UNIVERSIDAD NACIONAL DE COLOMBIA
SEDE MEDELLÍN
FACULTAD DE MINAS

DYNA
http://dyna.medellin.unal.edu.co/

# Matrix multiplication with a hypercube algorithm on multi-core processor cluster

José Crispín Zavala-Díaz [a], Joaquín Pérez-Ortega [b], Efraín Salazar-Reséndiz [b] & Luis César Guadarrama-Rogel [b]

[a] *Facultad de Contaduría, Administración e Informática, Universidad Autónoma del Estado de Morelos, Cuernavaca, México. crispin_zavala@uaem.mx*
[b] *Departamento de Ciencias Computacionales, Centro Nacional de Investigación y Desarrollo Tecnológico, Cuernavaca, México. jpo_cenidet@yahoo.com.mx, efras.salazar@cenidet.edu.mx, cesarguadarrama@cenidet.edu.mx*

Received: September 10th, 2014. Received in revised form: March 9th, 2015. Accepted: March 17th, 2015

**Abstract**
The algorithm of multiplication of matrices of Dekel, Nassimi and Sahani or Hypercube is analysed, modified and implemented on multi-core processor cluster, where the number of processors used is less than that required by the algorithm $n^3$. $2^3$, $4^3$ and $8^3$ processing units are used to multiply matrices of the order of 10x10, $10^2$x$10^2$ and $10^3$X$10^3$. The results of the mathematical model of the modified algorithm and those obtained from the computational experiments show that it is possible to reach acceptable speedup and parallel efficiencies, based on the number of used processor units. It also shows that the influence of the external communication link among the nodes is reduced if a combination of the available communication channels among the cores in a multi-core cluster is used.

*Keywords: Hypercube algorithm; multi-core processor cluster; Matrix multiplication*

# Multiplicación de matrices con un algoritmo hipercubo en un cluster con procesadores multi-core

**Resumen**
Se analiza, modifica e implementa el algoritmo de multiplicación de matrices de *Dekel, Nassimi y Sahani* o hipercubo en un cluster de procesadores multi-core, donde el número de procesadores utilizado es menor al requerido por el algoritmo de $n^3$. Se utilizan $2^3$, $4^3$ y $8^3$ unidades procesadoras para multiplicar matrices de orden de magnitud de 10X10, $10^2$X$10^2$ y $10^3$X$10^3$. Los resultados del modelo matemático del algoritmo modificado y los obtenidos de la experimentación computacional muestran que es posible alcanzar rapidez y eficiencias paralelas aceptables, en función del número de unidades procesadoras utilizadas. También se muestra que la influencia del enlace externo de comunicación entre los nodos disminuye si se utiliza una combinación de los canales de comunicación disponibles entre los núcleos en un cluster multi-core.

*Palabras Clave: Algoritmo Hipercubo, cluster de procesadores multi-core, Multiplicación de matrices*

## 1. Introduction

The multicore clusters are formed by either nodes or processors, often heterogeneous, connected with architecture of dynamic configuration, high speed bus; it allows point to point connections among each processing unit. The nodes are composed of processors and the processors in turn are composed of multiple cores, where each of them has the ability to run more than one process at a time [1]. These features allow us to propose solutions to various problems using computing parallel and concurrent [2,3], such as the one presented in this work: The matrix multiplication with the DNS algorithm (Dekel, Nassimi, Sahani or hypercube) on a multicore cluster.

The execution time of the DNS or Hypercube algorithm for multiplying *nxn* matrices is polylogarithmic $T\infty(\log n)$ and requires a polynomial number of processors $Hw(n^3)$, these two parameters classify it as an efficiently parallelizable algorithm [4]. But, in its implementation in a multicore cluster the following limitations are presented: First, the number of processing units available is finite and is lesser than those required by the hypercube algorithm for matrice sizes given $Hw(n^3)$; Second, the processing units are spread over various nodes and processors.

Where the speed of the communication links among the cores is different, they can be on the same processor, in the same node, either in different nodes or processors.

Similar modifications [5] are proposed to implement the hypercube algorithm on multicore clusters. In the modification proposed in this work the number of processing units remains constant, leaving as a variable the grain size or the number of the subarrays that divides the *nxn* matrices. Contrary to what is proposed by [5], where the number of processing units is determined by the size of the input.

The implementation of the algorithm is made by the library Message Passing Interface MPI, where MPI processes are assigned to the cores. If an MPI process is assigned to a core, then it will be parallel computation; but if more than one MPI process is assigned to the same core, then it will be concurrent computation. Computational experiments took place in the *ioevolution* multicore cluster. Where *nxn* matrices are multiplied by 8, 64 and 512 processing units, the matrix's size is a multiple of 72, from 72 x 72 up to 2304 x 2304 items.

Theoretical and experimental results show that the solution proposed in this work allows to multiply matrices of size multiples 1,000X1,000 in multi-core cluster, using a smaller number of processors than required processors by original algorithm; with modifications proposed, it is possible to achieve acceptable speedup and good parallel efficiencies, depending on the number of processing units used; the developed mathematical model of the modified DNS algorithm predicts the behavior of the parallel implementation on a multi-core cluster.

The article consists of the following parts: in the second the methodology and foundations of the Hypercube algorithm are presented; in the same section the analysis of its running time on a MIMD computer, of memory and distributed computation, is presented. Based on this analysis, the modifications to multiply matrices on a multi-core cluster are presented. The third part contains the computational experiments and analysis of results. Finally, in the fourth section the conclusions that come out of this work are presented.

## 2. Methodology

Algorithms for matrix multiplication based on the model *Simple Instruction Multiple Data* (SIMD) has been developed since the beginning of the parallel computation, such as: Systolic Array or Mesh of Processors, Cannon, Fox, DNS method (Dekel, Nassimi, Sahani) or hypercube and meshes of trees [4-8]. In all of them, except for DNS, the parallel process of solution consists of sequential steps of data distribution and computing. In general, in the first stage the data between processor units are distributed. In the second, each processor performs the multiplication of elements allocated. In the third, the first and second steps are repeated until all columns and rows of the matrices are rotated and multiplied. In the fourth stage the resulting elements are added. In contrast, in either the DNS or Hypercube algorithm, the distribution and multiplying of elements run only once, the first and second stages, and similarly to the other algorithms the fourth stage is executed once.

Then the DNS or Hypercube algorithm is described.

```
Parameter q { matrix size is 2^q X 2^q}
Global l
Local a, b, c, s, t
Begin
{Phase 1: Broadcast matrices A and B}
for l ←3q-1 down to 2q do
  for all P_m where BIT(m, l) = 1 do
     t ← BIT.COMPLEMENT(m, l)
     a ⇐ [t]a
     b ⇐ [t]b
  end for
end for
for l ← q-1 down to 0 do
 for all P_m, where BIT(m,l)≠BIT(m, 2q+l) do
    t ← BIT.COMPLEMENT(m, l)
    a ⇐ [t]a
 end for
end for
for l ←2q-1 down to q do
  for all P_m do
    t ← BIT.COMPLEMENT(m, l)
    b ⇐ [t]b
  end for
end for
{Phase 2: Do the multiplications in parallel}
For all P_m do
  c ← a x b
end for
{Phase 3: Sum the products}
for l ←2q to 3q-1 do
  for all P_m do
     t ←BIT.COMPLEMENT(m, l)
     s ⇐ [t]c
     c ← c + s
  end for
end for
End
```

Figure 1. Pseducodigo Hypercube Algorithm for matrix multiplication (Hypercube SIMD).
Source Quinn, Michael J., [4]

### 2.1. Hypercube algorithm

The algorithm is based on multiplying two *nxn* matrices using the model *Simple Instruction Multiple Data* (SIMD), the execution time is $\theta(\log n)$ and requires $n^3$ processors. This consists of $q$ steps to distribute and rotate the data among the processor units, the number of stages is given by $5q = 5 \log n$ and the matrix size by $n^3 = 2^{3q}$ [4].

The solution strategy is: distribute and rotate the elements of the matrices ($a_{ik}$, $b_{ik}$) in the computer nodes, connected by hypercube architecture. Once the rotation and distribution of elements is performed, each processing unit has the pair ($a_{ik}$, $b_{ki}$), then they are multiplied. After the multiplication, the products are sent to be added and get the $c_{ij}$ elements.

The matrix multiplication algorithm SIMD Hypercube comprises three phases: In the first, the elements are distributed and rotated in the processing units; in the second phase the elements are multiplied; and in the third, the products are summed to obtain the solution matrix.

### 2.2. Runtime hypercube algorithm in a multi-core cluster

For the execution time on a multi-core cluster it is

Table 1.
Execution time of hypercube algorithm in a computer of distributed memory and communication cost.

| Phase | Loop | time (*processor operations*) |
|---|---|---|
| First | First Initial distribution of ($a_{ij}$, $b_{ij}$) | $T_{comm1}$=g(2C(3q-1– 2q)) = g2C(q-1) |
| | Second rotation ($a_{ij}$) | $T_{comm2}$ = g(1C(q-1 - 0))= gC(q-1) |
| | Third rotation ($b_{ij}$) | $T_{comm3}$ = g(1C(2q – 1 - q) = gC(q-1) |
| Second | Unique Product ($a_{ij}$x$b_{ji}$) | $T_{cal}$ = 1 |
| Third | Unique Add the ($c_{ij}$) | $T_{comm4}$ = g(1C(3q-1 -2q)) = gC(q-1) |
| Total $T_p$ | | $T_{cal}$ + $T_{comm1}$ + $T_{comm2}$ + $T_{comm3}$ + $T_{comm4}$ = 1+ 5gC(q-1) |

Source: The authors

Table 2.
Time for calculations and communication of modified hypercube algorithm on a multi-core computer.

| Phase | Loop | time (*processor operations*) |
|---|---|---|
| First | First | $g\left(2C\left(\frac{n}{k}\right)^2(q-1)\right)$ |
| | Second | $g\left(C\left(\frac{n}{k}\right)^2(q-1)\right)$ |
| | Third | $g\left(C\left(\frac{n}{k}\right)^2(q-1)\right)$ |
| Second | Unique | $\left(\frac{n}{k}\right)^3$ |
| Third | Unique | $g\left(C\left(\frac{n}{k}\right)^2(q-1)\right)$ |
| Total | | $\left(\frac{n}{k}\right)^3 + g\left(5C\left(\frac{n}{k}\right)^2(q-1)\right)$ |

Source: The authors

necessary to introduce the factors that influence its implementation, such as: time used: calculation $T_{cal}$, communications $T_{comm}$, wait $T_{wait}$ and local operations $T_{local}$ [9]. The $T_{wait}$ and $T_{local}$ times are assumed to be zero for the following reasons: First, each processing unit receives the data and immediately executes the next operation ($T_{wait}$ = 0); Second, the processor units do not perform local operations for the distribution of the data ($T_{local}$ = 0). Since the beginning of the process, the units neighbouring for sending are defined. Therefore, parallel execution time is determined by:

$$T_p = T_{cal} + T_{comm} \quad (1)$$

To be able to add the $T_{cal}$ and $T_{comm}$ times, they must be expressed in the same units: runtime or in the number of operations of the processing unit, so it is necessary to standardize the communication time. To this end the constant $g$, eq. (2) [10], is introduced.

$$g = \frac{processor\ speed}{link\ speed} = \frac{floating\ point\ operations\ per\ second}{bits\ per\ second} \quad (2)$$

The constant calculates the communication time depending on processor operations [10]. Therefore, the communication time is determined by the expression:

$$T_{comm} = gC \quad (3)$$

To calculate the execution time in a parallel computer of distributed memory and distributed process the $T_{cal}$ and $T_{comm}$ times are entered in the algorithm of Fig. 1. The execution time of the phases of Hypercube algorithm is shown in Table 1.

Whereas $q$ = log$n$ time parallel execution is:

$$T_p(n) = T_{cal}+T_{comm} = 1+5gC(q-1) < 1+5gCq = 1+5gC\log n \quad (4)$$

If eq. (4) is compared with equation unmodified $T_p(n)$ = 1 + 5log$n$ [4], it shows that the time of the parallel execution, eq. (4), grows approximately $gC$ multiple, because normally g > 1 and $C$ is always greater than 1. This indicates that the parallel algorithm is faster than the sequential for the matrix's certain size. For example, if g = 2 and $C$ is equal to 64 bits, the matrix of size 14x14 is needed for which the modified hypercube parallel algorithm is going to be faster than the sequential. Additionally, 2,744 processing units are required to implement it.

Multiplying matrices of dimensions 1,000X1,000 in a multi-core cluster with this algorithm has the following drawback: the number of processing units is not available. Consequently, it needs to be modified in order to multiply matrices of this size, because the number of available processing units is less than $n^3$.

### *2.3. Amendments to the DNS algorithm to it runs on a multi-core computer*

The modification is to increase the grain size, as follows: to replace the scalar by sub matrices and that they follow the same procedure for the distribution and rotation of elements $a_{ij}$ and $b_{ij}$. Consequently, the distribution and rotation of sub matrices will be a function of the $q$ variable equal to log$n$. The value of the variable is determined by the number that divides the matrices to multiply. If the $n$ dimension of the matrix is divided into $k$ parts to generate submatrices $\frac{n}{k}X\frac{n}{k}$, then $q$ is equal to log$k$. The number of submatrices generated is $2^{3q} = k^3$ and the number of processors is given by $p = k^3$. The k variable can define the size of the submatrices and the number of processing units that are required.

The necessary modifications to the algorithm in Fig. 1 below:

- Replace to $a_{ij}$ by $\boldsymbol{A}_{ij}$
- Replace to $b_{ij}$ by $\boldsymbol{B}_{ij}$
- Replace to $c_{ij}$ by $\boldsymbol{C}_{ij}$
- Replace to $a_{ik}b_{kj}$ by $\sum_{i=1}^{n/k}\sum_{j=1}^{n/k}\sum_{l=1}^{n/k} a_{il}b_{lj}$

The computation and communication times of the modified algorithm are shown in Table 2.

The execution time of the parallel algorithm is determined by:

$$T_p(n) = T_{cal} + T_{comm} = \left(\frac{n}{k}\right)^3 + g\left(5C\left(\frac{n}{k}\right)^2(q-1)\right) \leq \left(\frac{n}{k}\right)^3 + 5gCq\left(\frac{n}{k}\right)^2 \quad (5)$$

Whereas $p = k^3$ and $q = \log k$ the expression (5) is reduced to:

$$T_p(n) = \frac{n^3}{k^3} + 5gCq\frac{n^2}{k^2} = \frac{n^3}{p} + \frac{5}{3}gC\frac{n^2}{p^{2/3}}logp \quad (6)$$

The $C$ variable, in the expression (6), is the type of data that is sent and $g$ is given by eq. (2). Table 3 shows the results of eq. (6) for a value of $g = 1$ and 2, $C = 32$ and a number of processing units 8, 64 and 512 ($k = 2$, 4 and 8). The results are presented in Table 3 based on the parallel speedup $Sp = (T_1/T_p)$.

The results of the eq. (6) shows that when $g = 1$ runtimes are smallest. This indicates that the algorithm is sensitive to the characteristics of the communication link. Regarding the parallel speedup, it is best using eight processing units for multiplying matrices smaller than 1,000X1,000, from 64 to matrices of smaller sizes than 10,000X10,000 and 512 for multiplying matrices over 10,000X10,000. Fig. 2 plots the data of Table 3.

As seen in Fig. 2, the largest differences occur in large arrays, the multiplication matrix is faster when more processing units are used. When using as a reference the parallel efficiency Ep = (Sp/p)X100, the better efficiencies are obtained when less processing units are used, as shown in Fig. 3.

As is shown in Fig. 3, the best efficiency is attained when the minor number of parallel processing units are used, about 100% for the larger matrices. However, the parallel efficiency decreases when more processing units are used, at best by 20%. Although the trend is that the efficiency will increase as the size of the matrix increases. This indicates that for 512 parallel processing units will reach its asymptotic value for a given size of matrices.

Table 3. Results of the eq. (6) to C = 32 (elaborated by us)

| Matrix size | *Sp* $g = 1$ | | |
|---|---|---|---|
| | 8 | 64 | 512 |
| 72X72 | 0.685714 | 0.372816 | 0.249878 |
| 144X144 | 1.263158 | 0.741313 | 0.499512 |
| 288X288 | 2.181818 | 1.465649 | 0.998051 |
| 576X576 | 3.428571 | 2.865672 | 1.992218 |
| 1152X1152 | 4.8 | 5.485714 | 3.968992 |
| 2304X2304 | 6 | 10.10526 | 7.876923 |
| 4608X4608 | 6.857143 | 17.45455 | 15.51515 |
| 9216X9216 | 7.384615 | 27.42857 | 30.11765 |
| 18432X18432 | 7.68 | 38.4 | 56.88889 |
| 36864X36864 | 7.836735 | 48 | 102.4 |
| Matrix size | *Sp* $g = 2$ | | |
| | 8 | 64 | 512 |
| 72X72 | 0.358209 | 0.186952 | 0.124969 |
| 144X144 | 0.685714 | 0.372816 | 0.249878 |
| 288X288 | 1.263158 | 0.741313 | 0.499512 |
| 576X576 | 2.181818 | 1.465649 | 0.998051 |
| 1152X1152 | 3.428571 | 2.865672 | 1.992218 |
| 2304X2304 | 4.8 | 5.485714 | 3.968992 |
| 4608X4608 | 6 | 10.10526 | 7.876923 |
| 9216X9216 | 6.857143 | 17.45455 | 15.51515 |
| 18432X18432 | 7.384615 | 27.42857 | 30.11765 |
| 36864X36864 | 7.68 | 38.4 | 56.88889 |

Source: The authors

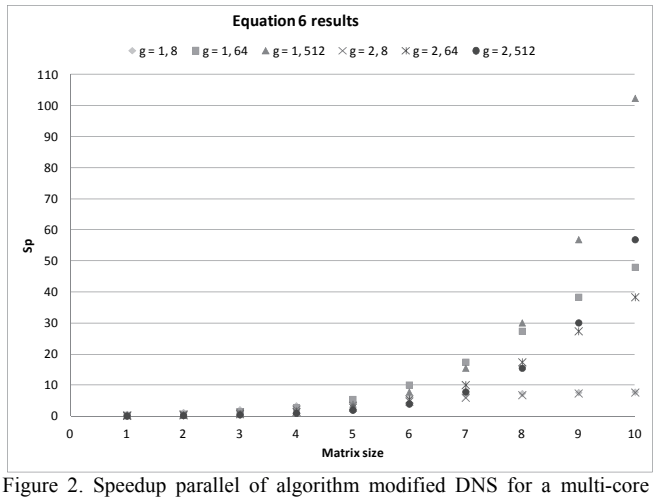


Figure 2. Speedup parallel of algorithm modified DNS for a multi-core cluster.
Source: The authors

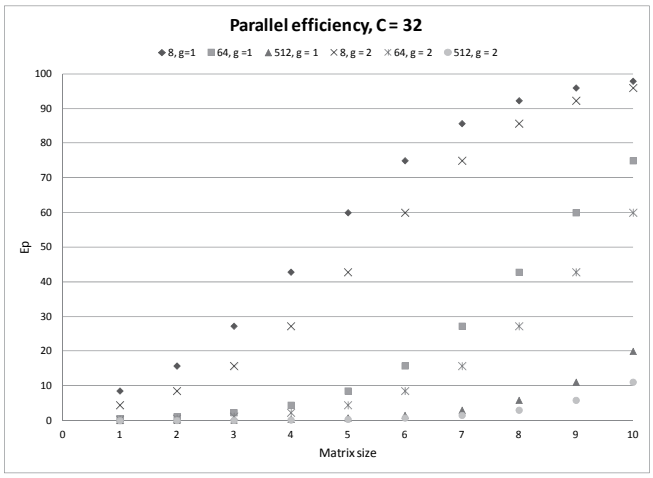


Figure 3. Parallel efficiency of DNS algorithm modified for multi-core cluster.
Source: The authors

The parallel efficiency low is related to the time spent on communications, it grows exponentially when the number of processing units is higher and, consequently, the data sent concurrently among processing units.

## 3. Computational experiments in the multi-core cluster

The computational experiment was conducted on a multi-core cluster with a finite number of processors and memory units. In consequence the size of the matrices of the computational experiment was limited.

In view of the fact that the cores can be programmed as independent units [1]. The algorithm was encoded in ANSI C and for sending data the MPI library was used. The MPI-processes are assigned to the cores. If only one of them is assigned to a core, then it will work with parallel computing. But, if more than one is assigned, then it will work with parallel and concurrent computing. All nodes in the multi-

Table 4.
Configuring multi-core cluster *ioevolution.*

| Node | Model | Processors per node | Cores per Processor | Total Cores |
|---|---|---|---|---|
| 0 | Intel Xeon CPU X3430 a 2394 MHz | 4 | 4 | 16 |
| 1 | Intel Xeon CPU X3430 a 2394 MHz | 4 | 4 | 16 |
| 2 | Intel Xeon CPU X3430 a 2394 MHz | 4 | 4 | 16 |
| 3 | Intel Xeon CPU E5645 a 2400.067 MHz | 12 | 6 | 72 |
| Total | | | | 120 |

Source: The authors

core cluster are used in the experimentation, each node has the same load in all tests, for which the same number of MPI-processes to each processing unit is assigned.

### *3.1. Description of multi-core cluster ioevolution*

The multi-core cluster used has the following characteristics:

Features cluster shows that there are different *g* constants. The first is when two processes are assigned to the same core; the second is when the cores are in the same processor; the third corresponds to the communication between processors in the same node; and the fourth is for communication between nodes in the cluster. Of these, it is considered the external link between nodes, it is by means of fibre optics with a bandwidth of 1 Gbit per second, and with the speed of nuclei the $g \approx 2.4$ is obtained.

### *3.2. Computational test on cluster ioevolution*

The computational tests consist of multiplying matrices without gaps and in different sizes. Its elements are real, 32-bit floating number, and are generated randomly. During each execution of the program different matrices are used.

The sizes of the matrices are multiples of the number 72, because this is a number divisible by *k* values (2, 4 and 8). Table 5 shows the average execution time obtained in 30 runs and the speedup of the parallel implementation is presented.

If the theoretical results of Table 3 and experimental results of Table 5 are compared, it is noted that the experimental values of speedup are closer to those calculated when $g = 1$. In order to determine the value of the *g* constant that most approximates the theoretical results to results experimentally, tests were performed with different values of *g*, of them the closest to the experimental is when $g = 1.3$. This is shown in Figs. 4, 5 and 6.

Table 5. Timing and speedup of the parallel implementation of the modified DNS algorithm in multi-core cluster *ioevolution*.

| **Matrix size** | **Time in seconds** | | | |
|---|---|---|---|---|
| | **1** | **8** | **64** | **512** |
| 72x72 | 0 | 0.008281 | 0.039979 | 1.488810 |
| 144x144 | 0.03 | 0.024755 | 0.069789 | 1.645584 |
| 288x288 | 0.234 | 0.105216 | 0.198098 | 1.971476 |
| 576x576 | 1.556667 | 0.484652 | 0.910284 | 3.854292 |
| 1152x1152 | 18.405667 | 3.279813 | 3.940699 | 7.879904 |
| 2304x2304 | 161.591667 | 31.395923 | 19.687011 | 30.453499 |
| **Matrix size** | ***Sp*** | | | |
| | **1** | **8** | **64** | **512** |
| 72x72 | -- | 0 | 0 | 0 |
| 144x144 | 1 | 1.21190087 | 0.42986594 | 0.01823061 |
| 288x288 | 1 | 2.22399846 | 1.18123433 | 0.11869277 |
| 576x576 | 1 | 3.21193063 | 1.71008850 | 0.40387880 |
| 1152x1152 | 1 | 5.61180378 | 4.67066058 | 2.33577306 |
| 2304x2304 | 1 | 5.14689969 | 8.20803438 | 5.30617719 |

Source: The authors

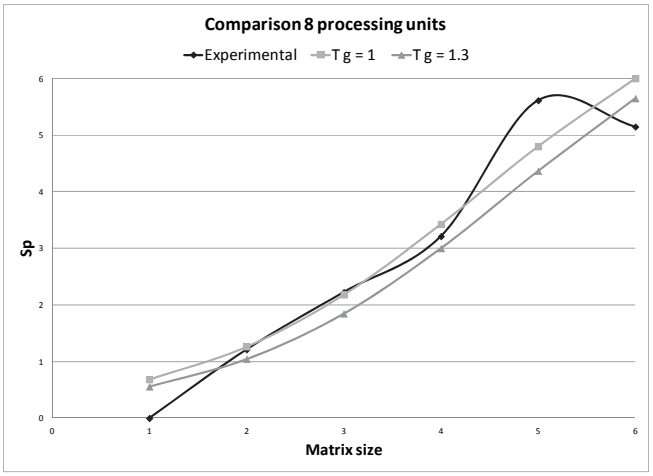


Figure 4. Comparison of parallel speedup, experimental vs. theoretical, of eight processing units.
Source: The authors

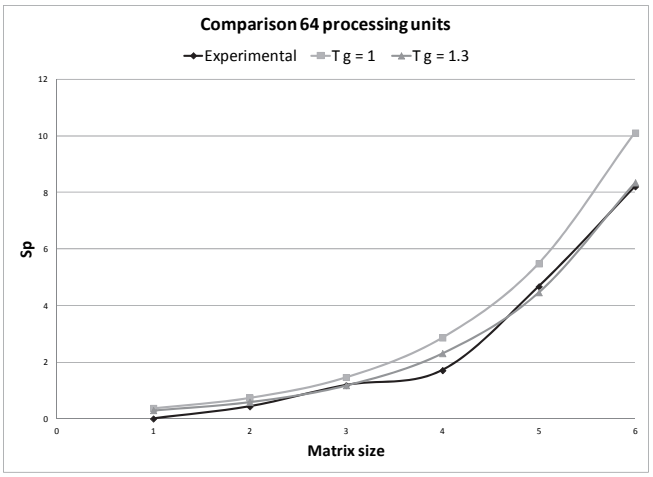


Figure 5. Comparison of parallel speedup, experimental vs. theoretical, 64 processing units.
Source: The authors

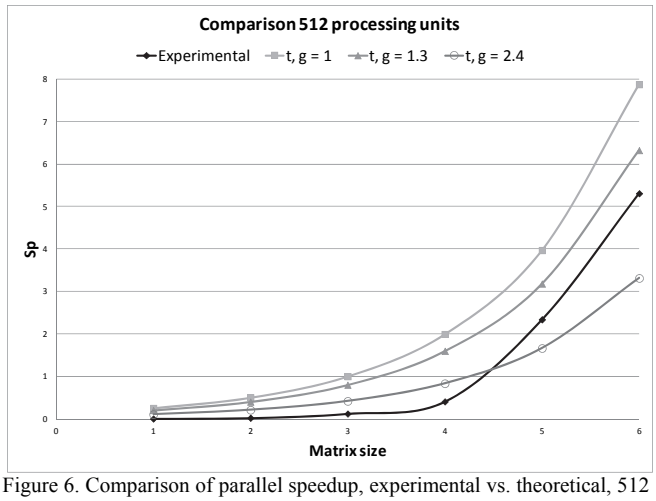


Figure 6. Comparison of parallel speedup, experimental vs. theoretical, 512 processing units.
Source: The authors

The three Figs. 4, 5 and 6 show experimental results, they are closer to the theoretical when $g = 1.3$, this value is lower than the $g$ calculated by the external link ($g = 2.4$). The obtained value of $g = 1.3$ indicates that the use of the combination of the different communication channels of multi-core cluster improves the parallel implementation.

The most approximate experimental results to the theoretical are when 8 and 64 processing units are used. However, when 512 units of processing are used, the more approximate experimental results to the theoretical is when the matrix's size is higher than 1,000X1,000.

In Fig. 6 the values obtained with g = 2.4 are also plotted. It is observed that for matrices with a size smaller than 1,000X1,000 the experimental speedup is lower than those calculated using the external link. In contrast to the matrix size bigger than 1,000X1,000, the experimental speedup is better than the theoretical considering that same link. At this size the experimental results follow the trend of theoretical with a g = 1.3. This shows that there are factors that influence in the reduction of parallel speedup.

Establishing communication among the processing units is one of the factors influencing the parallel speedup. It follows the above with the following, for the same size of matrices and different number of processor units the changes made are: the size of the sub matrices and the number of simultaneous communications. The subarrays are bigger when eight processing units are used and a lesser number of concurrent communications are performed. In contrast, the subarrays are the smallest with 512 processing units and the largest number of simultaneous communications of the three tested cases are performed. The time difference, between the theoretical and the experimental, increases as the number of simultaneous communications increases, although smaller matrices are sent, Figs. 4 and 6. This reveals that in sending data from one processing unit to another there is a mechanism for communication, which requires processor runtime.

Another factor that influences the running time is the number of processes that are calculated in each core. When 8 and 64 processing units are used, a process is assigned to each core and all processes are communicated simultaneously. The communication channel will be a function of where the process will be allocated on either the same processor or node.

On the other hand, when 512 processes are assigned to the cores, each one of them runs more of a process. In this case the transmission of data among processes may be sequential. Sequentially because there is a means of communication among the cores, and it is used for various processes. Therefore, when a process sends its data using the communication channel, other processes assigned to the same core have to wait their turn. This sequentiality is reflected in the experimental results, they follow the trend given by $g = 1.3$ and not given by $g = 2.4$.

It shows that from size matrices 1,152X1,152 the speedup maximum is obtained when eight processor units are used, for bigger matrices the speedup parallel rapidly diminishes. With this value a parallel efficiency of 70.15% is obtained. For the other tested cases, 64 and 512, the execution time of parallel implementation continued to improve. This coincides with the theoretical results, the best parallel efficiencies are obtained with 8 processors and matrices smaller than 1,000X1,000

## 4. Conclusions

It is concluded that:

- With the proposed changes to DNS or hypercube algorithm can multiply matrices of different orders of magnitude in a multi-core cluster, using a number of processors lesser than $n^3$.
- The influence of the external communication link between nodes in the cluster decreases, if a combination of communication channels available among the cores of a multi-core cluster is used.
- The amendment proposed in this paper, the number of processing units, is a function of the number of submatrices in that the matrix is divided.
- For larger problems it was shown that the influence of data access between processors affects parallel efficiency, rather than smaller problems. It is expected that new designs of processors [11, 12] will optimize access to data and consequently the best parallel efficiencies will be obtained.

## References

[1] Rauber, T. and Rünger, G., Parallel programming for multicore and cluster systems. Springer Heidelberg Dordrecht London New York 2010. DOI 10.1007/978-3-642-04818-0.

[2] Muhammad, A. I., Talat, A. and Mirza, S.H., Parallel matrix multiplication on multi-core processors using SPC$^3$ PM, Proceedings of International Conference on Latest Computational Technologies (ICLCT'2012), Bangkok, March 17-18, 2012

[3] L'Excellent, J.-Y. and Sid-Lakhdar, W.M., A study of shared-memory parallelism in a multifrontal solver, Parallel Computing 40 (3-4), pp 34-46, 2014. DOI 10.1016/j.parco.2014.02.003

[4] Quinn, M.J., Parallel computing (2nd ed.): Theory and practice. McGraw-Hill, Inc. New York, NY, USA 1994.

[5] Gupta, A. and Kumar, V., Scalability of parallel algorithms for matrix multiplication, Proceedings of International Conference on Parallel Processing, 3, pp. 115-123, 1991. DOI 10.1109/ICPP.1993.160.
[6] Alqadi, Z.A.A., Aqel, M. and El Emary, I.M.M., Performance analysis and evaluation of parallel matrix multiplication algorithms, World Applied Sciences Journal 5 (2), pp. 211-214, 2008.
[7] J. Choi, A new parallel matrix multiplication algorithm on distributed-memory concurrent computers, Technical Report CRPC-TR97758, Center for Research on Parallel Computation Rice University, [On line] 1997. Aviable at http://citeseerx.ist.psu.edu/viewdoc/download?. DOI=10.1.1.15.4213&rep=rep1&type=pdf.
[8] Solomonik, E. and Demmel, J., Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms, Proceeding Euro-Par'11 Proceedings of the 17th international conference on Parallel processing, Heidelberg, Volume Part II, Springer-Verlag Berlin, Heidelberg, 2011.
[9] Zavala-Díaz, J.C., Optimización con cómputo paralelo, teoría y aplicaciones, México, Ed. AmEditores, 2013.
[10] Sánchez, J. and Barral, H., Multiprocessor implementation models for adaptative algorithms, Journal IEEE Transactions on Signal Processing, USA, 44 (9), pp. 2319-2331, 1996.
[11] Un Nuevo Proyecto Financiado con Fondos Europeos trata de Lograr los Primeros Chips de Silicio para RAM Ã³ptica de 100 Gbps.Dyna 87 (1), pp. 24, 2012.
[12] La Politécnica de Valencia Patenta Mejoras en Procesadores. Dyna, 83 (8), pp. 155 2008.

**J.C. Zavala-Díaz**, received the PhD. degree in Computational Science at Monterrey Institute of Technology and Higher Education (ITESM), Mexico, in 1999, since 1999 he is a research professor at the Autonomous University of the State of Morelos, Mexico. He worked in Research Electrical Institute (IIE), México 1986-1994. His research interests include: parallel computing; modeling and problem solving of discrete and linear optimization; and optimization using metaheuristics.

**J. Pérez-Ortega,** received the PhD. degree in Computational Science at Monterrey Institute of Technology and Higher Education (ITESM), Mexico in 1999, since 2001 he is a research of National Research and Technological Development (CENIDET). He worked in Research Electrical Institute (IIE), México 1985-2001. His research interests include: optimization using metaheuristics; NP-problems; combinatorial optimization; distributed systems; and software engineering.

**E. Salazar-Reséndiz,** received the MSc. degree in Computational Science at National Research and Technological Development (CENIDET) in 2014. he works in Research Electrical Institute (IIE).

**C. Guadarrama-Rogel**, received the MSc. degree in Computational Science at National Research and Technological Development (CENIDET) in 2014. he works in Research Electrical Institute (IIE).