplicaciones paralelas y distribuidas [13]. El estándar define la sintaxis y semántica de funciones contenidas en una biblioteca diseñada para ser usada para la creación de programas que permitan usar múltiples procesadores que se comunican ya sea a través de una red de interconexión o usando memoria compartida [14]. Las implementaciones más ampliamente usadas son: Open MPI*** y MPICH****. Un esquema que combina memoria compartida (OpenMP) y memoria distribuida (MPI) se muestra en la Figura 6, que contiene 4 nodos, cada uno con 4 unidades de procesamiento (cada uno con un bloque de memoria local). Los 4 nodos se pueden comunicar a través de una red de interconexión usando mensajes.

3. Desarrollo

Como se ha mencionado, las herramientas usadas para tratar el problema son OpenMP y MPI. Para ello se muestra en la presente sección la configuración del clúster, las instrucciones principales en ambas bibliotecas y un esquema de las implementaciónes del AG.

3.1 Configuración del clúster *Beowulf*

Para la ejecución de los AGs se configuró un clúster Beowulf, que permite accesar a los recursos de cómputo y almacenamiento desde cualquier nodo en una red. La instalación implica instalar el compilador de C++, y los paquetes correspondientes a la implementación de OpenMPI o MPICH. El siguiente paso es la configuración de servidores SSH para establecer conexiones seguras entre los nodos. Al hacer esto todos los nodos tienen una configuración idéntica y se

***https://www.open-mpi.org/ (última visita 18 de abril de 2017)
****https://www.mpich.org/ (última visita 18 de abril de 2017)

genera un archivo con claves de autenticación que permiten establecer conexiones MPI sin pedir la contraseña cada vez que se deseen transmitir datos entre los nodos (ssh-keygen).

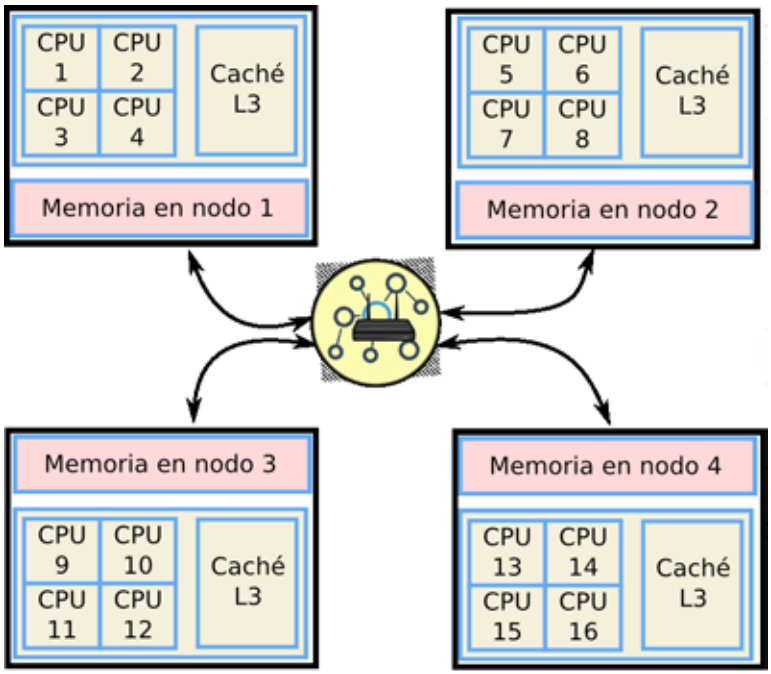

Figura 6: Configuración de acceso a memoria no uniforme (NUMA) que combina arquitecturas de memoria compartida y distribuida

Existen otras alternativas para desarrollar programas paralelos y distribuidos, por ejemplo: Intel® Threading Building Blocks (Intel® TBB), biblioteca que utiliza plantillas de C++ para el desarrollo de programas paralelizados. En esté se incluyen algoritmos, operaciones atómicas, un programador de tareas y un sistema de asignación de memoria escalable.

La biblioteca cuenta con una versión comercial con asistencia personalizada y una open source que pasó a distribuirse bajo la licencia Apache 2.0 en el 2017, además ésta al igual que OpenMP y MPI se ha utilizado en investigaciones de Matemáticas [15, 16], Física, medicina [17] y Computación [18].

En los últimos años, ha crecido el uso de unidades de procesamiento gráfico (GPUs), para ello existen principalmente dos alternativas que son OpenCL (software libre) y CUDA (distribuido como freeware), la cual se ha utilizado por ejemplo, en investigaciones de experimentos físicos [19, 20], de visión computacional y reconocimiento de patrones [15] o de data clustering [21].

Las principales alternativas para el desarrollo de aplicaciones paralelas y distribuidas son distribuidas libremente, a pesar de que cuentan con versiones comerciales no existe mayor diferencia que el servicio de asistencia personalizada que brinda la compañía propietaria de la tecnología, de esta forma es como el software libre a través de OpenMP, MPI e Intel® TBB lidera el desarrollo de aplicaciones paralelas y distribuidas.

3.2 Estructura general de un programa en C/C++ con MPI y OpenMP

Los programas que utilizan MPI y OpenMP se componen principalmente de las siguientes instrucciones básicas [22] :

MPI_Init(&argc, &argv): Prepara al sistema para que pueda recibir las llamadas de las funciones MPI.

MPI_Finalize(): Función que libera los recursos adquiridos por la biblioteca MPI.

MPI_Send y **MPI_Recv**: Son instrucciones que permiten implementar comunicación punto a punto.

Estas instrucciones tienen como restricción que pueden estar únicamente en la función main del programa. Además, pueden combinarse regiones de código MPI y OpenMP, usando para éste último la declaración de regiones paralelas usando las directivas **#pragma**. Una de las directivas básicas es parallel, la cual crea el número de hilos indicado por la variable de entorno OMP_NUM_THREADS [23].

3.3 Estructura general de los programas desarrollados.

La primer implementación realizada fue la secuencial. La Figura 7 muestra a nivel general los módulos implementados. Se utilizó la herramienta gprof para detectar las regiones de código que tienen mayor carga de ejecución y que son candidatas a ser paralelizadas. En el Cuadro 1, se muestra que la mayor parte del tiempo es usada en la función getPosRepetido, ésta es usada por el operador de cruza para verificar que no haya ciudades repetidas en un cromosoma. Para la versión paralela se optó por paralelizar el proceso de cruza, usando la directiva #pragma en el ciclo que aplica el operador de cruza.

Cuadro 1: Fragmento de la información generada por la herramienta gprof.

% Tiempo de ejecución	Tiempo acumulado	Número de llamadas	Función
93.55	47.28	18666904	getPosRepetido
3.70	49.15	50	cruza
2.13	50.23		compara
0.62	50.54	2400000	evalua
0.06	50.57	1	inicializar

La implementación con OpenMP y MPI distribuye el cálculo de la función de aptitud y cruza en los nodos. Enviando primero los cromosomas padre y recibiendo los descendientes que cumplen con el criterio del ciclo hamiltoniano. La evaluación de cada individuo se realiza de manera similar.

4. Resultados

Para monitorear la comunicación entre los hilos de ejecución en las implementaciones paralela y distribuida se usó la herramienta de instrumentación Score-P[*****] que agrega instrucciones al código original, de manera que al momento de ejecutar un programa se generen archivos de trazas. Los archivos de trazas generados en formato OTF[******] (Open Trace Format); posteriormente, pueden ser

[*****] https://www.vampir.eu/ (última visita 18 de abril de 2017)
[******] https://tu-dresden.de/zih/forschung/projekte/otf (última visita 18 de abril de 2017)

visualizados usando programas como Vampir[*******] con el objetivo de detectar el uso de los recursos como CPU, memoria y comunicaciones entre procesos. En la Figura 8 se aprecia el inicio de ejecución del AG implementado con OpenMP y MPI. Particularmente se aprecia el inicio del programa y la ejecución de una región paralela usando OpenMP. En la Figura 9 se visualiza la comunicación entre 4 procesos MPI, se aprecia el hilo principal y mensajes para envío y recepción de datos.

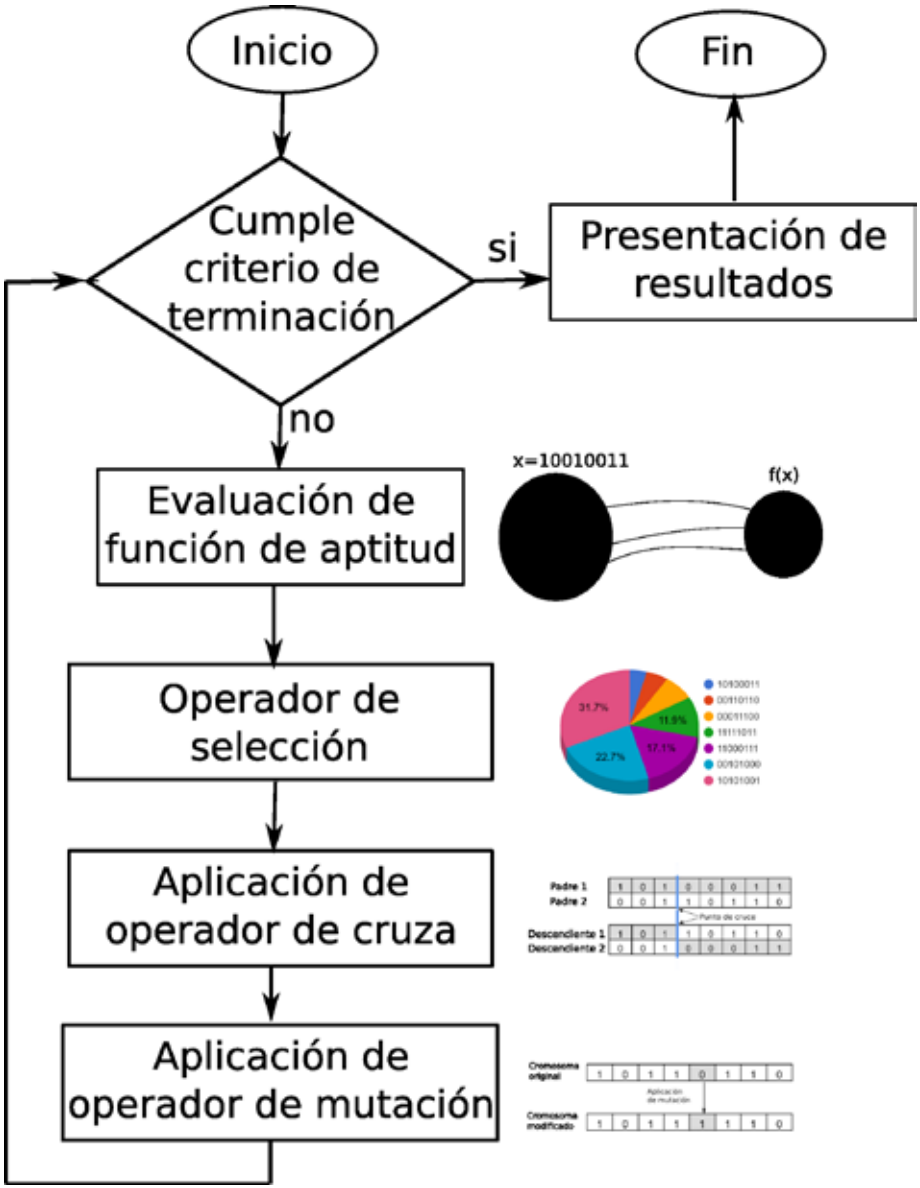

Figura 7: Módulos de la implementación de un AG

[*******] https://www.vampir.eu/ (última visita 18 de abril de 2017)

Figura 8: Uso de Vampir para visualizar las comunicaciones entre los hilos.

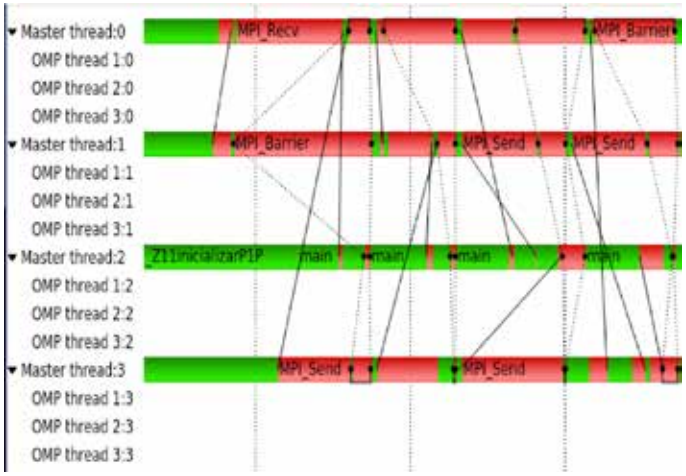

Figura 9: Visualización de comunicaciones entre procesos MPI usando la herramienta Vampir.

El cuadro 2 muestra los resultados en tiempo de ejecución del AG usando las tres versiones. Los tiempos de ejecución en la versión distribuida no son proporcionales a la cantidad de iteraciones debido a las comunicaciones (tiempos de espera y sincronización). Se aprecia que la versión que tiene mejor rendimiento es la paralela, esto debido a que no requiere tiempos de espera considerables. Las pruebas fueron realizadas en dos equipos Dell PowerEdge T310 con procesadores Xeon X3440@2.53 GHz interconectados por un Switch HP V1405-8G 10/100/1000Mbps.

Cuadro 2: Tiempo de ejecución del AG en la versión secuencial con una población de 48,000 individuos..

Versión de la implementación.	Número de generaciones	
	25	**50**
Secuencial	53.25 seg.	88.07 seg.
Paralela (OpenMP)	12.98 seg.	22.40 seg.
Distribuida – Paralela (OpenMP + MPI)	44.25 seg,	52.42 seg.

La implementación realizada consideraba tres tipos de operadores de selección: aleatoria, ranking y por torneo. En las figuras 9, 10 y 11 se muestra el comportamiento de los operadores de selección usados. Además durante la implementación se usa elitismo para conservar a los mejores elementos de cada generación.

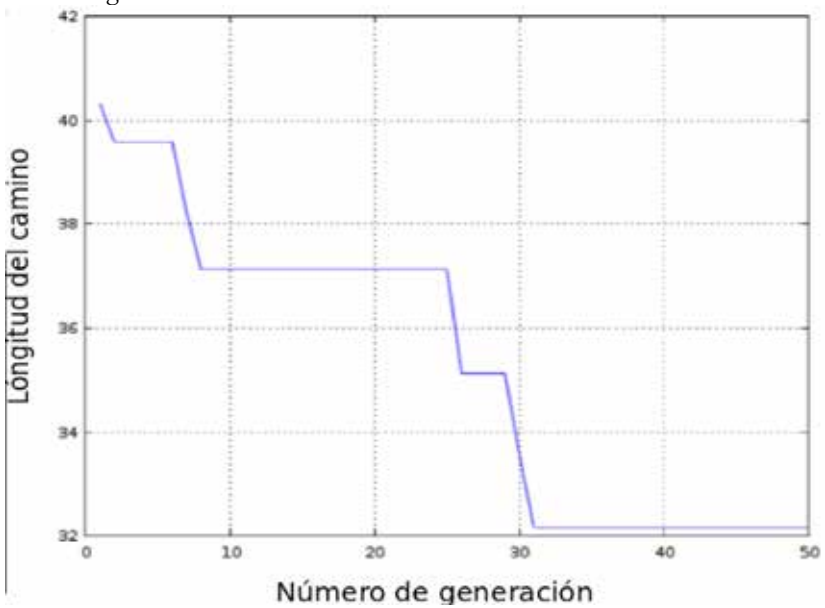

Figura 10: Convergencia a la solución usando el operador de selección aleatoria.

Puede apreciarse que la selección aleatoria converge más rápido a la solución, sin embargo los otros operadores se aproximan a la solución óptima con más generaciones.

Conclusiones

Durante el proyecto se obtuvo la aproximación a la solución óptima para el problema del TSP, siendo este un problema de gran importancia en aplicaciones reales. Se mostró el uso de diferentes operadores y se analizó y monitoreó el comportamiento de las aplicaciones tanto a nivel secuencial como en las ejecución paralela-distribuida.

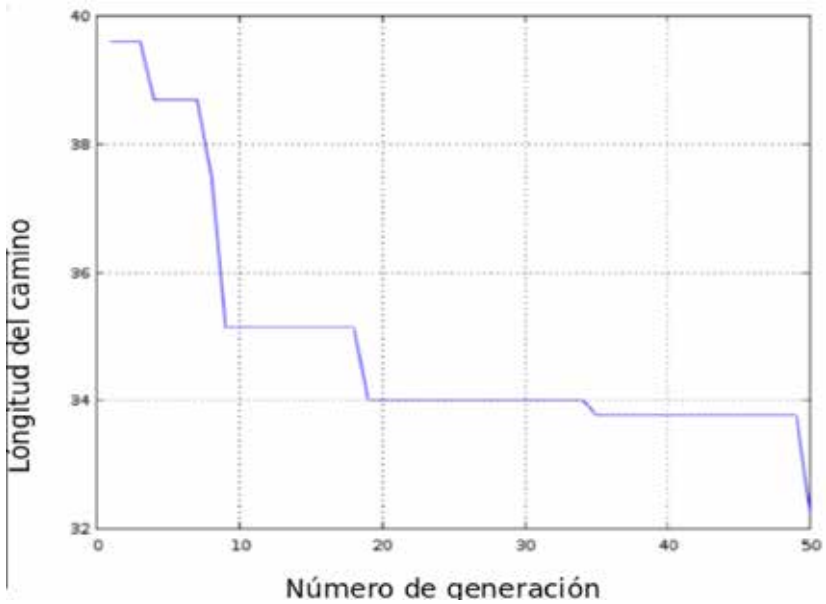

Figura 11: Convergencia a la solución usando el operador de selección rank.

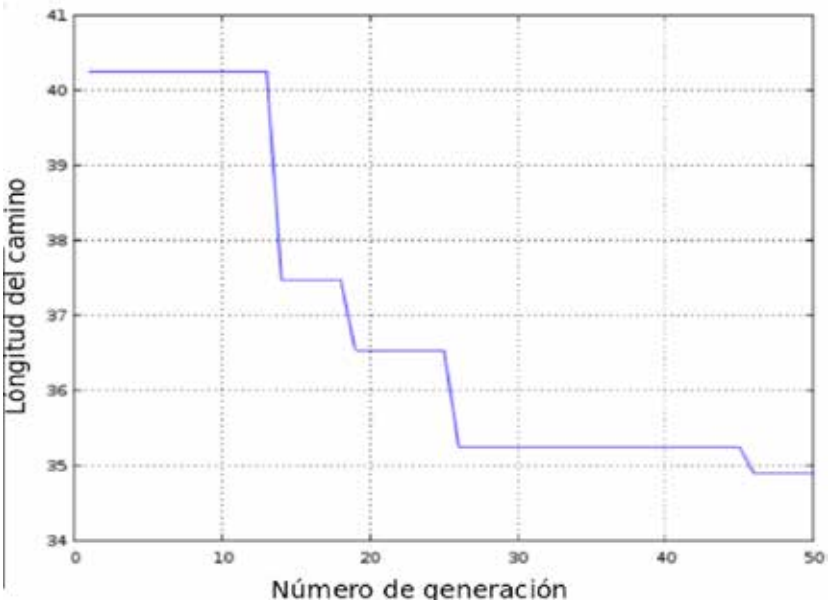

Figura 12: Convergencia a la solución usando el operador de selección por torneo

El uso de software libre permite realizar desarrollo de aplicaciones complejas sin costo. Generalmente los desarrollos libres tienen documentación y muy buen soporte, permitiendo de esta manera ser parte de otros proyectos más complejos.

En cuanto a la aplicación, se detectó visualmente que puede mejorarse el balanceo de carga para obtener mejor tiempo de ejecución en la versión paralela-distribuida.

Bibliografía

[1] Duarte, A., Pantrigo, J. J., & Gallego, M. (2007). Metaheurísticas. Madrid: Dykinson.
[2] Grefenstette, J., Gopal, R., Rosmaita, B., & Van Gucht, D. (1985, July). Genetic algorithms for the traveling salesman problem. In Proceedings of the first International Conference on Genetic Algorithms and their Applications (pp. 160-165).
[3] Gutin, G., & Punnen, A. P. (Eds.). (2006). The traveling salesman problem and its variations (Vol. 12). Springer Science & Business Media.
[4] Omara, F. A., & Arafa, M. M. (2010). Genetic algorithms for task scheduling problem. Journal of Parallel and Distributed Computing, 70(1), 13-22.
[5] Parhami, B. (2000). Computer Architecture: Algorithms and Hardware Designs.
[6] Rodrigues, E. R., Navaux, P. O., Panetta, J., Mendes, C. L., & Kalé, L. V. (2010, December). Optimizing an MPI weather forecasting model via processor virtualization. In High Performance Computing (HiPC), 2010 International Conference on (pp. 1-10). IEEE.
[7] Tang, Y. H., Lu, L., Li, H., Evangelinos, C., Grinberg, L., Sachdeva, V., & Karniadakis, G. E. (2017). OpenRBC: A Fast Simulator of Red Blood Cells at Protein Resolution. arXiv preprint arXiv:1701.02059.
[8] Al-Oraiqat, A. M. (2016). Parallel implementation of the coupled harmonic oscillator. International Journal of Engineering, Science and Technology, 8(6), 105-111.
[9] Lončar, V., Young-S, L. E., Škrbić, S., Muruganandam, P., Adhikari, S. K., & Balaž, A. (2016). OpenMP, OpenMP/MPI, and CUDA/MPI C programs for solving the time-dependent dipolar Gross–Pitaevskii equation. Computer Physics Communications, 209, 190-196.
[10] Boughezal, R., Campbell, J. M., Ellis, R. K., Focke, C., Giele, W., Liu, X., ... & Williams, C. (2017). Color-singlet production at NNLO in MCFM. The European Physical Journal C, 77(1), 7.
[11] Gonzalez, B. P., Sánchez, G. G., Donate, J. P., Cortez, P., & de Miguel, A. S. (2012, May). Parallelization of an evolving artificial neural networks system to forecast time series using openmp and mpi. In Evolving and Adaptive Intelligent Systems (EAIS), 2012 IEEE Conference on (pp. 186-191). IEEE.
[12] Gupta, S., Palsetia, D., Patwary, M. M. A., Agrawal, A., & Choudhary, A. (2014, May). A new parallel algorithm for two-pass connected component labeling. In Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International (pp. 1355-1362). IEEE.
[13] Bridges, P., Doss, N., Gropp, W., Karrels, E., Lusk, E., & Skjellum, A. (1995). User's Guide to MPICH, a Portable Implementation of MPI. Argonne National Laboratory, 9700, 60439-4801.
[14] Quinn, M. J. (2003). Parallel Programming. TMH CSE, 526.
[15] Henderson, P., & Vertescher, M. (2017). An Analysis of Parallelized Motion Masking Using Dual-Mode Single Gaussian Models. arXiv preprint arXiv:1702.05156.

[16] Horn, A., & Kroening, D. (2015, June). Faster linearizability checking via p-compositionality. In International Conference on Formal Techniques for Distributed Objects, Components, and Systems (pp. 50-65). Springer International Publishing.
[17] Dotti, A., Asai, M., Barrand, G., Hrivnacova, I., & Murakami, K. (2015, October). Extending Geant4 Parallelism with external libraries (MPI, TBB) and its use on HPC resources. In Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC), 2015 IEEE (pp. 1-2). IEEE.
[18] Playne, D. P., & Hawick, K. A. (2011, May). Auto-generation of parallel finite-differencing code for mpi, tbb and cuda. In Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on (pp. 1168-1175). IEEE.
[19] Qi, R., Botello-Smith, W. M., & Luo, R. (2017). Acceleration of Linear Finite-Difference Poisson-Boltzmann Methods on Graphics Processing Units. arXiv preprint arXiv:1704.02745.
[20] Carter, F., Hitschfeld, N., Navarro, C., & Soto, R. (2017). GPU parallel simulation algorithm of Brownian particles with excluded volume using Delaunay triangulations. arXiv preprint arXiv:1703.02484.
[21] Javadi, R., & Ashkboos, S. (2017). An Efficient Parallel Data Clustering Algorithm Using Isoperimetric Number of Trees. arXiv preprint arXiv:1702.04739.
[22] Michael, J. Q. (2004). Parallel Programming in C with MPI and OpenMP. Dubuque, IA: McGraw-Hill.
[23] Pacheco, P. (2011). An introduction to parallel programming. Elsevier.