

31 DE MARZO DE 2017

TEMA 3. PRÁCTICA MPI

SERVICIOS DE ALTAS PRESTACIONES Y DISPONIBILIDAD

AUTORES:

Jeferson Arboleda Gómez
Pedro Miguel Luzón Martínez
Alberto Mora Plata
Santiago Mora Soler

ÍNDICE

1	Configuración y toma de contacto	9
1.1	Programas MPI Locales	9
1.2	Programas MPI en clúster con TORQUE	9
2	Nivel inicial – Send/Receive.....	11
2.1	Ejercicio 1 – Ordenación de un array de enteros	11
2.2	Ejercicio 2 – Suma de matrices.....	12
2.3	Ejercicio 3 – Suma de matrices.....	13
2.4	Ejercicio 4 – Multiplicación de matrices	14
2.5	Ejercicio 5 – Búsqueda de números primos	17
	Referencias.....	17

ÍNDICE DE FIGURAS

Figura 2.1. Representación de la ordenación de un array usando MPI [1]	11
Figura 2.2. Ejecución del programa del ejercicio 2.1	12
Figura 2.3: Ejecución del programa del ejercicio 2.2	13
Figura 2.4. Ejemplo de suma de matrices	13
Figura 2.5. Ejemplo de suma de matrices con MPI	14
Figura 2.6: Row Major Order.	15
Figura 2.7: uso del fichero ejecutable.	15
Figura 2.8: Impresión de las matrices A y B.....	16
Figura 2.9: Comprobación apartado 1.....	16
Figura 2.10: comprobación del apartado 2.....	16
Figura 2.11: Comprobación del apartado 3.....	16
Figura 2.12. Búsqueda de números primos con MPI.....	17

ÍNDICE DE TABLAS

Tabla 1.1. Principales parámetros PBS.....	10
--	----

1 CONFIGURACIÓN Y TOMA DE CONTACTO

1.1 PROGRAMAS MPI LOCALES

En primer lugar, se debe configurar el demonio `mpd` para que se puedan comunicar los procesos locales a través de paso de mensajes.

```
#Crear el fichero conf del demonio mpd
$ cd $HOME
$ touch .mpd.conf
$ chmod 600 .mpd.conf

#Añadir contraseña propia (passwd) al demonio mpd
$ MPD_SECRETWORD=passwd >> .mpd.conf

#Lanzar el demonio mpd en background
$ Mpd &
```

El siguiente paso es crear un ejecutable, compilarlo y ejecutarlo.

```
#Compilar el programa MPI
$ mpicc helloWorld.c -o helloWorld

#Dar permisos al fichero ejecutable
$ chmod 777 helloWorld

#Ejecutar el programa mpi seleccionando el nº de procesos a ejecutar(-n x)
$ mpiexec -n 4 ./helloWorld
```

Tarea 1. Realiza los pasos anteriores en tu pc, prueba a variar el número de procesos. ¿Cuántos procesos se pueden crear? ¿Cuántos procesos se pueden ejecutar simultáneamente? ¿Qué es avaricia?

1.2 PROGRAMAS MPI EN CLÚSTER CON TORQUE

Un trabajo en TORQUE tiene los siguientes estados:

- **Creación:** En primer lugar, debemos escribir el script que contendrá los parámetros del trabajo que vamos a crear utilizando la sintaxis PBS.
- **Emisión:** Una vez que dispongamos del script utilizaremos el comando `qsub jobscript` para enviar a TORQUE el trabajo en cuestión.
- **Ejecución:** Mientras un trabajo se encuentra en ejecución podemos solicitar su estado a través del comando `qstat`.
- **Finalización:** Cuando un trabajo es completado, siempre y cuando se haya configurado correctamente, se generarán los ficheros `stdout` y `stderr` en el directorio `$HOME`. Si deseamos abortar un proceso antes de que se complete deberemos utilizar el comando `qdel jobId`.

Tabla 1.1. Principales parámetros PBS

Directiva	Descripción
#PBS -N miTrabajo	Asigna un nombre al trabajo.
#PBS -l nodes=4:ppn=2	El número de nodos y procesadores a usar.
#PBS -l walltime=01:00:00	El tiempo máximo que el trabajo puede tardar en ejecutarse.
#PBS -k eo	Mantener los ficheros de error y salida después de la ejecución.
#PBS -j oe	Une los ficheros de error y salida en un solo fichero.
#PBS -m b	Envía un email cuando el trabajo comienza.
#PBS -m e	Envía un email cuando el trabajo finaliza.
#PBS -m a	Envía un email si el trabajo aborta.
#PBS -r n	Indica que el trabajo no debe reejecutarse si falla.
#PBS -V	Exporta todas la variables de entorno al trabajo.

Variable	Descripción
PBS_JOBID	Identificador del trabajo.
PBS_JOBNAME	Nombre del trabajo.
PSB_O_WORKDIR	Directorio de trabajo.

En primer lugar, se deberá configurar el directorio de trabajo de TORQUE, para ello utilizaremos:

```
#Cambiar el valor de la variable del directorio de trabajo (el que queramos)
$ PBS_O_WORKDIR=$HOME/dirTorque

#Acceder al directorio de trabajo
$ cd $PBS_O_WORKDIR

#Una vez en el directorio de trabajo creamos el script del trabajo y le
$ nano jobScript
$ chmod 777 jobScript
```

En el script de trabajo deberemos indicar valores como el nombre del trabajo (-N), los ficheros que generará (-k), número de nodos y procesos por nodo, además del walltime (-l) y finalmente el programa MPI compilado que se ejecutará (en este caso, ./helloWorld). Este es un ejemplo:

```
#!/bin/bash
#PBS -N helloWorld
#PBS -k eo
#PBS -l nodes=1:ppn=8
#PBS -l walltime=00:30:00
cd $PBS_O_WORKDIR
mpiexec ./helloWorld
```

Tarea 2. Lanza el job script creado a TORQUE, localiza el fichero de salida (o) e interprétalo. ¿Hay alguna diferencia con la ejecución local?

Tarea 3. Cambia el fichero jobScript para cambiar el número de nodos a 4. ¿Encuentras ahora algún tipo de diferencia? ¿Qué son avaricia, envidia, soberbia, gula e ira? ¿Qué relación tienen con pk2?

2 NIVEL INICIAL – SEND/RECEIVE

2.1 EJERCICIO 1 – ORDENACIÓN DE UN ARRAY DE ENTEROS

La ordenación de cualquier array conlleva un coste computacional que puede reducirse mediante diferentes algoritmos de ordenación. No obstante, haciendo uso de MPI se puede reducir este coste computacional a $O(N)$. En la Figura 2.1 se pueden observar los pasos que se llevan a cabo para la ordenación en el caso de tener únicamente 2 procesos:

1. El *proceso 0* (“maestro”) divide el array original en dos sub-arrays de 6 elementos.
2. El *proceso 0* envía al *proceso 1* el segundo sub-array para que se encargue de ordenarlo.
3. Ambos procesos ordenan sus sub-arrays.
4. El *proceso 1* envía el sub-array ordenado al *proceso 0*.
5. El *proceso 0* ordena el array resultante a partir de los dos sub-arrays tomando el menor elemento de cada uno. De esta forma sólo se recorre cada elemento del vector una vez.

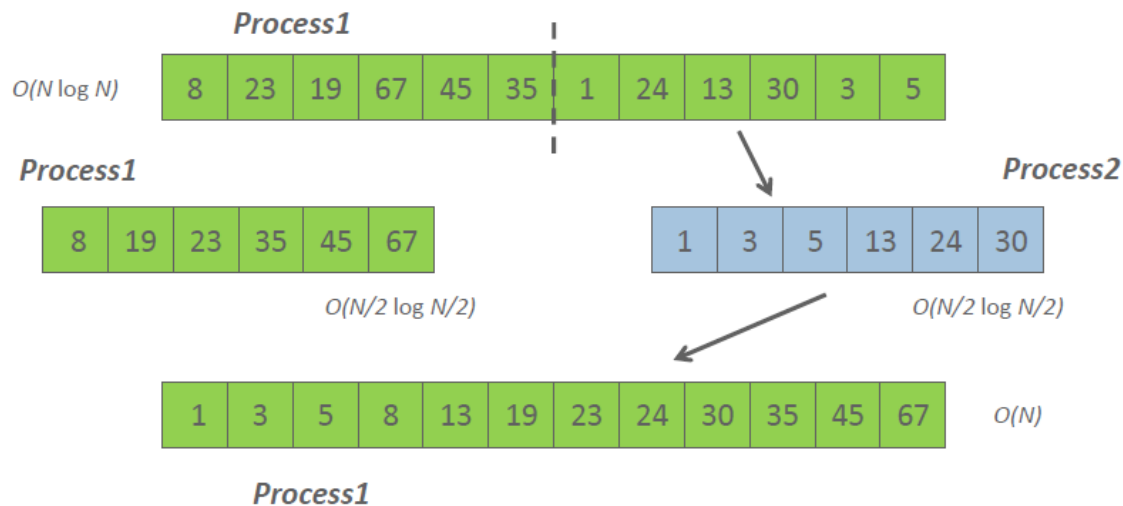


Figura 2.1. Representación de la ordenación de un array usando MPI [1]

El ejercicio que se propone es desarrollar el siguiente programa:

- Recibe un número de procesos (*n_processes*), especificado a través del comando *mpiexec*, y un número de elementos que ordena cada proceso (*n_elem_per_process*), pasado como parámetro.
- La multiplicación de los dos parámetros anteriores da como resultado el tamaño del array ‘global’ a ordenar, cuyos elementos son aleatorios.
- El *proceso 0* envía al resto de procesos su array correspondiente y posteriormente los recibe ordenados para unirlos y ordenarlos.

El esqueleto de este programa puede encontrarse en el fichero *practica_ejercicio_2_1.c*. Únicamente se han de añadir las partes de código que se indican dentro de *main* sin poder modificar otras líneas de código o añadir nuevas variables.

```
$ mpiexec -n 3 ./practica_ejercicio_2_1 2
```

Output:

```
Proceso 0 -  
  Procesos: 3  
  Tamano del vector a ordenar: 6  
Proceso 0 - Array para ordenar: [ 40, 1, 79, 84, 56, 23 ]  
Proceso 0 - Array ordenado: [ 1, 40 ]  
Proceso 0 - Array recibido de 1: [ 79, 84 ]  
Proceso 0 - Array recibido de 2: [ 23, 56 ]  
Proceso 0 - Array global ordenado: [ 1, 23, 40, 56, 79, 84 ]
```

Figura 2.2. Ejecución del programa del ejercicio 2.1

Si se quiere profundizar más acerca de este tema, existen algoritmos de ordenación que pueden paralelizarse, como es el caso del algoritmo *Odd-Even Transposition Sort* [2]. Por ello, se incluye un programa en el fichero *odd_even_sort.c* que hace uso de este algoritmo utilizando MPI.

2.2 EJERCICIO 2 – SUMA DE MATRICES

Este programa deberá realizar la suma de dos matrices de misma dimensión, utilizando para ello varios procesos. Dichos procesos se deberán comunicar con paso de mensajes bloqueante (MPI_Send, MPI_Recv, MPI_Sendrecv y MPI_Sendrecv_replace). El primer proceso generará las matrices, A y B, y repartirá sus filas a los demás (inclusive a sí mismo) para proceder a hacer la suma. Al acabar, los demás procesos enviarán el resultado de vuelta al primer proceso para que las junte e imprima el resultado.

En particular, cada proceso sumará pares de filas, pertenecientes a las matrices A y B respectivamente. Dichas filas se repartirán entre el número de procesos usados en la ejecución. Un proceso puede llegar a operar más de un par de filas, si así surgiera a la hora del reparto. Es decir, si se usan 2 procesos y las matrices tienen 4 filas, cada proceso trabajará con 2 pares de filas (2 filas de A y 2 filas de B) que se sumarán para producir 2 filas en total.

El programa tomará como parámetros las dimensiones de las matrices. Al ser iguales, se requerirán dos parámetros: el número de filas y el número de columnas.

Un ejemplo de ejecución del programa sería el siguiente (Figura 2.3). En él se suman dos matrices 4x6 usando dos procesos. El esqueleto para empezar a programar en él se encuentra en el fichero *practica_ejercicio_2_2.c*.

Nota: sería muy conveniente que el número de filas sea divisible entre el número de procesos a utilizar, para no complicar la implementación. Nivel avanzado – Operaciones Colectivas

```

mcalu0011@avaricia:~/mpi$ mpiexec -n 2 ./practica_ejercicio_2_2 4 6
Matriz A:
83      77      93      86      49      62
90      63      40      72      11      67
82      62      67      29      22      69
93      11      29      21      84      98
Matriz B:
86      15      35      92      21      27
59      26      26      36      68      29
30      23      35      2       58      67
56      42      73      19      37      24
Tarea para el proceso 0:
[83 77 93 86 49 62 90 63 40 72 11 67 86 15 35 92 21 27 59 26 26 36 68 29 ]
Tarea para el proceso 1:
[82 62 67 29 22 69 93 11 29 21 84 98 30 23 35 2 58 67 56 42 73 19 37 24 ]
Resultado del proceso 0:
[169 92 128 178 70 89 149 89 66 108 79 96 ]
Resultado del proceso 1:
[112 85 102 31 80 136 149 53 102 40 121 122 ]
Matriz A + B:
169      92      128      178      70      89
149      89      66      108      79      96
112      85      102      31      80      136
149      53      102      40      121      122

```

Figura 2.3: Ejecución del programa del ejercicio 2.2

2.3 EJERCICIO 3 – SUMA DE MATRICES

Este programa deberá realizar la suma de dos matrices de misma dimensión, utilizando para ello operaciones de computación colectiva. A continuación, se muestra la definición de suma de matrices, así como un ejemplo de suma (Figura 2.4).

$$\begin{aligned}
 \mathbf{A} + \mathbf{B} &= (\mathbf{a}_{ij} + \mathbf{b}_{ij}) \\
 \mathbf{A} &= \begin{pmatrix} 2 & 0 & 1 \\ 3 & 0 & 0 \\ 5 & 1 & 1 \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} 1 & 0 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 0 \end{pmatrix} \\
 \mathbf{A} + \mathbf{B} &= \begin{pmatrix} 2+1 & 0+0 & 1+1 \\ 3+1 & 0+2 & 0+1 \\ 5+1 & 1+1 & 1+0 \end{pmatrix} = \begin{pmatrix} 3 & 0 & 2 \\ 4 & 2 & 1 \\ 6 & 2 & 1 \end{pmatrix} \\
 \mathbf{A} - \mathbf{B} &= \begin{pmatrix} 2-1 & 0-0 & 1-1 \\ 3-1 & 0-2 & 0-1 \\ 5-1 & 1-1 & 1-0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & -2 & -1 \\ 4 & 0 & 1 \end{pmatrix}
 \end{aligned}$$

Figura 2.4. Ejemplo de suma de matrices

Teniendo eso en cuenta, completa el código del programa `3_1_matrix_addition_sol.c` para hallar la suma de dos matrices que ya están generadas en el programa. Si los parámetros del programa (tamaño de matrices y semilla) no se modifican, la salida debería ser como la de la Figura 2.5.

```
jag ... > repos > airq > mpi-examples ./mpi_exec.sh practica/3_1_matrix_addition_sol.c 4
matriz a
3.000000 6.000000 7.000000 5.000000 3.000000
5.000000 6.000000 2.000000 9.000000 1.000000
2.000000 7.000000 0.000000 9.000000 3.000000
6.000000 0.000000 6.000000 2.000000 6.000000
1.000000 8.000000 7.000000 9.000000 2.000000
0.000000 2.000000 3.000000 7.000000 5.000000
9.000000 2.000000 2.000000 8.000000 9.000000
7.000000 3.000000 6.000000 1.000000 2.000000
matriz b
0.000000 9.000000 3.000000 8.000000 4.000000
1.000000 3.000000 0.000000 8.000000 5.000000
8.000000 3.000000 4.000000 7.000000 3.000000
3.000000 6.000000 0.000000 1.000000 2.000000
7.000000 7.000000 1.000000 6.000000 6.000000
8.000000 7.000000 0.000000 4.000000 6.000000
6.000000 4.000000 6.000000 1.000000 2.000000
2.000000 4.000000 7.000000 4.000000 2.000000
matriz c = a + b
3.000000 15.000000 10.000000 13.000000 7.000000
6.000000 9.000000 2.000000 17.000000 6.000000
10.000000 10.000000 4.000000 16.000000 6.000000
9.000000 6.000000 6.000000 3.000000 8.000000
8.000000 15.000000 8.000000 15.000000 8.000000
8.000000 9.000000 3.000000 11.000000 11.000000
15.000000 6.000000 8.000000 9.000000 11.000000
9.000000 7.000000 13.000000 5.000000 4.000000
```

Figura 2.5. Ejemplo de suma de matrices con MPI

2.4 EJERCICIO 4 – MULTIPLICACIÓN DE MATRICES

En este ejercicio, se desea llevar a cabo el producto de dos matrices (A y B) de forma paralela y guardar el resultado en otra matriz (C). Para ello necesitaremos declarar las tres matrices:

- **A**: de dimensiones m filas y n columnas. $A := (a_{ij})_{m \times n}$ y $B := (b_{ij})_{n \times p}$
- **B**: de dimensiones n filas y p columnas.
- **C**: de dimensiones m filas y p columnas. $C = AB := (c_{ij})_{m \times p}$

Es muy importante tener en cuenta las dimensiones de las matrices a la hora de diseñar el paralelismo existente en nuestro código.

La operación que deberemos realizar para obtener cada elemento de la matriz resultado $C[i,j]$ se suele resumir de forma sencilla como “filas por columnas”. En notación matemática, esta es la fórmula:

$$c_{ij} = \sum_{r=1}^n a_{ir} b_{rj}$$

Si observamos la operación de forma global, esta es la representación idónea:

$$C = AB = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & \cdots & b_{1p} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{np} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + \cdots + a_{1n}b_{n1} & \cdots & a_{11}b_{1p} + \cdots + a_{1n}b_{np} \\ \vdots & \ddots & \vdots \\ a_{m1}b_{11} + \cdots + a_{mn}b_{n1} & \cdots & a_{m1}b_{1p} + \cdots + a_{mn}b_{np} \end{pmatrix}$$

Todas las matrices se han declarado como un vector de elementos de tamaño filas x columnas, de esta forma trabajaremos con vectores en vez de con matrices, facilitando así el código paralelo. Antes de diseñar el paralelismo vamos a prestar especial atención a la Figura 2.6, la cual muestra cómo se van a ordenar todas nuestras matrices según el método de ordenación que se ha elegido: Row Major Order.

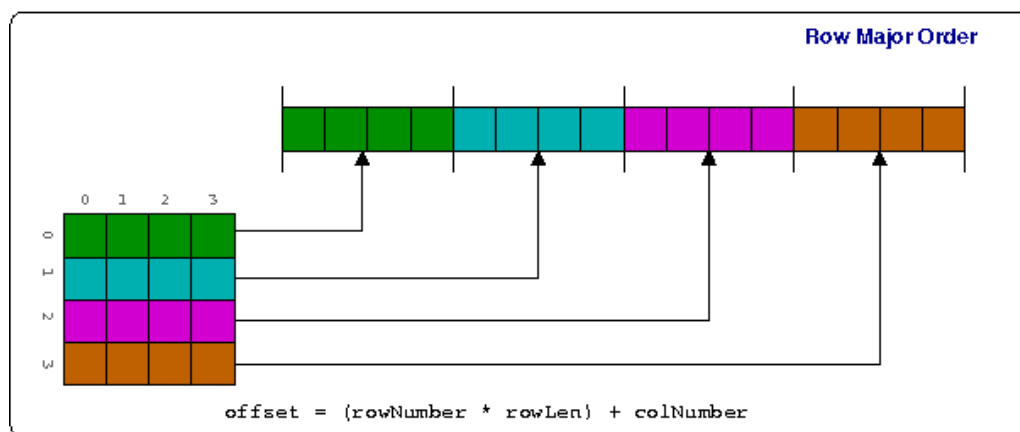


Figura 2.6: Row Major Order.

Una vez que tenemos en cuenta estas explicaciones, se debe completar el esqueleto que podemos encontrar en el fichero “practica_ejercicio_3_2.c”. Para ir probando si vamos por buen camino se recomienda realizar un apartado, compilar y ejecutar para comprobar la salida. Las dimensiones serán las siguientes:

- **dim_m** → nº de procesos.
- **dim_n** → pasado en el primer parámetro.
- **dim_p** → pasado en el segundo parámetro.

Los apartados que se han definido en la práctica son los siguientes:

- **Apartado 1:** Envía la matriz B a todos los procesos, imprímela con cada proceso.
- **Apartado 2:** Envía una fila de la matriz A a cada proceso e imprime el buffer recibido para comprobar que funciona correctamente.
- **Apartado 3:** Calcula la matriz C e imprímela por pantalla con el proceso 1.

El uso del ejecutable que se creará tras compilar el archivo es el que se muestra en la Figura 2.7.

```
>> A( m x n ) * B( n x p ) = C( m x p )
>> Usage: mpiexec -n [numProcesses = dimM] ./ejercicio4 [dimN] [dimP]
```

Figura 2.7: uso del fichero ejecutable.

El esqueleto, ya genera e imprime las 3 matrices con las que deberemos trabajar, en primer lugar, compila el fichero y comprueba que imprime por pantalla las dos matrices que se muestran en la Figura 2.8. Para ello utiliza 4 procesos, y pasa como parámetros 5 y 3.

```

mcalu0110@avaricia:~/Tema3$ mpiexec -n 4 ./practica_ejercicio_3_2_sol 5 3

>> Procesador avaricia, proceso 0 de 4 procesos existentes.
>> Tengo que multiplicar dos matrices de dimensiones: A ( 4 x 5 ) y B ( 5 x 3 )

>> A matrix is...

| 3.155979      2.849434      2.406013      4.841266      3.757932      |
| 0.537027      5.702739      9.700047      5.154223      4.295290      |
| 4.081150      1.501347      5.865514      6.316350      6.138596      |
| 4.113392      1.070919      8.716255      2.643858      6.215432      |

>> B matrix is...

| 6.707429      3.580329      2.083556      |
| 5.341752      3.845116      8.445560      |
| 8.835520      4.615307      6.505120      |
| 7.724178      4.963475      9.661098      |
| 0.573612      7.369488      4.502364      |

```

Figura 2.8: Impresión de las matrices A y B.

Para comprobar el apartado 1, observa que todos los procesos reciben la matriz B (ver Figura 2.9).

```

[0] >> B matrix is...

| 6.707429      3.580329      2.083556      |
| 5.341752      3.845116      8.445560      |
| 8.835520      4.615307      6.505120      |
| 7.724178      4.963475      9.661098      |
| 0.573612      7.369488      4.502364      |

```

Figura 2.9: Comprobación apartado 1.

Para comprobar el apartado 2, presta atención a lo que recibe cada proceso. A cada uno le debemos enviar la columna de la matriz A correspondiente (ver Figura 2.10).

```

>> 2 received buffer is...

4.081150 1.501347 5.865514 6.316350 6.138596

```

Figura 2.10: comprobación del apartado 2.

Para comprobar el apartado 3, comprueba que tu matriz C se corresponde con la de la Figura 2.11.

```

>> C matrix is...

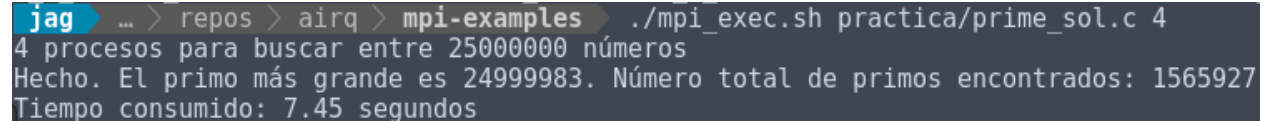
| 97.198242      85.083870      109.983650      |
| 162.045609     125.856064      6.138596      |
| 139.528488     124.045212      6.138596      |
| 134.310394     118.000572      6.138596      |

```

Figura 2.11: Comprobación del apartado 3

2.5 EJERCICIO 5 – BÚSQUEDA DE NÚMEROS PRIMOS

En este ejercicio se deben utilizar operaciones de computación colectiva para calcular números primos. El programa (incompleto) se encuentra en *prime.c*, hay que completarlo para obtener una salida parecida a la Figura 2.12.

A terminal window showing the execution of an MPI program. The prompt is 'jag' followed by a blue arrow. The directory path is '... > repos > airq > mpi-examples'. The command is './mpi_exec.sh practica/prime_sol.c 4'. The output is: '4 procesos para buscar entre 25000000 números', 'Hecho. El primo más grande es 24999983. Número total de primos encontrados: 1565927', and 'Tiempo consumido: 7.45 segundos'.

```
jag ... > repos > airq > mpi-examples ./mpi_exec.sh practica/prime_sol.c 4
4 procesos para buscar entre 25000000 números
Hecho. El primo más grande es 24999983. Número total de primos encontrados: 1565927
Tiempo consumido: 7.45 segundos
```

Figura 2.12. Búsqueda de números primos con MPI

En este ejercicio, el algoritmo para calcular números primos no es lo más importante, por lo que se da una función auxiliar ya hecha, basta con comprender cómo funciona para realizar el ejercicio, sin entrar en sus detalles.

REFERENCIAS

- [1] P. B. T. H. K. R. W. B. y X. Z. , «Introduction to MPI. Guides | MPICH,» 6 Junio 2014. [En línea]. Available: <https://www.mpich.org/documentation/guides/>. [Último acceso: Marzo 2017].
- [2] University of Mary Washington, «Parallel sorting,» [En línea]. Available: <http://cs.umw.edu/~finlayson/class/fall14/cpsc425/notes/18-sorting.html>. [Último acceso: Marzo 2017].