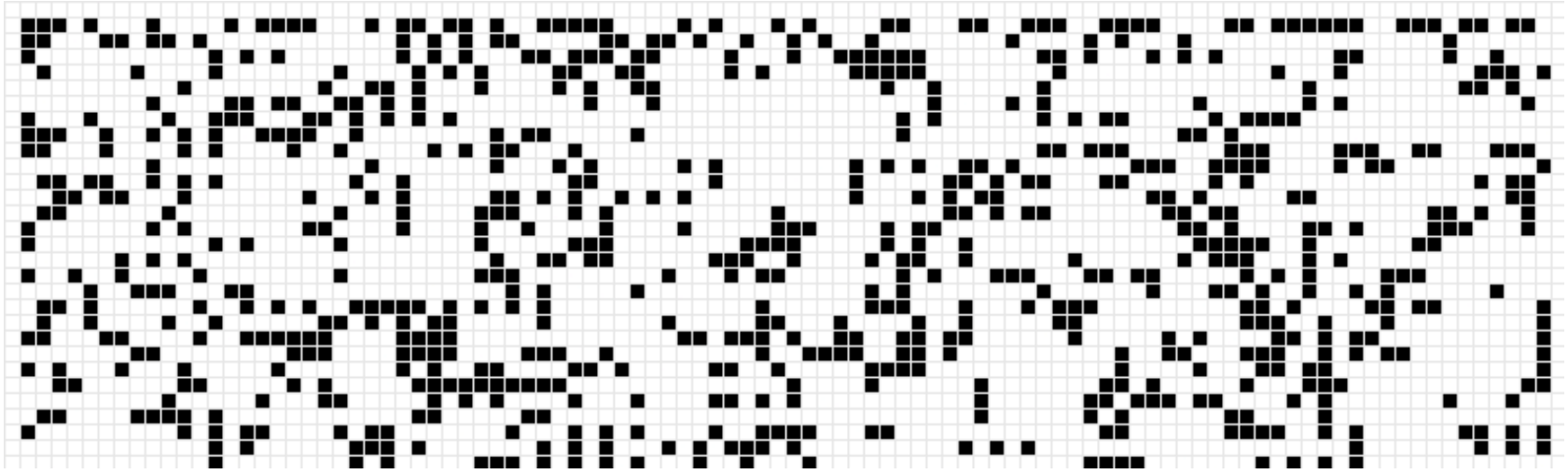


Juego de la Vida de John Conway

Paralelización usando MPI y OpenMP con balanceo de carga

Máster Universitario en Ingeniería Informática
M1704 - Programación Paralela

Jaime Iglesias Blanco

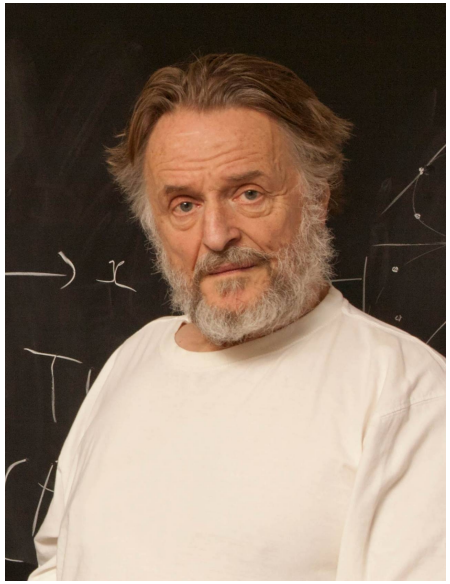
Universidad de Cantabria - Facultad de Ciencias

Mayo 2024

Índice de contenidos

Descripción del problema

El **juego de la vida** es un autómata celular ideado por el matemático británico John Horton Conway.



- Rejilla ortogonal bidimensional «infinita» de celdas cuadradas, cada una de las cuales se encuentra en un estado: «vivo» o «muerto»
- Cada celda interactúa con sus ocho vecinas, que son las celdas directamente adyacentes en horizontal, vertical o diagonal.

Reglas del juego:

- Cualquier célula viva con menos de dos vecinos vivos muere (infrapoblación).
- Cualquier célula viva con más de tres vecinos vivos muere (superpoblación).
- Cualquier célula viva con dos o tres vecinos vivos vive, sin cambios, hasta la siguiente generación.
- Cualquier célula muerta con exactamente tres vecinos vivos volverá a la vida.

Modos de ejecución

UC | Universidad de Cantabria

Normal

Permite realizar el computo del programa sin guardar el resultado.

Imprimir Resultado

Permite visualizar el resultado final por consola.

Imprimir Iteraciones

Permite visualizar el tablero en cada iteración por consola.

Imprimir Fichero

Permite guardar el resultado final en un fichero. Comando: `diff -qs f1 f2`

Debug

Permite mostrar mensajes que muestran el valor de variables relevantes.

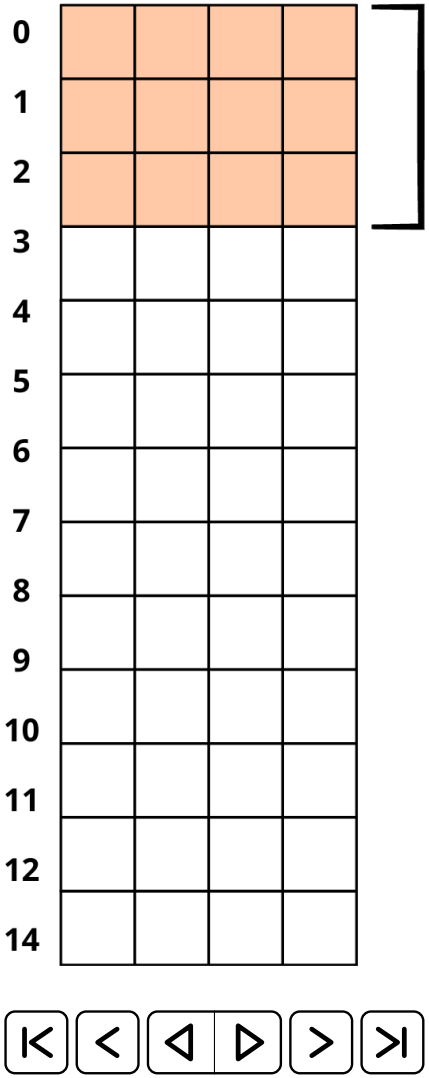
Índice de contenidos

Implementación con MPI

UC | Universidad de Cantabria

División de la carga de trabajo:

- Descomposición del dominio. Particionado de datos:
  - **Bloques 2D por filas**
    - Localidad espacial.
    - Minimiza dependencias.
- Se divide de forma uniforme.
  - Último proceso parte restante.
- Cada uno de los procesos necesita compartir filas adyacentes con el proceso vecino.



Modificación de la estructura de datos:

- El programa original se utiliza un array de punteros a otros arrays. (`board[i][j]`)
- Se utiliza un array para almacenar toda la matriz de forma continua. El acceso a cada elemento se realiza como `board[i * ncols + j]`

Implementación con MPI

Reserva de memoria en cada uno de los procesos:

- El proceso 0, lee los datos de entrada y tiene una estructura de datos lineal con el tablero completo, así como su parte del tablero local.
- El resto de procesos solo tienen en memoria su parte del tablero local.

Envío de los datos a los distintos procesos:

- El envío del tablero se realiza a través de la función `MPI_Scatterv`:

```
1 MPI_Scatterv(board, sendcounts, displs, MPI_CHAR, local_prev,
2              local_rows * local_cols, MPI_CHAR, 0, MPI_COMM_WORLD);
```

- Previamente el proceso 0 ha calculado las 2 estructuras de datos necesarias para configurar el envío:
  - `sendcounts[nprocs]`: Índica la cantidad de datos que enviar.
  - `displs[nprocs]`: Índica el offset o desplazamiento de los datos.
- El origen de los datos es `board` y el destino es `local_prev`, local a cada proceso.

Implementación con MPI

UC | Universidad de Cantabria

Cálculo del offset de inicio y fin de cada proceso:

- El proceso 0 debe computar desde la fila 0 hasta la última fila - 1.
- El último proceso debe computar desde la fila 1 a la última fila .
- El resto de procesos computan desde la fila 1 a la última fila - 1.

Parte de cómputo:

- Se modifica la función `play` para indicar el tamaño del tablero local, así como los offsets.

```
1 play(local_prev, local_next, local_rows, local_cols, start, end);
```

- En la función se recorre el tablero local, por lo tanto se modifican los límites de los bucles:

```
1 for (i = start; i < end; i++)
2     for (j = 0; j < size_cols; j++)
3         a = adjacent_to(board, size_rows, size_cols, i, j)
```

- Lo mismo aplica para la función `adjacent_to`, utilizada para contar las casillas adyacentes «vivas».

Implementación con MPI

UC | Universidad de Cantabria

Intercambio de filas compartidas entre iteraciones:

```
1  if (rank == 0){
2      // Send the end row to the next process
3      MPI_Send(local_prev + (local_rows - 2) * local_cols, local_cols, ..., rank + 1, ...
4      // Receive the last row from the next process
5      MPI_Recv(local_prev + (local_rows - 1) * local_cols, local_cols, ..., rank + 1, ...
6  }
7
8  if (rank > 0 && rank < nprocs - 1) {
9      // Send the start row to the previous process
10     MPI_Send(local_prev + local_cols, local_cols, ..., rank - 1, ..., ...);
11     // Receive the first row from the previous process
12     MPI_Recv(local_prev, local_cols, .., rank - 1, ..., ..., ...);
13
14     // Send the end row to the next process
15     MPI_Send(local_prev + (local_rows - 2) * local_cols, local_cols, ..., rank + 1, ...
16     // Receive the last row from the next process
17     MPI_Recv(local_prev + (local_rows - 1) * local_cols, local_cols, ..., rank + 1, ...
18 }
19
20 if (rank == nprocs - 1){
21     // Send the start row to the previous process
22     MPI_Send(local_prev + local_cols, local_cols, ..., rank - 1, ...
23     // Receive the first row from the previous process
24     MPI_Recv(local_prev, local_cols, ..., rank - 1, ...
25 }
```

Implementación con MPI

Obtención de resultados:

- El envío del tablero se realiza a través de la función `MPI_Gatherv`:

```
1 MPI_Gatherv(local_prev, local_rows * local_cols, MPI_CHAR, board,
2             sendcounts, displs, MPI_CHAR, 0, MPI_COMM_WORLD);
```

- Se utilizan las mismas estructuras de datos que en el envío con `MPI_Scatterv`.
- Posteriormente se imprime por consola o fichero.

Liberación de recursos:

- Se libera la memoria utiliza en cada uno de los procesos.

Instrumentalización del programa:

- Se utiliza la función `MPI_Wtime();` para determinar los instantes de tiempo de ejecución de ciertas partes del programa:
  - Todo el programa: se computa el tiempo total de ejecución (» stderr).
  - Sección de cómputo (ROI): Únicamente mide el tiempo de la sección de cómputo del algoritmo (» stdout).

Índice de contenidos

Implementación con OpenMP

UC | Universidad de Cantabria

- Se paraleliza el computo dentro de un tablero local, para poder ir aplicando las reglas del juego en varias filas de forma paralela.
- Se utiliza una planificación estática ya que todas las filas tienen el mismo número de columnas.

```
1  #ifdef _OPENMP
2  #ifdef LOAD_BALANCING
3  ...
4  #endif
5  #pragma omp parallel for private(i, j, a) schedule(static)
6  #endif
7  for (i = start; i < end; i++)
8      for (j = 0; j < size_cols; j++){
9          a = adjacent_to(board, size_rows, size_cols, i, j);
10         ...
```

- El número de threads se establece mediante la variable de entorno en los jobs.

```
1  export OMP_NUM_THREADS=N
```

Índice de contenidos

Balanceo de carga

UC | Universidad de Cantabria

Estrategia utilizada:

- Balanceo de carga estático de forma distribuida.
  - Solución determinista y verificable. Disminuye las comunicaciones.
  - Se puede adaptar fácilmente y hacerlo centralizado (+1 comunicación).
- Se implementa como una función que permite obtener el número de filas para cada proceso, a partir de una definición del cluster y unos parámetros.

```c
typedef struct
{
    char *hostname[100];
    int niceness;
} ClusterNode;


typedef struct
{
    ClusterNode *nodes;
    int size;
} Cluster;
```

```c
Cluster cluster = {
    .nodes = (ClusterNode[]){
        {.hostname = "n16-80", .niceness = 3},
        {.hostname = "n16-81", .niceness = 3},
        {.hostname = "n16-82", .niceness = 3},
        {.hostname = "n16-83", .niceness = 3},
        {.hostname = "n16-90", .niceness = 1},
        {.hostname = "n16-92", .niceness = 1},
        {.hostname = "n16-93", .niceness = 1},
    },
    .size = 7,
};
```

```c
int *load_balancing(int nprocs, int size, int rank, MPI_Comm comm);
```

Balanceo de carga

Obtención del hostname y su niceness:

- Cada proceso ejecuta la función de balanceo de carga y permite obtener su «niceness» a partir de su hostname.
- Todos los procesos se comparten el «niceness» del resto con la función `MPI_Allgather` (`Gather` + `Broadcast`).

```
1  MPI_Allgather(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, niceness_by_rank, 1, MPI_INT, comm);
```

Cómputo del número de filas a procesar por cada proceso:

- Se obtiene la suma de todos los «niceness» de los procesos en ejecución.
  - No se obtiene de la definición del cluster ya que puede haber más nodos.
- Se asigna mayor número de filas a los procesos con mayor «niceness».
  - Las filas restantes se asignan a los primeros nodos.

```
1  for (int i = 0; i < nprocs; i++)
2      total_niceness += niceness_by_rank[i];
3  for (int i = 0; i < nprocs; i++)
4      rows[i] = (size / total_niceness) * niceness_by_rank[i];
5  for (int i = 0; i < size % total_niceness; i++)
6      rows[i]++;
```

Balanceo de carga

UC | Universidad de Cantabria

Modificaciones en el programa paralelo original:

- La distribución de filas se obtiene mediante la función `load_balancing(...)`.

```
#ifdef LOAD_BALANCING
int *rows_distribution = load_balancing(nprocs, size, rank, MPI_COMM_WORLD);
local_rows = rows_distribution[rank];
local_cols = size;
#else
...
```

- Las estructuras de datos necesarias para configurar el envío del tablero (`MPI_Scatterv` y `MPI_Gatherv`) utilizan directamente la distribución calculada.

```
#ifdef LOAD_BALANCING
sendcounts[0] = local_rows * local_cols;
displs[0] = 0;
for (i = 1; i < nprocs - 1; i++){
    sendcounts[i] = rows_distribution[i] * local_cols + TWO_ADJACENT_ROWS * local_cols;
    displs[i] = displs[i - 1] + sendcounts[i - 1] - TWO_ADJACENT_ROWS * local_cols;
}
sendcounts[nprocs - 1] = rows_distribution[nprocs - 1] * local_cols + local_cols;
displs[nprocs - 1] = displs[nprocs - 2] + sendcounts[nprocs - 2] - TWO_ADJACENT_ROWS * local_cols;
#else
...
```

Ajuste del número de threads

UC | Universidad de Cantabria

- El número de threads se establece manualmente según el #cores del nodo.

```
#ifdef _OPENMP
#ifdef LOAD_BALANCING
system("nproc --all > nproc.txt");
FILE *nproc_file = fopen("nproc.txt", "r");
int nproc;
fscanf(nproc_file, "%d", &nproc);
fclose(nproc_file);
if (nproc == n1680maxCPUs) { // == 20 threads
    omp_set_num_threads(n1680usableCPUs); // 4 threads
} else {
    omp_set_num_threads(nproc);
}
#endif
...
```

- De este modo:
  - En los nodos `n16-[80..83]` se ejecutarán como máximo 4 threads.
  - En los nodos `n16-[90..93]` se ejecutarán tantos como sea posible (4).

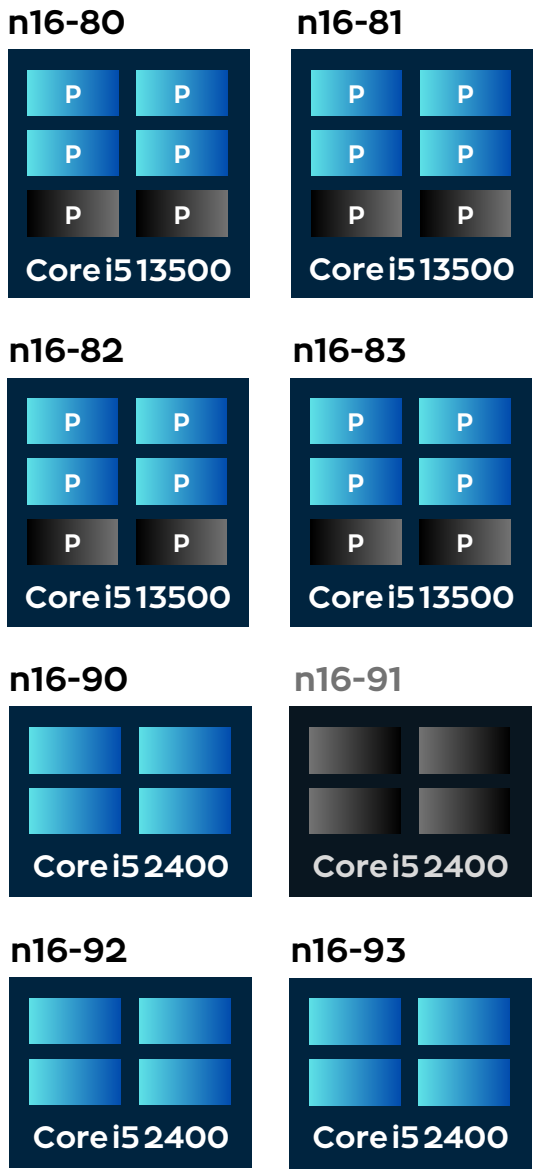
Índice de contenidos

Pruebas de rendimiento

UC | Universidad de Cantabria

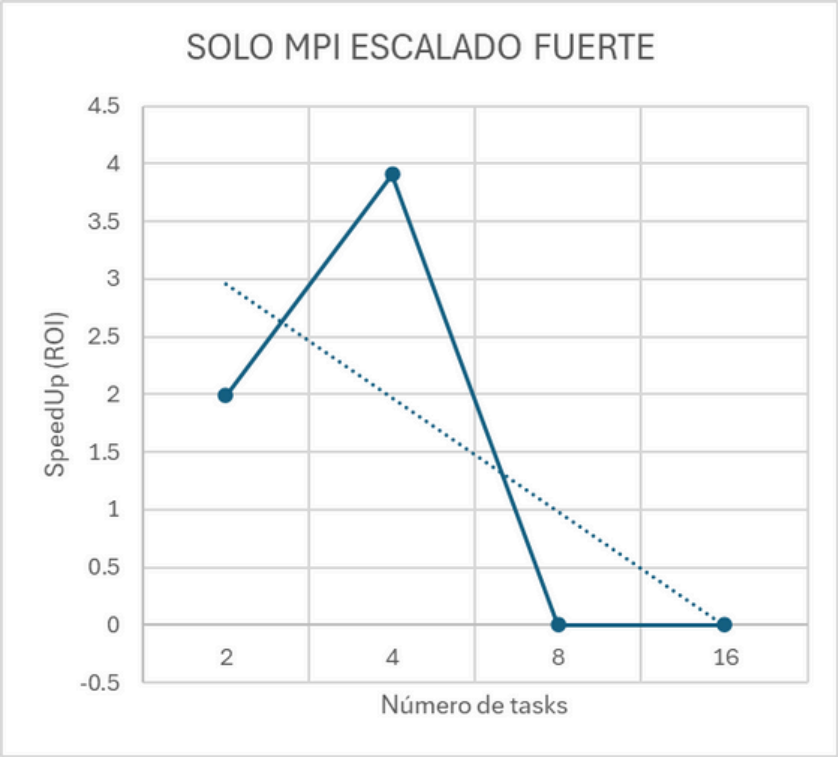
1. **Escalado fuerte:** `n16-[80-83]`
   1) MPI: Con 2, 4, 8, 16 tareas.
   2) MPI + OpenMP: 2 y 4 tareas, 2, 4 threads.
2. **Escalado débil:** `n16-[80-83]`
   1) MPI: Con 4, 9 y 16 tareas, y una carga de trabajo de 10 iteraciones sobre 2000x2000, 3000x3000 y 4000x4000.
3. **Balanceo de carga:** `n16-[80-83,90,92,93]`
   1) Sin algoritmo:
      1) MPI: 28 tareas.
      2) MPI + OpenMP: 7 tareas con 4 threads.
   2) Con algoritmo
      1) MPI: 28 tareas.
      2) MPI + OpenMP: 7 tareas con 4 threads.

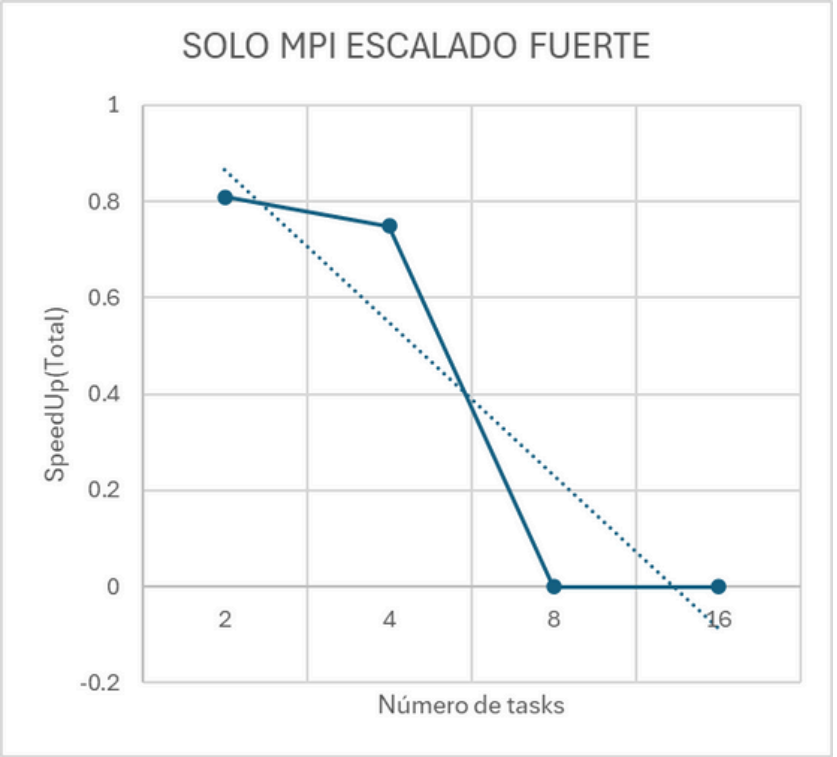
- Todas las pruebas para las que no se indican los datos de entrada se ejecutan con un tablero de 5000x5000 y 10 iteraciones.



Resultados - SpeedUp MPI

- El rendimiento obtenido en el ROI es muy bueno, ya que prácticamente se consigue un x2 o un x4 para 2 y 4 tasks respectivamente.
- En la figura de la derecha, podemos observar el observar el overhead que supone paralelizar un programa en memoria distribuida, si el grado de paralelismo es muy bajo.



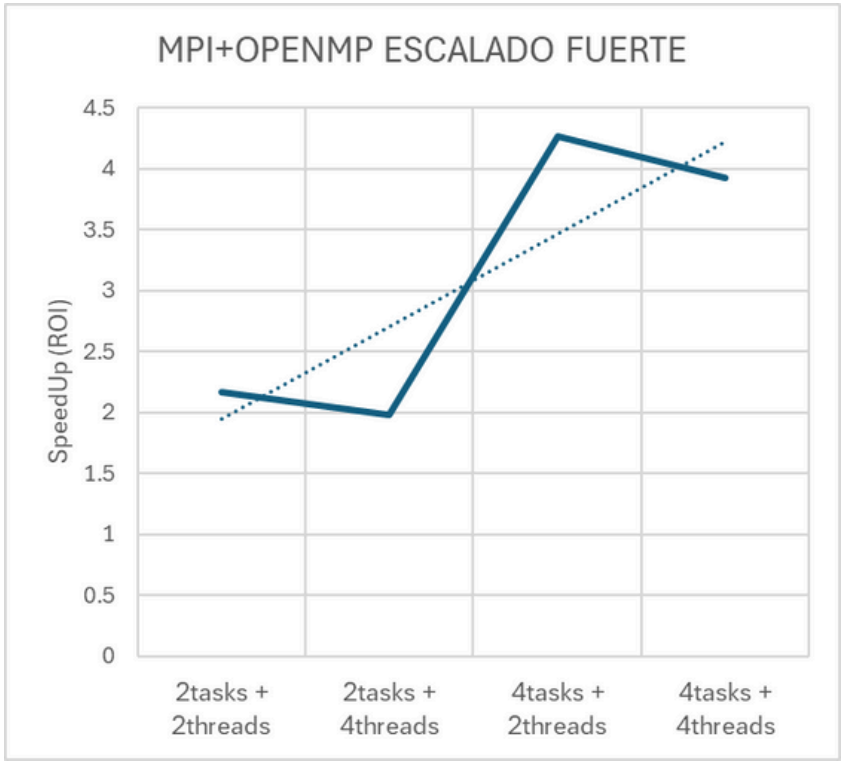


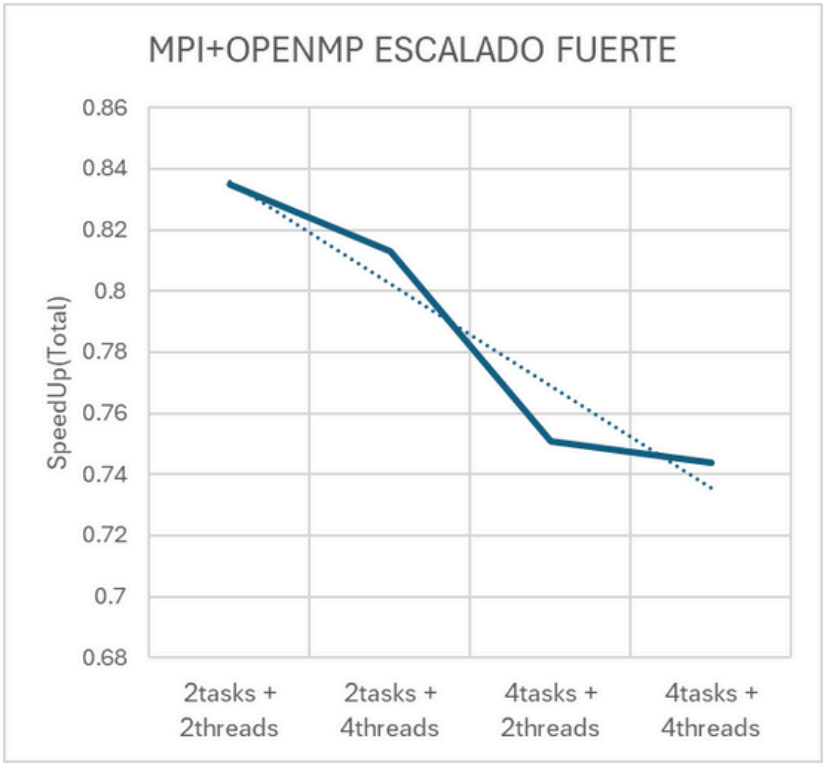
- Los jobs que se lanzan sobre los 8 y 16 nodos no se han podido ejecutar.

Resultados - SpeedUp MPI + Open MP

UC | Universidad de Cantabria

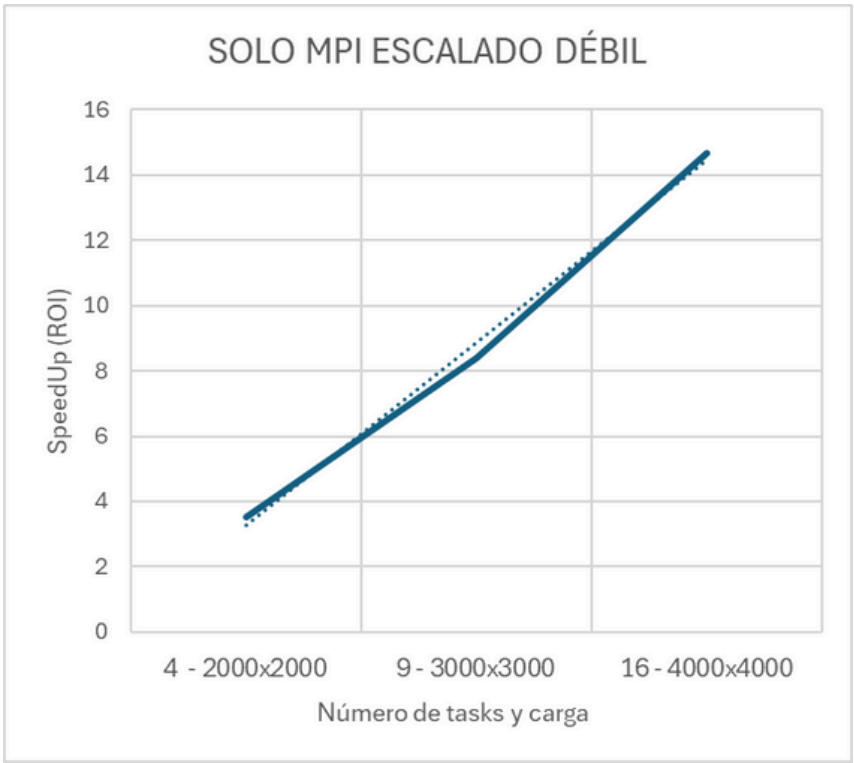
- El rendimiento es inferior respecto al mismo número de divisiones en MPI.
- Las versiones con 2 threads obtienen mejores resultados que con 4 threads.
- Se observa también claramente el overhead que supone realizar un mayor número de divisiones respecto al tiempo de ejecución total.

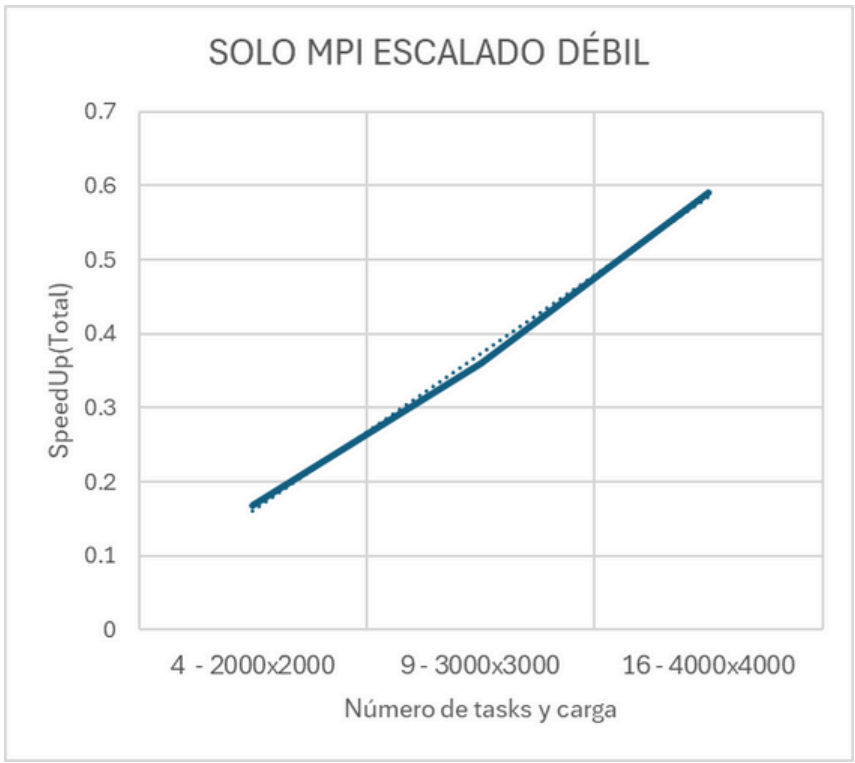




Resultados - SpeedUp MPI

- Se observa una mejora de rendimiento lineal al aumentar la carga de trabajo y procesos de forma proporcional.
- En el total, no se observa mejoría, debido a que la carga de trabajo no es demasiado elevada para el programa secuencial.

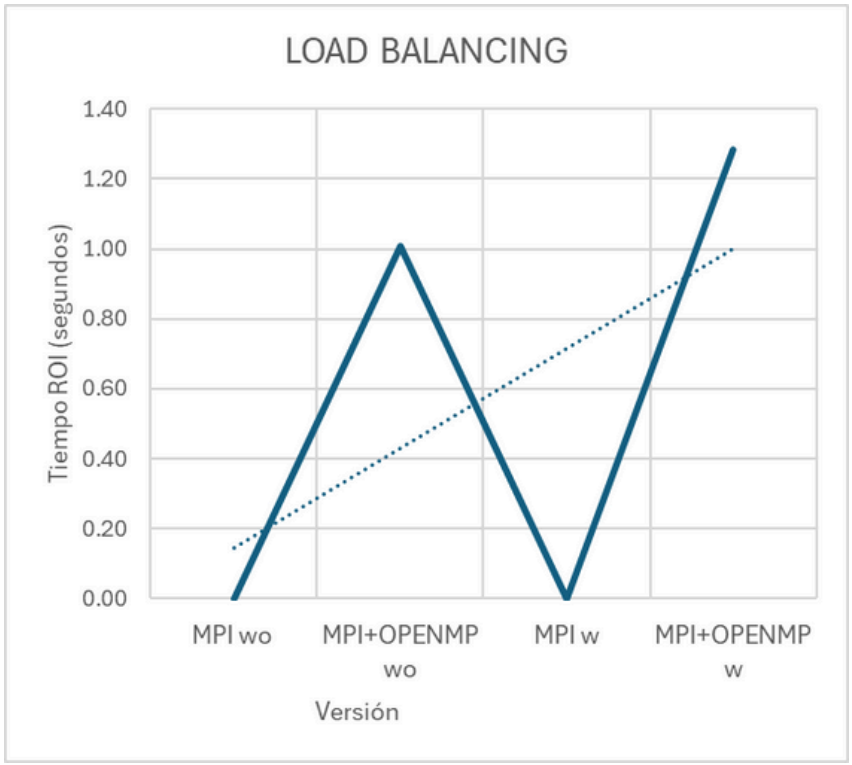


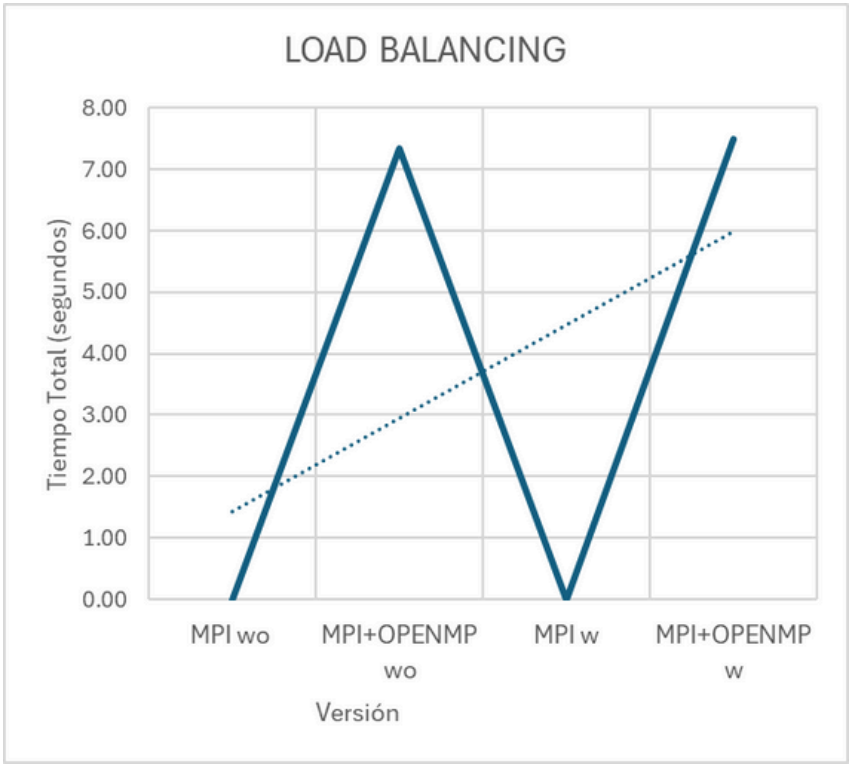


Resultados - Balanceo de carga

UC | Universidad de Cantabria

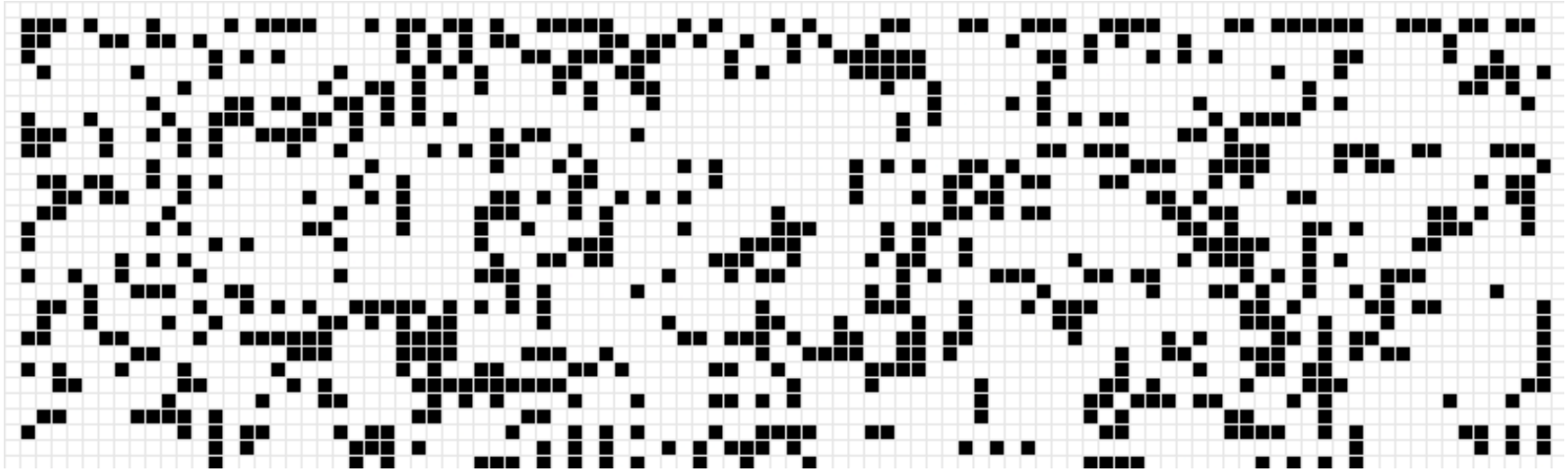
- Los resultados no son correctos, ya que la versión sin balanceo de carga (`wo`) obtiene mejores resultados que la versión ajustada (`w`) al cluster heterogéneo.





- Los jobs que se lanzan sobre los 28 nodos, utilizando únicamente con MPI, no se han podido ejecutar.

Preguntas



Muchas gracias por vuestra atención

Máster Universitario en Ingeniería Informática
M1704 - Programación Paralela

Jaime Iglesias Blanco

Universidad de Cantabria - Facultad de Ciencias

Mayo 2024