Análisis de Rendimiento de Paralelización con MPI (Contar)

Jaime Hernández Delgado

Curso 2024/25

1. Descripción del Código

El código analizado (contar_mpi.c) implementa un algoritmo para contar ocurrencias de un número específico en un vector de enteros aleatorios. Las principales características del programa son:

- Generación de un vector de enteros aleatorios de tamaño variable.
- Búsqueda secuencial y paralela del número 10 (NUMBUSCADO) en dicho vector.
- Repetición del proceso de búsqueda 100.000 veces para obtener mediciones significativas.
- Distribución del trabajo entre múltiples procesos usando MPI.
- Medición de tiempos de ejecución secuencial y paralelo.
- Cálculo de métricas de rendimiento: speedup y eficiencia.

1.1. Estructura del algoritmo paralelo

El programa sigue estos pasos principales:

1. Inicialización del entorno MPI y obtención del identificador de proceso y número total de procesos.
2. El proceso raíz (rank 0) lee el tamaño del vector y lo comunica al resto de procesos.
3. El proceso raíz genera el vector con números aleatorios.
4. Ejecución secuencial (solo en el proceso raíz) para establecer una línea base de comparación.
5. Distribución del vector entre todos los procesos participantes.
6. Ejecución paralela: cada proceso cuenta ocurrencias en su porción del vector.
7. Recolección de resultados parciales y cálculo del resultado global.
8. Cálculo y presentación de las métricas de rendimiento.

1.2. Fragmento de código relevante

A continuación se muestra un fragmento del código donde se implementa la distribución del trabajo:

```
// Distribucion del vector entre los procesos
int elemsPorProc = tamanyo / nprocs;
int elemsSobrantes = tamanyo % nprocs;
int elemsLocales = (myrank == 0) ? elemsPorProc + elemsSobrantes : elemsPorProc;
int *vectorLocal = (int *)malloc(elemsLocales * sizeof(int));

// Envio del vector a todos los procesos
if (myrank == 0) {
    // El proceso 0 se queda con su parte
    for (int i = 0; i < elemsLocales; i++) {
        vectorLocal[i] = vector[i];
    }
    // Envia al resto de procesos
    int offset = elemsLocales;
    for (int i = 1; i < nprocs; i++) {
        MPI_Send(&vector[offset], elemsPorProc, MPI_INT, i, 0, MPI_COMM_WORLD);
        offset += elemsPorProc;
    }
} else {
    MPI_Recv(vectorLocal, elemsPorProc, MPI_INT, 0, 0, MPI_COMM_WORLD,
    MPI_STATUS_IGNORE);
}
```

Listing 1: Distribución del vector entre procesos

2. Resultados Experimentales

2.1. Tabla de resultados

La siguiente tabla muestra los resultados obtenidos al ejecutar el programa con diferentes configuraciones:

Tamaño	Procesos	tsec (s)	tpar (s)	speedUp	Eficiencia (%)
10000	2	0.823	0.523	1.57	78.68
10000	4	0.80671	0.32245	2.50	62.55
10000	6	0.89288	0.165	5.41	90.19
10000	8	1.35226	0.254	5.32	66.55
20000	2	1.55801	1.0256	1.52	75.96
20000	4	1.29824	0.529303	2.45	61.32
20000	6	1.44479	0.2568	5.63	93.77
20000	8	1.54875	0.312455	4.96	61.96
30000	2	2.09553	1.1562	1.81	90.62
30000	4	2.09501	0.548	3.82	95.58
30000	6	2.09622	0.3568	5.88	97.92
30000	8	2.18628	0.446625	4.90	61.19
40000	2	2.86025	1.694312	1.69	84.41
40000	4	2.84156	1.115016	2.55	63.71
40000	6	2.83997	0.910159	3.12	52.01
40000	8	2.83945	0.818236	3.47	43.38
50000	2	3.49992	1.97478	1.77	88.62
50000	4	3.53169	1.270855	2.78	69.47
50000	6	3.48102	1.01195	3.44	57.33
50000	8	3.47797	1.219327	2.85	35.65
60000	2	4.06523	2.303202	1.77	88.25
60000	4	4.02335	1.404457	2.86	71.62
60000	6	3.88685	1.116054	3.48	58.04
60000	8	4.12098	1.416883	2.91	36.36
70000	2	4.7641	2.614279	1.82	91.12
70000	4	4.82936	1.57106	3.08	76.85
70000	6	4.66822	1.20245	3.88	64.70
70000	8	4.7596	1.041892	4.57	57.10
80000	2	5.7803	2.926225	1.91	95.31
80000	4	5.42746	1.716538	3.16	79.05
80000	6	5.32012	1.308429	4.07	67.77
80000	8	5.42965	1.696572	3.20	40.00
90000	2	5.94497	3.19937	1.86	92.91
90000	4	6.08996	1.850767	3.29	82.26
90000	6	5.97192	1.387162	4.31	71.75
90000	8	6.05025	1.841894	3.28	41.06
100000	2	6.70461	3.518477	1.91	95.28
100000	4	6.59994	2.011263	3.28	82.04
100000	6	6.70073	1.517496	4.42	73.59
100000	8	6.93489	1.996337	3.47	43.42

Cuadro 1: Rendimiento del algoritmo de conteo en función del tamaño del vector y número de procesadores.

2.2. Análisis de los resultados

A partir de los datos obtenidos, podemos analizar varios aspectos del rendimiento:

2.2.1. Tiempo de ejecución

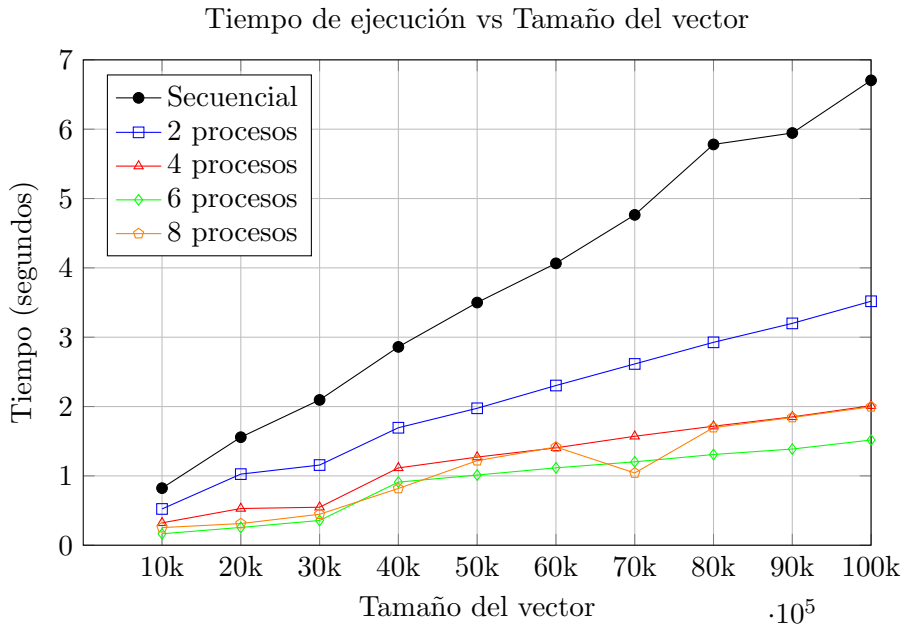

Figura 1: Comparación de tiempos de ejecución para diferentes configuraciones.

Los tiempos paralelos son consistentemente menores que los secuenciales, demostrando el beneficio de la paralelización. La reducción de tiempo es más significativa conforme aumenta el número de procesadores, aunque no de manera proporcional.

2.2.2. Speedup

El speedup representa cuántas veces más rápido es el algoritmo paralelo respecto al secuencial. Se calcula como:

$$Speedup = \frac{t_{sec}}{t_{par}} \tag{1}$$

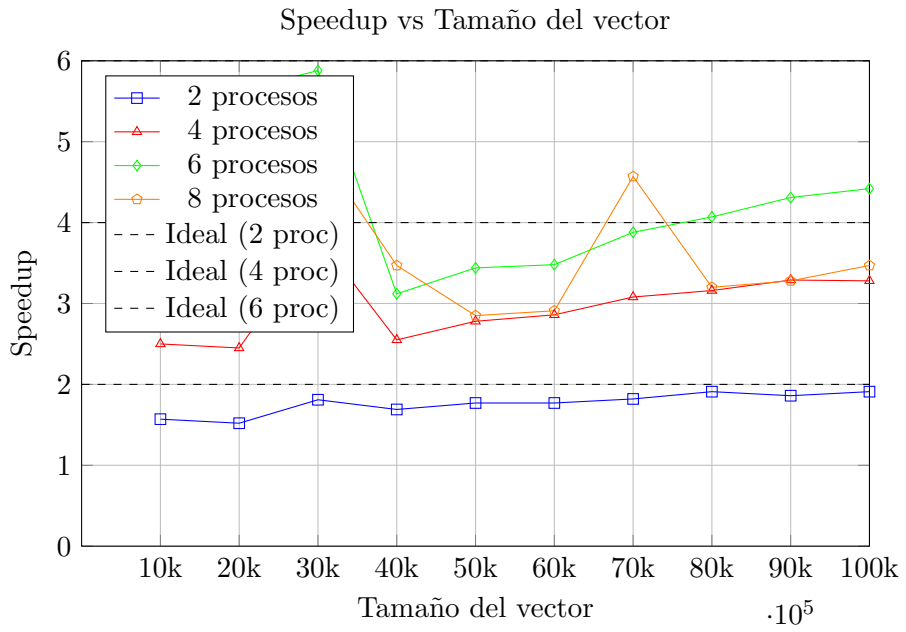

Figura 2: Speedup obtenido para diferentes configuraciones.

Los resultados muestran que:

- Con 2 procesadores, el speedup se mantiene entre 1.5 y 1.9, cercano al valor ideal de 2.
- Con 4 procesadores, el speedup oscila entre 2.5 y 3.8, alejándose del valor ideal de 4.
- Con 6 procesadores, se observan valores entre 3.1 y 5.9, con algunos casos cercanos al ideal de 6.
- Con 8 procesadores, el speedup varía entre 2.8 y 5.3, significativamente por debajo del ideal.

Es notable que con 6 procesadores se obtienen algunos de los mejores resultados relativos, y que aumentar a 8 procesadores a menudo no mejora el rendimiento e incluso puede degradarlo en algunos casos.

2.2.3. Eficiencia

La eficiencia mide el grado de aprovechamiento de los recursos de cómputo. Se calcula como:

$$Eficiencia = \frac{Speedup}{número\ de\ procesadores} \times 100\,\% \tag{2}$$

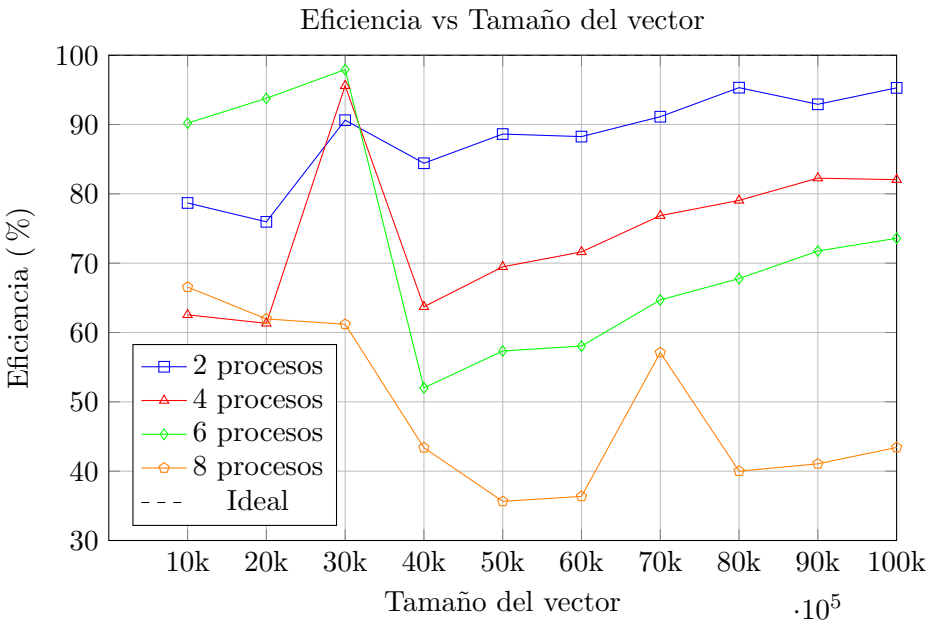

Figura 3: Eficiencia obtenida para diferentes configuraciones.

El análisis de la eficiencia muestra:

- La configuración con 2 procesadores mantiene una eficiencia alta, generalmente por encima del 75 % y llegando a superar el 95 % en los tamaños mayores.
- Con 4 procesadores, la eficiencia oscila entre el 60 % y el 95 %, tendiendo a mejorar con tamaños de vector mayores.
- Con 6 procesadores hay un comportamiento interesante: para tamaños pequeños (hasta 30.000) la eficiencia es excelente (90-98 %), pero cae significativamente para tamaños mayores (50-70 %).
- La configuración con 8 procesadores muestra la peor eficiencia, especialmente en tamaños medios y grandes, rondando el 35-45 % en muchos casos.

3. Conclusiones

Del análisis realizado, podemos extraer las siguientes conclusiones:

1. La paralelización del algoritmo de conteo mediante MPI proporciona mejoras significativas de rendimiento respecto a la versión secuencial, con speedups que llegan a aproximarse al ideal en algunas configuraciones.
2. El rendimiento no escala linealmente con el número de procesadores, observándose una disminución de la eficiencia conforme aumenta el número de estos, especialmente al pasar de 6 a 8 procesadores.
3. La configuración óptima para este problema específico parece estar en el rango de 4 a 6 procesadores, dependiendo del tamaño del problema.
4. Los tamaños de problema más pequeños (10.000-30.000) muestran un comportamiento particularmente eficiente con 6 procesadores, alcanzando eficiencias cercanas al 100 %.

5. El rendimiento con 8 procesadores es inconsistente y generalmente inferior al esperado, lo que sugiere que factores como la contención de recursos y la sobrecarga de comunicación están afectando negativamente.

En conclusión, el algoritmo de conteo implementado con MPI demuestra la efectividad de la paralelización para mejorar el rendimiento en problemas de búsqueda intensiva, aunque con las limitaciones inherentes a la escalabilidad en sistemas distribuidos.