# Implementación y análisis de rendimiento de un algoritmo paralelo de multiplicación de matrices en 2,5D

Federico Sanchez[1], Ana Laura Molina[1], Nelson Rodriguez[1], María Murazzo[1]

[1] Departamento de Informática, Facultad de Ciencias Exactas, Físicas y Naturales, Universidad Nacional de San Juan. San Juan, Argentina.
fgsanchez@unsj-cuim.edu.ar, amolina@unsj-cuim.edu.ar, nelson@iinfo.unsj.edu.ar, marite@unsj-cuim.edu.ar

**Resumen.** En este trabajo se propone una implementación 2D-compatible del algoritmo paralelo de Cannon para multiplicación de matrices en su versión 2,5D. Dicha implementación fue realizada utilizando una distribución 2D de matrices en una grilla 2,5D de procesos. El objetivo consiste en evaluar el rendimiento de esta implementación en comparación con un algoritmo paralelo 1D previamente diseñado. Para tal fin, los desarrollos fueron ejecutados sobre un cluster homogéneo, conformado por 8 nodos, para diversos tamaños de problema. Los resultados obtenidos confirman que esta nueva alternativa 2D-compatible supera, en términos de rendimiento, a la solución 1D. La nueva implementación arroja una reducción del 6%, como mínimo, del tiempo de ejecución para todos los escenarios estudiados.

**Palabras clave:** programación paralela, multiplicación de matrices, algoritmo 2,5D.

## 1 Introducción

En el algoritmo clásico de multiplicación de matrices ideado por Strassen [1] cada una de las $n^3$ multiplicaciones puede realizarse de forma independiente. Es allí donde surge la sencillez de paralelizar su ejecución de forma balanceada. Debido a que el volumen de cómputo no puede ser reducido, se debe trabajar en optimizar la comunicación y el almacenamiento de datos.

Tras la investigación realizada en [2], donde se comparó el rendimiento obtenido al ejecutar diversas estrategias de multiplicación de matrices en arquitecturas distribuidas, se detectó la necesidad de paralelizar las comunicaciones, a fin de reducir el tiempo en el que las unidades de procesamiento disponibles permanecían ociosas y disminuir el costo de la paralelización.

Una alternativa estudiada, que permite paralelizar comunicaciones, es la presentada en [3]. En esta solución 2,5D, basada en el algoritmo de Cannon, los procesos no se

consideran como entidades aisladas, sino incluidos en múltiples grillas compuestas por un grupo de estos. Durante la ejecución, cada proceso multiplica un bloque 2D de matriz A con un bloque 2D de matriz B, en lugar de bloques unidimensionales (filas).

En este trabajo, se ha desarrollado una implementación 2D-compatible al algoritmo de multiplicación de matrices 2,5D, y se plantea como objetivo analizar su rendimiento en comparación con una implementación 1D desarrollada en trabajos previos. Para llevar a cabo dicho análisis, se ejecutaron diversas pruebas sobre los distintos algoritmos en diferentes escenarios definidos.

El presente artículo se organiza de la siguiente manera. En la sección 2 se describen soluciones desarrolladas previamente como estrategias para efectuar la multiplicación de matrices. En la sección 3 se presenta un resumen del algoritmo de Cannon en su versión 2,5D. En la sección 4 se incluye una descripción del desarrollo generado. En la sección 5 se enuncian las condiciones de hardware y software bajo las cuales se llevó a cabo el estudio de rendimiento. En la sección 6 se muestra gráficamente la reducción porcentual del tiempo de ejecución obtenida en los diferentes escenarios considerados, la cual da origen a las conclusiones, ofrecidas en la sección 7.

## 2 Trabajos Previos

De forma previa a la implementación presentada en este trabajo, se analizaron diversas estrategias de distribución de datos para efectuar multiplicaciones de matrices de gran tamaño [2].

Para la definición de dichas estrategias se propuso descomponer el cómputo considerando que, como cada elemento de C requiere la misma cantidad de cálculo, se puede balancear la carga mediante una distribución uniforme de bloques de C entre los p procesos disponibles.

En ambas soluciones, se realizó una distribución de datos unidimensional (filas) y uniforme. Cada proceso es responsable de calcular un bloque de n / p filas de C (siendo n la cantidad de filas y columnas de la matriz) y para ello, necesita obtener las correspondientes n / p filas de la matriz A y toda la matriz B.

Sin embargo, el número de filas de las matrices A y B no siempre es múltiplo de la cantidad de procesos disponibles. En estos casos, luego de hacer una distribución uniforme queda un número de filas de A sin distribuir y una porción de cálculo sin realizarse. Para resolver este inconveniente, se desarrollaron dos alternativas en las cuales se aborda el problema de las filas excedentes de distinta manera.

- Alternativa 1: las filas excedentes (en caso de existir) serán asignadas una a cada proceso hasta agotarlas, quedando así algunos procesos con una fila más para calcular que otros. En la Fig. 1. se puede observar que el proceso P0 trabaja sobre una fila más que los restantes.

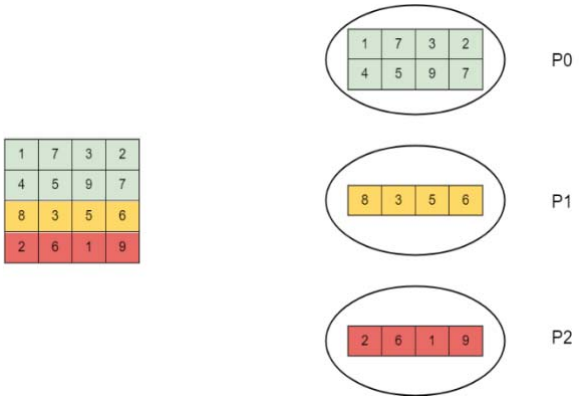


**Fig.1.** Distribución de filas de la matriz A entre p procesos realizada bajo la alternativa 1

- Alternativa 2: si el número de filas de la matriz A no es múltiplo de la cantidad de procesos disponibles, a dicha matriz se le agregan tantas filas, con componentes cero, como sean necesarias para que el nuevo número de filas sea múltiplo del número de procesos. Luego las filas se distribuyen equitativamente como se aprecia en la Fig. 2.

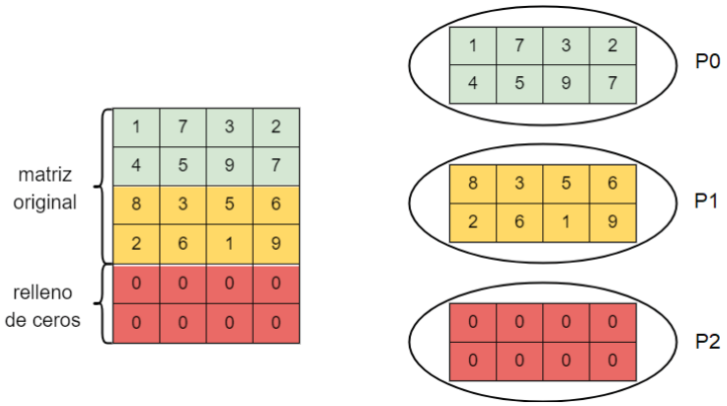


**Fig.2.** Distribución de filas de la matriz A entre p procesos realizada bajo la alternativa 2

La implementación de estas estrategias se basó en una modificación del modelo maestro-esclavo, donde el maestro es el encargado de distribuir los datos que necesita cada esclavo para llevar a cabo el cómputo y posteriormente ejecutar su porción del mismo. La deficiencia del modelo planteado, radica en que las comunicaciones se ejecutan de forma secuencial y previa al cómputo. En este caso, el último nodo esclavo en recibir datos permanece ocioso hasta que el maestro lleve a cabo los envíos

anteriores. Además, cada nodo debe esperar a que se envíen todas las filas de la matriz A para comenzar a recibir la matriz B. Esta misma problemática (comunicaciones secuenciales) se presenta cuando los esclavos envían los resultados obtenidos al maestro.

Tras comparar el rendimiento de ambas alternativas, se concluyó que si bien los resultados son similares, la alternativa 2 presenta un aumento mínimo en el tiempo de ejecución promedio y, en consecuencia, la alternativa 1 fue seleccionada para la comparación con la solución desarrollada en el presente artículo.

## 3 Algoritmo de Multiplicación de Matrices 2.5D

En esta sección se describe el algoritmo de multiplicación de matrices de Cannon [4] en su versión 2,5D presentada en [3].

La versión original del algoritmo de Cannon es considerada un algoritmo 2D ya que distribuye las matrices en una grilla de procesos de tamaño $\sqrt{p} * \sqrt{p}$ (suponiendo que la variable p representa el número total de procesos disponibles para realizar la ejecución). Posteriormente surgieron otras alternativas para calcular la multiplicación de matrices en donde se repartían los datos de entrada en una grilla de $\sqrt[3]{p} * \sqrt[3]{p} * \sqrt[3]{p}$ procesos, por lo que se denominan estrategias 3D.

En esta nueva versión del algoritmo de Cannon, la grilla está conformada por c niveles y cada uno de ellos compuesto por $\sqrt{\frac{p}{c}} * \sqrt{\frac{p}{c}}$ procesos. De acuerdo a la clasificación descripta anteriormente, es posible identificarla como un algoritmo 2,5D ya que se presenta como una solución intermedia entre las estrategias 2D y 3D.

Para llevar a cabo la multiplicación de matrices en 2,5D, inicialmente se distribuyen las matrices de entrada A y B entre los procesos ubicados en el primer nivel de la grilla. De esta manera, el proceso $P_{0,1,0}$ recibe los bloques: $A_{0,1,0}$ y $B_{0,1,0.}$ Cada bloque intercambiado posee un tamaño de $\frac{n}{\sqrt{\frac{p}{c}}} * \frac{n}{\sqrt{\frac{p}{c}}}$. Luego, cada proceso se encarga de enviar los datos asignados a los siguientes niveles de la grilla, obteniéndose así c copias de los datos de entrada.

Posteriormente, se producen uno o múltiples reordenamientos inter-nivel puesto que cada proceso debe poseer los datos adecuados para realizar una o múltiples multiplicaciones de bloques. Por ejemplo, de nada sirve que $P_{0,1,0}$ realice la multiplicación de los bloques $A_{0,1,0}$ y $B_{0,1,0}$ recibidos ya que dichos bloques no deben ser multiplicados bajo el algoritmo de Strassen. Un proceso ejecutará múltiples multiplicaciones de bloques en caso de que $\sqrt{\frac{p}{c^3}}$ sea mayor que 1.

El algoritmo de Cannon está planteado de modo tal que aquellos c procesos que se encuentran en niveles distintos pero en la misma ubicación (x, y) procesen un único bloque de la matriz C resultante. Tal es así que los resultados parciales obtenidos en los c procesos, se suman en el proceso ubicado en el nivel 0.

Finalmente, se obtiene la matriz C completa unificando en una misma variable los resultados de cada proceso del primer nivel.

# 4 Implementación del Algoritmo de Multiplicación de Matrices 2,5D

De forma similar a la implementación 2D-compatible presentada en [5] para el algoritmo SUMMA, para el diseño de la solución 2D-compatible del algoritmo de Cannon se contempló una grilla 2,5D de procesos y una distribución 2D de bloques. La misma es ilustrada en la Fig. 3. para una grilla conformada por ocho procesos ubicados en dos niveles.

Por simplicidad, la implementación generada sólo es aplicable a matrices cuadradas y a una grilla de procesos cuadrada.

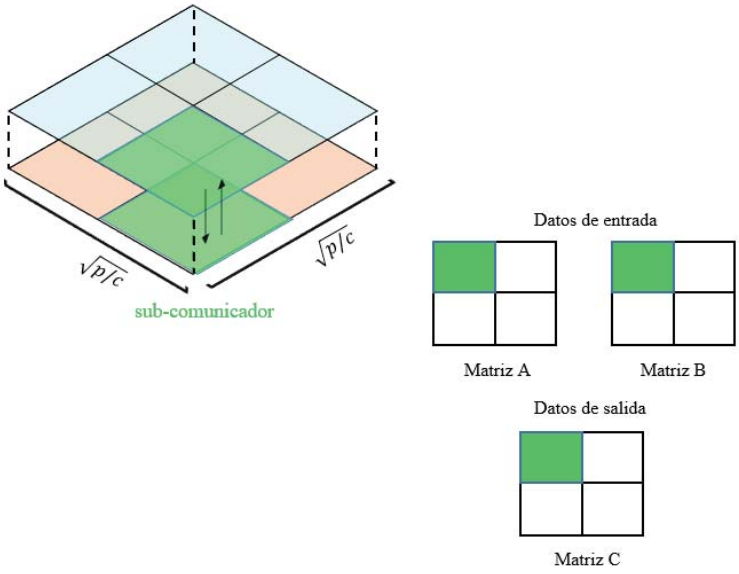


**Fig. 3.** Representación gráfica de la grilla 2,5D de procesos y distribución 2D de bloques de matrices

Inicialmente, se crean tantos procesos como nodos estén disponibles y haciendo uso de la operación MPI_Comm_split se crean sub-comunicadores para que cada proceso del nivel 0 pueda enviar los bloques de datos de entrada masivamente a sus procesos correspondientes en los restantes niveles. Para que cada proceso del primer nivel acceda a los datos de entrada, el proceso con id=0, reordena las matrices de entrada A y B y envía a través de una comunicación no bloqueante (MPI_Isend) los bloques correspondientes a los procesos de su mismo nivel. El reordenamiento de las matrices se vuelve obligatorio en esta implementación 2D-compatible, ya que el envío de datos vía MPI se puede producir únicamente sobre posiciones de memoria contiguas.

Una vez que cada proceso del primer nivel recibe los bloques de matriz, los envía mediante el sub-comunicador creado y haciendo uso de la operación colectiva

MPI_Bcast. Al finalizar estas tareas, se cuenta con 2 copias de las matrices A y B, una por cada nivel, en el caso particular analizado en la Fig. 3.

Posteriormente, se producen múltiples distribuciones de bloques inter-nivel mediante MPI_Isend y MPI_Irecv. Cada vez que un proceso determina que cuenta con los datos suficientes para la multiplicación, ya sea porque los recibió o porque contaba con ellos inicialmente, ejecuta el cómputo correspondiente y queda disponible para un nuevo intercambio de datos, en caso de corresponder. Por cada multiplicación de bloques ejecutada por un proceso, debe acumular los resultados obtenidos.

En este momento, aún resta sumar los resultados parciales de los procesos pertenecientes a un mismo sub-comunicador para obtener los valores definitivos de un bloque de la matriz C. Para esto, los procesos de cada nivel envían los bloques resultantes a los procesos del primer nivel, para que estos últimos ejecuten la suma de bloques y así se obtenga el bloque de C definitivo.

Finalmente cada proceso del nivel 0 debe enviar el bloque de matriz C obtenido al proceso 0, para que el mismo unifique todos los bloques en uno sólo. En lugar de que el proceso maestro reciba todos los bloques y luego reordene la matriz C, como se implementó en [5], se prefirió que cada proceso envíe fila por fila su bloque y cada fila sea ubicada en la matriz C resultante de forma ordenada, logrando así una mejor distribución del trabajo entre procesos a costa de un aumento en la comunicación.

Bajo esta implementación, la cantidad de memoria necesaria por proceso, a excepción del proceso 0 que requiere de más memoria para obtener el resultado final, es aproximadamente $3 * n^2 * \frac{c}{p}$. Si comparamos este valor, con la cantidad aproximada de memoria requerida para la implementación 1D: $2 * \frac{n^2}{p} + n^2$, notamos que se ha obtenido una reducción significativa. Sin embargo, se produce un aumento en la cantidad de comunicaciones realizadas, ya que como inicialmente los procesos no reciben todos los datos necesarios, se deben realizar intercambios (inter-nivel) para obtenerlos. No obstante, ciertas comunicaciones se realizan en paralelo: comunicaciones por sub-comunicador y comunicaciones inter-nivel posteriores a la distribución original. Las consecuencias generadas por estas variaciones es lo que se pretende conocer con las múltiples ejecuciones de los algoritmos en diversos escenarios.

## 5 Diseño Experimental

Para evaluar el rendimiento de las implementaciones generadas, se utilizó un cluster homogéneo compuesto por 8 nodos. Dicho cluster pertenece al modelo de memoria distribuida debido a que los nodos (o computadoras) que lo forman están conectados a través de una red de área local con una velocidad de 100 Mb/seg, y cada uno de ellos cuenta con un espacio de memoria RAM de 8 GB. Además, cada nodo cuenta con un procesador multinúcleo Intel i5 3.2Ghz. Debido a la disponibilidad de computadoras existentes en el gabinete de computación, el máximo de nodos a añadir al cluster es 24. Sin embargo la implementación del algoritmo 2D-compatible requiere de una cantidad de nodos tal que $\sqrt{\frac{p}{c^3}}$ y $\sqrt{\frac{p}{c}}$ sean números enteros. Al seleccionar un valor c=2, el

mínimo valor de p posible es 8, mientras que el siguiente valor posible es 32. Aumentar el valor de c en una unidad implica la necesidad de aumentar el valor de p a 108. Al contar únicamente con 24 nodos, la única configuración posible es c=2 y p=8.

En la capa de software el cluster cuenta con un sistema operativo Ubuntu 14.04 LTS y NFS (Network File System) como sistema de archivos distribuido. En la capa de desarrollo, el lenguaje con el que se cuenta es C sumado a la librería MPICH2, necesaria para trabajar con MPI.

El rendimiento de las implementaciones se analizó para una cantidad fija de nodos (disponibles en el cluster) y con las siguientes variaciones en el tamaño de la matriz: 5000, 6000, 7000, 8000, 9000,10000 filas (y columnas).

La medición del tiempo de ejecución de cada escenario se inició una vez finalizada la inicialización aleatoria de las matrices A y B, es decir previo a la distribución inicial de datos, y se finalizó una vez obtenida la matriz C por el proceso 0. Además de medir el tiempo total de ejecución, se analizó su descomposición en tiempo asociado a la comunicación y tiempo asociado al cómputo. Las mediciones fueron llevadas a cabo, en todos los casos, por el proceso 0.

Para contemplar las variaciones de rendimiento a lo largo de las múltiples ejecuciones, se midió el tiempo de ejecución de cada escenario cinco veces. El valor considerado como resultado en la siguiente sección se corresponde con el promedio de los cinco tiempos obtenidos.

## 6 Resultados

En esta sección se presentan los resultados obtenidos tras la ejecución de ambas implementaciones en el cluster para los seis escenarios planteados.

En la Tabla 1, se muestra el tiempo de ejecución promedio en segundos obtenido para cada escenario.

**Tabla 1.**: Comparación de tiempos de ejecución promedio en múltiples escenarios

| | **5000** | **6000** | **7000** | **8000** | **9000** | **10000** |
|---|---|---|---|---|---|---|
| **1D** | 62,667 | 102,333 | 153,000 | 237,000 | 307,000 | 405,333 |
| **2,5D** | 57,33 | 83,67 | 136,00 | 213,67 | 286,67 | 379,67 |

En la Fig. 4, se grafica la evolución de la reducción porcentual del tiempo de ejecución que otorga la nueva implementación para cada escenario evaluado. Se puede apreciar que en todos los casos analizados la reducción supera el 6%, alcanzando su máximo valor para un tamaño de matriz de 6000 filas y columnas. Es posible notar que mientras aumenta el tamaño de la matriz, la reducción disminuye. Para poder analizar este último fenómeno en detalle, resultó de interés conocer la descomposición de la reducción en: reducción asociada a la comunicación y reducción asociada al cómputo.

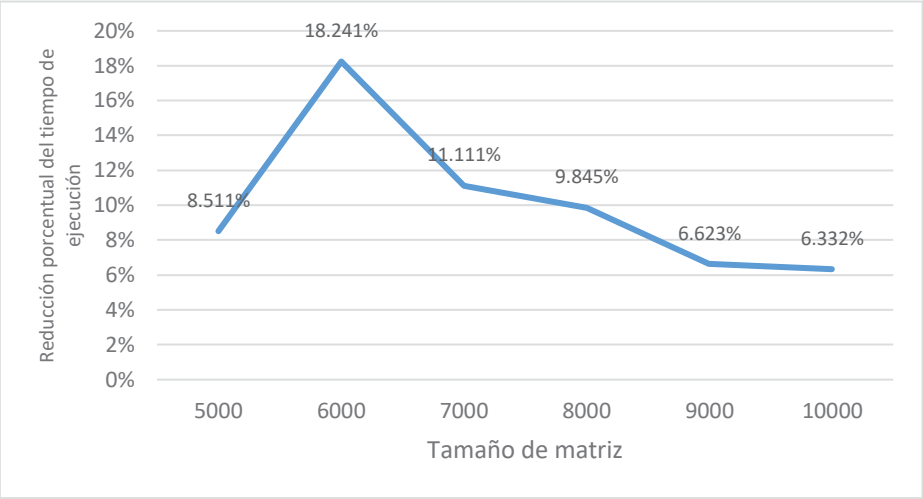


**Fig. 4.** Evolución de la reducción porcentual del tiempo de ejecución

La Fig. 5 permite afirmar que la reducción total conseguida es en gran parte consecuencia de la reducción asociada al cómputo.

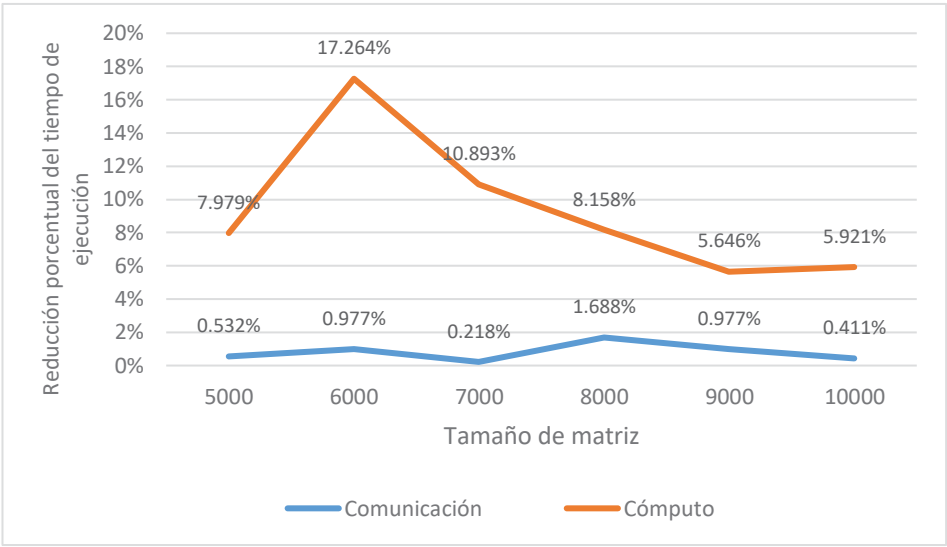


**Fig. 5.** Comparación de la reducción porcentual asociada a la comunicación y reducción porcentual asociada al cómputo

## 7 Conclusiones y Trabajos Futuros

Tras analizar la evolución y composición de la reducción total obtenida con la nueva implementación para los diversos escenarios, es posible obtener las siguientes conclusiones.

En primer lugar, la escasa reducción asociada a la comunicación permite determinar que la paralelización en las comunicaciones entre niveles y la reducción de la cantidad de datos intercambiados por cada comunicación se compensa con el aumento en la cantidad de comunicaciones, resultando así una reducción mínima (o incluso un aumento) de comunicación.

En segundo lugar, la reducción asociada al cómputo obtenida en cada caso, no se relaciona con la forma de realizar la multiplicación de bloques, que es idéntica en ambos casos, sino a la rapidez en el acceso a los datos al haberse conseguido una reducción en la cantidad de datos almacenados por proceso. Como a medida que aumenta el tamaño del problema se supera la capacidad de memoria caché de cada nodo, el rendimiento del algoritmo disminuye y se asemeja cada vez más al algoritmo 1D.

En tercer lugar, podemos atribuirle a la distribución 2D de bloques la capacidad de reducir la cantidad de datos almacenados y, en consecuencia, disminuir el tiempo asociado al cómputo.

Estudios futuros se van a enfocar en realizar pruebas en arquitecturas distribuidas que posean una mayor cantidad de nodos disponibles para la ejecución, permitiendo así probar el rendimiento del nuevo algoritmo para una grilla configurada con cuatro niveles. Bajo esta configuración, se evaluará el impacto de reducir la cantidad de datos almacenados por nodo en el rendimiento. Además, para maximizar el rendimiento mediante un enfoque híbrido, se implementará la multiplicación de bloques de matrices a través de OpenMP, permitiendo así una mejor utilización de los recursos de cada nodo.

## Referencias

1. Strassen, V.: Gaussian elimination is not optimal. Numer. Math. 13, 354–356 (1969)
2. Sanchez, F.: Paralelismo Híbrido para resolver problemas de cómputo intensivo, (2018)
3. Solomonik, E., Demmel, J.: Communication-optimal parallel 2.5 D matrix multiplication and LU factorization algorithms. In: European Conference on Parallel Processing. pp. 90–109 (2011)
4. Cannon, L.E.: A cellular computer to implement the Kalman filter algorithm, (1969)
5. Mukunoki, D., Imamura, T.: Implementation and Performance Analysis of 2.5 D-PDGEMM on the K Computer. In: International Conference on Parallel Processing and Applied Mathematics. pp. 348–358 (2017)