Optimización de *MCTDH Heidelberg package* para simulación de electrones en un doble punto cuántico

Alumno: Méndez Martín (••)
Docentes: Wolovick Nicolás; Bederían Carlos
Universidad Nacional de Córdoba (UNC)
Facultad de Matemática, Astronomía, Física y Computación (FaMAF)
Trabajo final del curso de posgrado de Computación Paralela

(•) Para mayor claridad se recomienda ver figuras en la versión digital.

(••) Email: martinmendez@mi.unc.edu.ar

I. Introducción

I-1. ¿Qué problema físico queremos simular?

En el presente trabajo se buscó mejorar la performance de simulaciones relacionadas al cálculo de autoestados y espectro de energías de hamiltonianos cuánticos. El modelo físico de estudio se trato de tres electrones en un punto cuántico doble. Enfocándonos en nanohilos con pares de puntos cuánticos embebidos, podremos partir del siguiente término Hamiltoniano que actúa en cada electrón

$$h(\vec{r_i}) = \frac{-1}{2m^*}(\nabla_i)^2 + V_t(x_i, y_i) + V_l(z_i) \tag{1}$$

en donde m^* es la masa efectiva y los términos V_t y V_l representan los confinamientos transversal y longitudinal del hilo respectivamente, desde el doble punto cuántico embebido, estos términos en forma explícita serán

$$V_t(x_i, y_i) = \frac{1}{2}m^*\omega^2(x_i + y_i)^2 \tag{2}$$

$$\begin{aligned} V_l(z_i) = -V_L \exp\left[-b_L\left(z_i + \frac{R}{2}\right)^2\right] - ... \\ ... - V_R \exp\left[-b_R\left(z_i - \frac{R}{2}\right)^2\right] \end{aligned} \tag{3}$$

donde R es la distancia entre centros de los puntos cuánticos, $b_{L,R}$ la parametrización de los tamaños de los puntos cuánticos (izquierdo y derecho respectivamente) y $V_{L,R}$ la profundidad de los potenciales (izquierdo y derecho respectivamente).

Finalmente, usando unidades atómicas y considerando un régimen de confinamiento transversal fuerte (donde $\omega = 1{,}0[\text{a.u.}] > V_{L,R}$) las simulaciones tridimensionales se pueden representar muy bien usando un modelo efectivo unidimensional y proponiendo una función de onda separable de la forma

$$\Psi(\vec{r}_1, \vec{r}_2, \vec{r}_3) = \psi(z_1, z_2, z_3)\phi_0(x_1, y_1)\phi_0(x_2, y_2)\phi_0(x_3, y_3) \tag{4}$$

donde $\phi_0(x_i, y_i)$ es la función de onda transversal (bidimensional) del estado fundamental de un único electrón para el nanohilo y $\psi(z_1, z_2, z_3)$ es la función de onda efectiva longitudinal. Entonces el hamiltoniano de análisis corresponderá,

$$H^{(0)}_{\text{eff}}(z_1, z_2, z_3) = \sum_{i=1}^{3} h_{\text{eff}}(z_i) \tag{5}$$

$$h_{\text{eff}}(z_i) = -\frac{1}{2}\frac{\partial^2}{\partial(z_i)^2} + V_l(z_i) \tag{6}$$

I-2. Agregando término perturbativo

Ahora bien, para ir un paso más allá y tratar de estudiar un comportamiento más complejo, incorporamos un término de interacción entre electrones de la forma

$$V^{\text{int}}_{\text{eff}}(z_1, z_2, z_3) = \left[V^{\text{int}}_{\text{eff}}(z_{1,2}) + V^{\text{int}}_{\text{eff}}(z_{1,3}) + V^{\text{int}}_{\text{eff}}(z_{2,3})\right] \tag{7}$$

$$V^{\text{int}}_{\text{eff}}(z_{i,j}) = \begin{cases} \sqrt{\frac{\pi}{2l^2}} \exp\left[(\zeta_{i,j})^2\right]\left[1 - \text{erf}(\zeta_{i,j})\right] & , \ z_{i,j} \tilde{z} \\ \frac{1}{z_{i,j}} & , \ z_{i,j} > \tilde{z} \end{cases} \tag{8}$$

donde $\zeta = \frac{z_{12}}{\sqrt{2l^2}}$ es tal que la distancia entre electrones $z_{i,j} = |z_i - z_j|$ es escalada por una longitud característica de confinamiento transversal dada por el siguiente valor de expectación $l = \sqrt{\langle\phi_0|x^2|\phi_0\rangle} = \frac{1}{\sqrt{\omega}}$ y donde $\text{erf}()$ denota la función error. Además, notemos que el comportamiento asintótico de el potencial de interacción evidencia la correcta dependencia Coulombiana como esperaríamos, para distancias entre electrones mayor a alguna distancia característica y, en cambio, no diverge para posiciones tales que $z_i \approx z_j$. Por lo tanto, consideramos la distancia entre centros de los puntos cuánticos R suficientemente grande como para que los autoestados estén bien localizados. Finalmente el hamiltoniano del sistema completo que permite modelizar el comportamiento de tres electrones en un punto cuántico doble con interacciones de a pares será de la forma,

$$H_{\text{tot}} = H^{(0)}_{\text{eff}}(z_1, z_2, z_3) + \lambda V^{\text{int}}_{\text{eff}}(z_1, z_2, z_3) \tag{9}$$

donde λ es el término perturbativo que deberá se tal que el término de interacción sea pequeño comparado con el hamiltoniano del sistema sin perturbar $0 \leq \lambda \leq 1$. Finalmente, podemos mencionar que dentro del factor λ está incluida la información del material a través de la permisividad dieléctrica relativa.

En el presente trabajo se utilizaron los valores mostrados en I para los parámetros de simulación.

CUADRO I
SETEO DE PARÁMETROS PARA SIMULACIÓN

Parámetro	Valor	Descripción
$\|V_L\|$	0.9 [a.u.]	profundidad del qdot izquierdo
$\|V_R\|$	0.6 [a.u.]	profundidad del qdot derecho
b_L	0.22	tamaño del qdot izquierdo
b_R	1.0	tamaño del qdot derecho
ω	1.0	frecuencia angular
l	1.0	largo caract. de confinamiento transv.
R	9.0	distancia entre centros de qdots

I-3. ¿Qué algoritmo queremos optimizar?

Los cálculos numéricos se llevaron a cabo utilizando el método denominado MCTDH por sus siglas en inglés *Multiconfiguration time-dependent Hartree* que es un algoritmo para propagar eficientemente paquetes de onda y por ello, es muy utilizado para estudiar dinámica cuántica. En particular, se utilizó el paquete de código abierto denominado *MCTDH Heidelberg package* que se ha estado desarrollando en la universidad de Heidelberg (Alemania) hace ya 18 años y permite tratar con grandes sistemas, es decir, sistemas con múltiples grados de libertad. Dicho paquete se encuentra desarrollado principalmente en los lenguajes de programación C y Fortran.

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                16
On-line CPU(s) list:   0-15
Thread(s) per core:    2
Core(s) per socket:    8
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 158
Model name:            Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz
Stepping:              12
CPU MHz:               4594.262
CPU max MHz:           5000.0000
CPU min MHz:           800.0000
BogoMIPS:              7200.00
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              16384K
NUMA node0 CPU(s):     0-15
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep
    mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2
    ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art
    arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc aperfmperf
    eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3
    sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt
    tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch
    invpcid_single intel_pt ssbd ibrs ibpb stibp tpr_shadow vnmi flexpriority
    ept vpid fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm mpx
    rdseed adx smap clflushopt xsaveopt xsavec xgetbv1 dtherm ida arat pln pts
    hwp hwp_notify hwp_act_window hwp_epp md_clear spec_ctrl intel_stibp
    flush_l1d arch_capabilities
command: cpuinfo
```

Fig. 1. Salida del comando **cpuinfo** nodo 18 en cluster Bandurria

II. RESULTADOS Y DISCUSIONES

II-A. ¿Cuál es el Hardware y Software disponible?

Dentro del Grupo de Teoría de la Materia Condensada (GTMC-FaMAF) tenemos a disposición un cluster llamado *Bandurria*, además, en el curso de Computación Paralela nos han proporcionado cuentas de usario para correr en dos servidores llamados *ZX81* y *Jupiterace*, ambos pertenecientes al CCAD y finalmente se cuenta con una computadora de escritorio personal. Se estudiaron las características de cada uno de los hardwars mencionados cuyos resultados se detallan a continuación.

II-A1. Cluster Bandurria

Teniendo en cuenta que este cluster cuenta con muchos nodos, y podríamos pensarlo como la unión de muchas PCs conectadas, se trató de identificar aquellos nodos similares para correr en paralelo y obtener resultados comparables entre si.

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                16
On-line CPU(s) list:   0-15
Thread(s) per core:    2
Core(s) per socket:    8
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 165
Model name:            Intel(R) Core(TM) i7-10700 CPU @ 2.90GHz
Stepping:              5
CPU MHz:               4692.498
CPU max MHz:           4800.0000
CPU min MHz:           800.0000
BogoMIPS:              5800.00
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              16384K
NUMA node0 CPU(s):     0-15
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic
    sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr
    sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc
    art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc
    aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx
    est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic
    movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm
    abm 3dnowprefetch invpcid_single intel_pt ssbd ibrs ibpb stibp
    ibrs_enhanced tpr_shadow vnmi flexpriority ept vpid fsgsbase
    tsc_adjust bmi1 avx2 smep bmi2 erms invpcid mpx rdseed adx smap
    clflushopt xsaveopt xsavec xgetbv1 dtherm ida arat pln pts hwp
    hwp_notify hwp_act_window hwp_epp pku ospke md_clear spec_ctrl
    intel_stibp flush_l1d arch_capabilities
```

Fig. 2. Salida del comando **cpuinfo** nodo 21 en cluster Bandurria

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                16
On-line CPU(s) list:   0-15
Thread(s) per core:    2
Core(s) per socket:    8
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 165
Model name:            Intel(R) Core(TM) i7-10700K CPU @ 3.80GHz
Stepping:              5
CPU MHz:               4857.385
CPU max MHz:           5100.0000
CPU min MHz:           800.0000
BogoMIPS:              7600.00
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              16384K
NUMA node0 CPU(s):     0-15
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic
    sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr
    sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc
    art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc
    aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx
    est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic
    movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm
    abm 3dnowprefetch invpcid_single intel_pt ssbd ibrs ibpb stibp
    ibrs_enhanced tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust
    bmi1 avx2 smep bmi2 erms invpcid mpx rdseed adx smap clflushopt
    xsaveopt xsavec xgetbv1 dtherm ida arat pln pts hwp hwp_notify
    hwp_act_window hwp_epp pku ospke md_clear spec_ctrl intel_stibp
    flush_l1d arch_capabilities
```

Fig. 3. Salida del comando **cpuinfo** nodo 22 en cluster Bandurria

La salida al ejecutar el comando **qhost -q** que es específico del administrador de colas llamado SGE que utiliza Bandurria nos permite obtener características globales del hardware donde cada columna de la imagen (ver figura 30) se refiere a una de ellas en particular como ser:

- HOSTNAME: Nombre del nodo de computación.

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                16
On-line CPU(s) list:   0-15
Thread(s) per core:    2
Core(s) per socket:    8
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 167
Model name:            11th Gen Intel(R) Core(TM) i9-11900K @ 3.50GHz
Stepping:              1
CPU MHz:               4861.633
CPU max MHz:           5300.0000
CPU min MHz:           800.0000
BogoMIPS:              7008.00
Virtualization:        VT-x
L1d cache:             48K
L1i cache:             32K
L2 cache:              512K
L3 cache:              16384K
NUMA node0 CPU(s):     0-15
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr
pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe
syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good
nopl xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor
ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2
x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm
abm 3dnowprefetch invpcid_single intel_pt ssbd ibrs ibpb stibp ibrs_enhanced
tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 avx2 smep
bmi2 erms invpcid mpx avx512f avx512dq rdseed adx smap avx512ifma clflushopt
avx512cd sha_ni avx512bw avx512vl xsaveopt xsavec xgetbv1 dtherm ida arat pln
pts hwp hwp_notify hwp_act_window hwp_epp hwp_pkg_req avx512vbmi umip pku
ospke avx512_vbmi2 gfni vaes vpclmulqdq avx512_vnni avx512_bitalg
avx512_vpopcntdq md_clear spec_ctrl intel_stibp flush_l1d arch_capabilities
```

Fig. 4. Salida del comando **cpuinfo** nodo 24 en cluster Bandurria

```
long@compute-0-18

Intel(R) processor family information utility, Version 2021.1 Build 20201112 (id: b9c9d2fc5)
Copyright (C) 2005-2020 Intel Corporation.  All rights reserved.

=====  Processor composition  =====
Processor name    : Intel(R) Core(TM) i9-9900K
Packages(sockets) : 1
Cores             : 8
Processors(CPUs)  : 16
Cores per package : 8
Threads per core  : 2

=====  Processor identification  =====
Processor       Thread Id.      Core Id.        Package Id.
0               0               0               0
1               0               1               0
2               0               2               0
3               0               3               0
4               0               4               0
5               0               5               0
6               0               6               0
7               0               7               0
8               1               0               0
9               1               1               0
10              1               2               0
11              1               3               0
12              1               4               0
13              1               5               0
14              1               6               0
15              1               7               0
=====  Placement on packages  =====
Package Id.     Core Id.        Processors
0               0,1,2,3,4,5,6,7         (0,8)(1,9)(2,10)(3,11)(4,12)(5,13)(6,14)(7,15)

=====  Cache sharing  =====
Cache   Size            Processors
L1      32  KB          (0,8)(1,9)(2,10)(3,11)(4,12)(5,13)(6,14)(7,15)
L2      256 KB          (0,8)(1,9)(2,10)(3,11)(4,12)(5,13)(6,14)(7,15)
L3      16  MB          (0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15)
```

Fig. 5. Salida del comando **lscpu** nodo 18 en cluster Bandurria

```
long@compute-0-21

Intel(R) processor family information utility, Version 2021.1 Build 20201112 (id: b9c9d2fc5)
Copyright (C) 2005-2020 Intel Corporation.  All rights reserved.

=====  Processor composition  =====
Processor name    : Intel(R) Core(TM) i7-10700
Packages(sockets) : 1
Cores             : 8
Processors(CPUs)  : 16
Cores per package : 8
Threads per core  : 2

=====  Processor identification  =====
Processor       Thread Id.      Core Id.        Package Id.
0               0               0               0
1               0               1               0
2               0               2               0
3               0               3               0
4               0               4               0
5               0               5               0
6               0               6               0
7               0               7               0
8               1               0               0
9               1               1               0
10              1               2               0
11              1               3               0
12              1               4               0
13              1               5               0
14              1               6               0
15              1               7               0
=====  Placement on packages  =====
Package Id.     Core Id.        Processors
0               0,1,2,3,4,5,6,7         (0,8)(1,9)(2,10)(3,11)(4,12)(5,13)(6,14)(7,15)

=====  Cache sharing  =====
Cache   Size            Processors
L1      32  KB          (0,8)(1,9)(2,10)(3,11)(4,12)(5,13)(6,14)(7,15)
L2      256 KB          (0,8)(1,9)(2,10)(3,11)(4,12)(5,13)(6,14)(7,15)
L3      16  MB          (0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15)
```

Fig. 6. Salida del comando **lscpu** nodo 21 en cluster Bandurria

```
long@compute-0-22

Intel(R) processor family information utility, Version 2021.1 Build 20201112 (id: b9c9d2fc5)
Copyright (C) 2005-2020 Intel Corporation.  All rights reserved.

=====  Processor composition  =====
Processor name    : Intel(R) Core(TM) i7-10700K
Packages(sockets) : 1
Cores             : 8
Processors(CPUs)  : 16
Cores per package : 8
Threads per core  : 2

=====  Processor identification  =====
Processor       Thread Id.      Core Id.        Package Id.
0               0               0               0
1               0               1               0
2               0               2               0
3               0               3               0
4               0               4               0
5               0               5               0
6               0               6               0
7               0               7               0
8               1               0               0
9               1               1               0
10              1               2               0
11              1               3               0
12              1               4               0
13              1               5               0
14              1               6               0
15              1               7               0
=====  Placement on packages  =====
Package Id.     Core Id.        Processors
0               0,1,2,3,4,5,6,7         (0,8)(1,9)(2,10)(3,11)(4,12)(5,13)(6,14)(7,15)

=====  Cache sharing  =====
Cache   Size            Processors
L1      32  KB          (0,8)(1,9)(2,10)(3,11)(4,12)(5,13)(6,14)(7,15)
L2      256 KB          (0,8)(1,9)(2,10)(3,11)(4,12)(5,13)(6,14)(7,15)
L3      16  MB          (0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15)
```

Fig. 7. Salida del comando **lscpu** nodo 22 en cluster Bandurria

- ARCS: Arquitectura-funcional.
- NCPU,NSOC,NCOR,NTHR: Números de cores lógicos, número de socalos, número de cores físicos y número de hilos por cada nodo, respectivamente.
- LOAD: carga del nodo en tiempo de ejecución de qhost.
- MEMTOT: Memoria total del nodo.
- MEMUSE: Memoria usada en tiempo de ejecución de qhost.
- SWAPTO: Memoria de intercambio total.
- SWAPUS: Memoria de intercambio en uso.

Entonces, identificamos los nodos 18, 21, 22 y 24 como aquellos que podrían ser similares, ya que poseen el mismo número cores lógicos, físicos y de hilos (NCPU=16,NCOR=8,NTHR=16) y misma arquitectura (ARCH=lx-amd64). Luego, para estos cuatro nodos se diseño un script para ejecutar los comandos **cpuinfo** y **lscpu** en cada nodo de forma excluyente y los resultados se muestran en las figuras 1, 2, 3 y 4 para el comando **cpuinfo**; y las figuras 5, 6, 7 y 8 para el comando **lscpu**. Con estos resultados vemos que hay diferencias significativas entre los cuatro nodos estudiados. Por un lado, podemos notar que si bien los nodos 18 y 24 cuentan con chips i9, funcionan a diferente frecuencia de procesamiento y la distribución de memoria en sus niveles de cache son distintos. Por otro lado, los nodos 21 y 22 son iguales, es decir, cuentan con un mismo chip i7, mismo rango de frecuencias de procesamiento

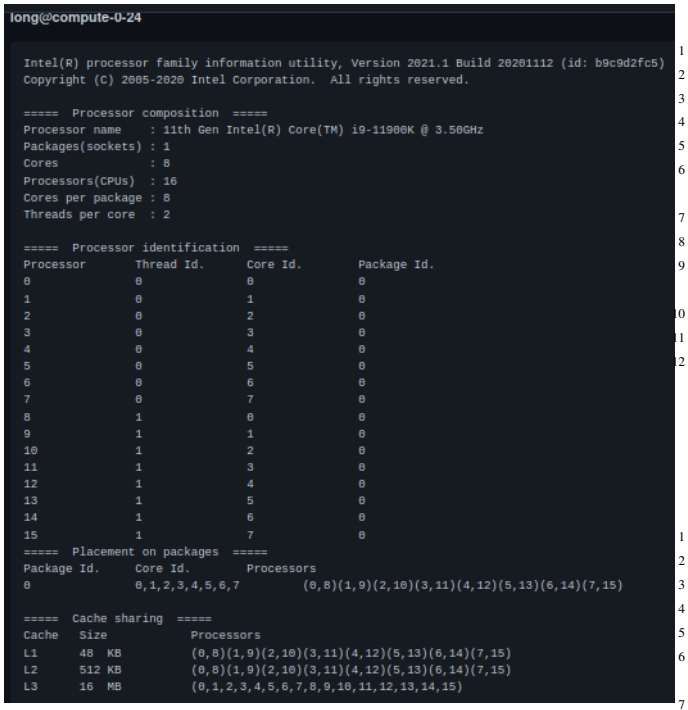

Fig. 8. Salida del comando **lscpu** nodo 24 en cluster Bandurria

y misma distribución de memoria. Entonces, estrictamente hablando, deberíamos usar sólo los nodos 21 y 22 (como se ha realizado en el presente trabajo) ya que, presentan diferencias despreciables (diferencias de fabricación), sin embargo, en términos prácticos para realizar las simulaciones más rápido necesitaríamos utilizar más de dos nodos, por ello podrían utilizarse los nodos 18 y 24 pero deberemos tener en cuenta las diferencias sustanciales que existen entre todos los nodos al momento de comparar rendimientos.

II-A2. Servidores ZX81 y Jupiterace

Estos servidores pertenecen al CCAD ubicado en la provincia de Córdoba y ambos cuentan con el mismo procesador y memoria con las siguientes características principales **Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz, 128 GiB de RAM DDR4-2133, SSD 240 GiB, HDD 2*4 TiB** cuya microarquitectura es del tipo **Broadwell**. Entonces, estos servidores cuentan con un procesador de 14 núcleos (cada uno con 2 threads lógicos) distribuidos en dos pastillas equitativamente, haciendo un total de 28 threads por servidor.

II-A3. Computadora personal

Para obtener información de este recurso ejecutamos, por un lado, los comandos **x86info** y **x86info –cache** y los resultados se muestran en II-A3 y II-A3. De estas salidas podemos buscar específicamente el procesador (ver página de Intel) para ver sus características, donde vemos que cuenta con 4 cores y 4 threads, frecuencias máximas y mínimas de procesamiento son 3,3[GHz] y 2,7[GHz] respectivamente, ancho de banda de memoria máxima de 34,1[GB/s] y arquitectura de tipo Skylake. De este último dato podemos ver cuántas unidades de punto flotante cuenta esta arquitectura (ver página de Wikichip) y vemos que cuenta con 2 FMA (suma y multiplicación de números flotantes), los cuales se ejectuan en 2 puertos a la vez, esto nos da una performance pico de 86,4[GFLOPs] para operaciones flotantes de tipo float64.

Listing 1: Salida de comando x86info

```
mendez@mendez-famaf:~$ x86info
x86info v1.31pre  Dave Jones 2001-2011
Feedback to <davej@redhat.com>.

Found 4 identical CPUs
Extended Family: 0 Extended Model: 5 Family: 6 Model
    : 94 Stepping: 3
Type: 0 (Original OEM)
CPU Model (x86info's best guess): Unknown model.
Processor name string (BIOS programmed): Intel(R)
    Core(TM) i5-6400 CPU @ 2.70GHz

Total processor threads: 4
This system has 1 dual-core processor with hyper-
    threading (2 threads per core) running at an
    estimated 2.70GHz
```

Listing 2: Salida de comando x86info –cache

```
mendez@mendez-famaf:~$ x86info --cache
x86info v1.31pre  Dave Jones 2001-2011
Feedback to <davej@redhat.com>.

Found 4 identical CPUs
Extended Family: 0 Extended Model: 5 Family: 6 Model
    : 94 Stepping: 3
Type: 0 (Original OEM)
CPU Model (x86info's best guess): Unknown model.
Processor name string (BIOS programmed): Intel(R)
    Core(TM) i5-6400 CPU @ 2.70GHz

Cache info
TLB info
 Data TLB: 4KB pages, 4-way associative, 64 entries
 Data TLB: 4KB pages, 4-way associative, 64 entries
 64 byte prefetching.
Found unknown cache descriptors: 63 76 b6 c3 ff
Total processor threads: 4
This system has 1 dual-core processor with hyper-
    threading (2 threads per core) running at an
    estimated 2.70GHz
```

Por otro lado, ejecutamos el comando **lsb_release -a** para obtener información del sistema operativo de este recurso obteniendo el resultado que se muestra en II-A3.

Listing 3: Salida de comando lsb_release -a

```
mendez@mendez-famaf:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 22.04.1 LTS
Release:        22.04
Codename:       jammy
```

Además, para determinar la topología del recurso se corrió el comando **lstopo** obteniendo el resultado que se muestra en la figura 9, donde podemos ver las capacidades de las lineas de cache L1, L2 y L3; distribución de cores reales y lógicos; y almacenamiento externo (disco rígido).

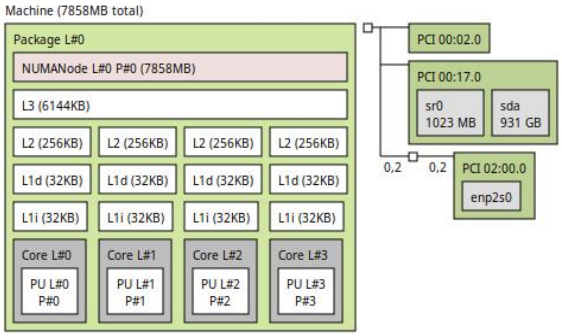

Fig. 9. Salida del comando **lstopo** computadora personal

II-B. Asegurando resultados numéricos aumentando precisión de los métodos numéricos

Se llevo a cabo un primer análisis de convergencia numérica para asegurarnos de que los resultados computacionales sean correctos o estén dentro de las tolerancias impuestas. Para ello, se tuvieron que aumentar las precisiones de los integradores que utiliza el paquete como ser Constant Mean-Field scheme (CMF), Runge-Kutta integrator for SPFs (RK8/spf) e integrador de Davinson para vector de coeficientes (rrDAV/A). En todos los casos se utilizó una precisión del orden de $\mathcal{O}(10^{-9})$, lo cual claramente incrementa el coste computacional a expensas de obtener mejores resultados de simulación.

II-C. Construyendo un environment específico

Cabe mencionar que la creación de un environment específico es sumamente importante pues, permite asegurar una correcta comparación de los resultados al simular el mismo problema en distintos recursos y lograr elegir fielmente la configuración que permite obtener más rendimiento.

Al realizar corridas de un mismo problema en distintos recursos (hardware) y luego comparar los resultados, es fundamental que el programa (en nuestro caso el paquete MCTDH) sea compilado bajo las "*mismas condiciones*" en todos los recursos, donde por "*mismas condiciones*" nos referimos a compilar con la misma versión de: compiladores, programas, librerías, APIs, etc. Entonces, la herramienta elegida para lograr esto fue **CONDA**, la cual nos permitió crear un environment específico en cada recurso donde se instaló todo el software necesario respetando una misma versión. Luego, dentro de cada environment, se llevó a cabo la compilación del paquete MCTDH, de esta manera lograríamos obtener un mismo código binario para ejecutar en las simulaciones, el cual estaría compilado bajo las mimas características (aunque obviamente estos programas tendrían diferencias en cuanto a arquitecturas específicas de cada recurso). En el siguiente link se puede consultar mayor información del procedimiento mencionado, como así también distintas problemáticas surgidas (problemas específicos al emplear CONDA) con su correspondiente solución.

II-D. Ahora si... Mejorando performance

II-D1. Vuelta de tuerca: Usando tratamiento de partículas idénticas y activando paralelismo

Uno de las primeras pruebas realizadas para verificar si se lograba disminuir el tiempo de computo fue implementar el tratamiento de *partículas idénticas* en el contexto de la física cuántica. Básicamente, se explotó el hecho de que no estamos interesados en incorporar las propiedades de indistinguibilidad de las partículas al modelo computacional (en nuestro caso particular, serían propiedades que se aplicarían a los electrones). Por ello, tratar o no a las partículas como indistinguibles, en términos físicos sería irrelevante, sin embargo, en términos computacionales no, pues el paquete de MCTDH trata de forma diferente ambas configuraciones, es más, el rendimiento (medido en tiempo de CPU) mejora si las simulaciones se realizan considerando a las partículas como idénticas, ya que, simplemente cada una de ellas es copia de la otra y el paquete sólo considera un subsistema en particular, es decir que, como el Hamiltoniano del sistema tiene propiedades de simetría, las preserva al momento de evolucionar el sistema y si, por ejemplo, se comienza con un estado totalmente simétrico, el estado continuará siéndol (y viceversa si se comenzara con un estado totalmente antisimétrico).

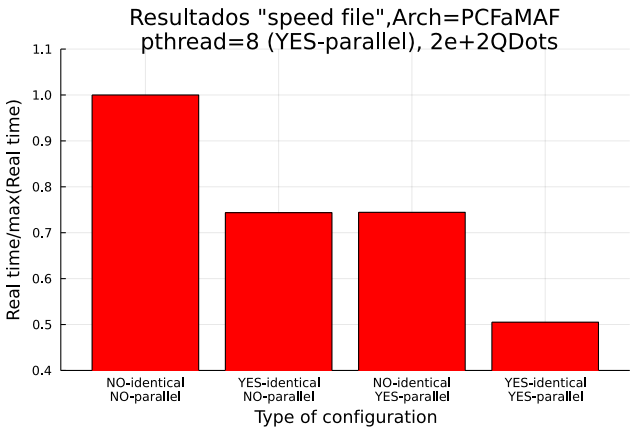

Fig. 10. Mejoras algorítmicas y paralelización

Por otro lado, se comparó qué tanto disminuía el tiempo de computo del programa paralelizándolo o no. La forma de paralelización elegida fue aquella basada en modelo *POSIX threads* cuya API está implementada en el sistema operativo. En particular, se consideraron ocho threads para las simulaciones que requerían paralelización.

Entonces, teniendo en cuenta que se trataba de una comparación estimativa con el fin de determinar dónde focalizar la atención para mejorar la performance del programa, se consideró un sistema más pequeño para disminuir el tiempo de simulación entonces, se midieron los tiempos de CPU para un sistema de dos electrones en un doble punto cuántico y donde los parámetros de estas simulaciones se presentan en las tablas I y II. Además, como se puede observar en el código 4, el análisis de rendimiento se realizó midiendo el tiempo de CPU para una simulación que calcula las autoenergías del hamiltoniano del sistema (2 electrones en un doble punto cuántico con perturbación Coulombiana) utilizando el método de relajación (usando la keyword relaxation=follow dentro de la sección RUN-SECTION) implementado en el paquete MCTDH para una función de onda inicial fija (ver sección INIT_WF-SECTION) y una configuración de integradores específica (ver sección INTEGRATOR-SECTION).

CUADRO II
SETEO DE PARÁMETROS PARA SIMULACIÓN

Parámetro	Valor	Descripción
λ	0.05	perturbación
N_{DVR}	501	puntos de la grilla Sine-DVR
L_{DVR}	50	ancho de la grilla Sine-DVR
N_{X_i}	2	grados de libertad

Listing 4: Program: Configuración fija de simulación

```
INTEGRATOR-SECTION
     CMF=0.1,1.0d-6
     RK8/spf=1.0d-8
     SIL=500,1.0D-6
end-integrator-section
INIT_WF-SECTION
     build
          X1 eigenf heff pop=1
          X2 map     X1
     end-build
     A-coeff
          2 3 (1.0,0.0)
          3 2 (-1.0,0.0)
     end-A-coeff
end-init_wf-section
```

Los resultados obtenidos se muestran en la figura 10 donde podemos ver que el tiempo de CPU disminuye notablemente

(casi en un 50 %) si simulamos con una configuración de partículas idénticas y activamos el paralelismo. Cabe aclarar que los resultados fueron normalizados con el tiempo máximo obtenido ($t_{max} = 804{,}35$ [s]). Ahora bien, esta conclusión solo sería válida para esta simulación particular, sin embargo, como criterio se optó por tomar esta configuración (partículas idénticas y paralelismo) y a partir de esta intentar mejorar el rendimiento de la simulación con otras técnicas.

*II-D2. **Escalamiento del sistema según el parámetro perturbativo***

Para analizar cómo escalaba el tiempo de CPU del problema a medida que aumentaba el parámetro perturbativo λ se graficaron los resultados variando este parámetro y registrando los tiempos de ejecución, los resultados se muestran en la figura 11, donde se puede notar que, a grandes rasgos, a medida que *prendemos* más la perturbación más tiempo de CPU es requerido para resolver el problema. En la mayor parte del trabajo fue utilizado un parámetro de $\lambda = 0{,}2$ para obtener un tiempo razonable de simulación que permita extraer soluciones, sin embargo, para testeo final de mejor configuración obtenida fue necesario incrementar notablemente el parámetro λ para analizar qué tanto influye el tamaño del sistema en las simulaciones.

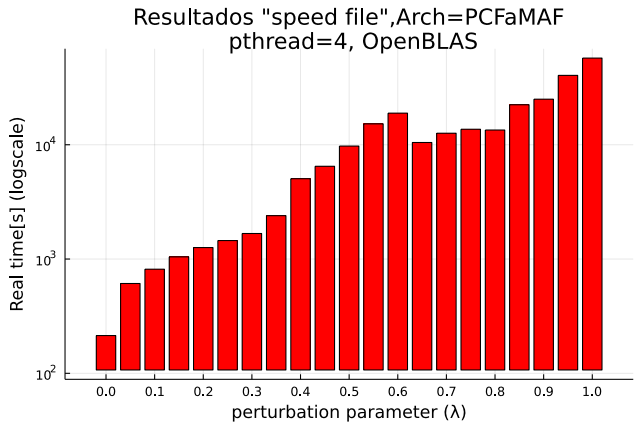

Fig. 11. Escalamiento del problema según parámetro perturbativo

*II-D3. **Optimizando operaciones algebraicas mediante OpenBLAS***

OpenBLAS es una implementación de código abierto para la API llamada BLAS (Basic Linear Algebra Subprograms) con optimización especializadas según diferentes tipos de procesadores. Esta implementación consigue un rendimiento comparable a la API llamada **Intel MKL** (Math Kernel Library), esto es cierto sobre todo en la parte BLAS, mientras que la parte LAPACK se queda atrás ya que, MKL optimiza LAPACK y BLAS, mientras que OpenBLAS sólo optimiza BLAS. En las máquinas que soportan el conjunto de instrucciones AVX2, OpenBLAS puede lograr un rendimiento similar al de MKL, pero actualmente casi no hay bibliotecas de código abierto comparables a MKL en CPUs con el conjunto de instrucciones AVX512 (notemos que esto es importante en los clusters que tendremos como recursos, ya que algunos de ellos cuentan con dichas unidades vectoriales modernas y podríamos sacar ventaja de ellas. Asimismo, en la sección IV se comenta sobre un análisis comparativo entre OpenBLAS y MKL). Para analizar la performance del problema al simular con y sin OpenBLAS se partió de la configuración mostrada en la tabla III. Además, notando que OpenBLAS utiliza variables especificas de OpenMP para paralelizar sus algoritmos debemos asignar valores a ciertas variables de entorno para asegurar una correcta paralelización (lograr una buena comunicación entre OpenMP y OpenBLAS) y existen varias configuraciones posibles al momento de utilizar OpenBLAS y por ello se ejecutaron cada una de ellas y se tomaron los tiempos de CPU demandados, luego, el mejor resultado se comparó con las simulaciones sin utilizar OpenBLAS.

Primeramente se seteó la configuración de variables de entorno (ver II-D3) que se mantendrá fija para todas las simulaciones. Luego, se analizaron tres configuraciones de ejecución utilizando OpenBLAS, las cuales se detallan en II-D3.

Listing 5: Configuración fija de variables de entorno OpenMP y Posix

```
# OpenMP
    export OMP_SCHEDULE="auto"
    export OMP_STACKSIZE="512M"
# OpenBLAS
    export USE_THREAD=0
    export USE_LOCKING=1
# Posix threads
    usepthreads=4
```

Listing 6: Configuración de paralelización para OpenBLAS

```
# CONFIGURATION 01
    export OMP_NUM_THREADS=1
    export OPENBLAS_NUM_THREADS=4
# CONFIGURATION 02
    export OMP_NUM_THREADS=4
    export OPENBLAS_NUM_THREADS=1
# CONFIGURATION 03
    export OMP_NUM_THREADS=4
    export OPENBLAS_NUM_THREADS=4
```

Los resultados obtenidos se pueden observar en las figuras 12 (normalizando respecto al tiempo máximo obtenido $t_{max} = 10216$ [s]) y 13 (normalizando respecto al tiempo máximo obtenido $t_{max} = 10416$ [s]) donde se puede observar que el tiempo de de CPU es mucho menor (hasta un 20 % más rápido) al utilizar OpenBLAS lo cual era de esperarse, sin embargo, rescatamos la importancia de simular con una correcta configuración de variables de entorno para efectivamente visualizar la mejora de rendimiento, donde podemos ver que bajo la configuración óptima reducimos más de un 20 % el tiempo de CPU.

CUADRO III
SETEO DE PARÁMETROS PARA SIMULACIÓN

Parámetro	Valor	Descripción
λ	0.5	perturbación
N_{DVR}	501	puntos de la grilla Sine-DVR
L_{DVR}	50	ancho de la grilla Sine-DVR
N_{X_i}	3	grados de libertad

*II-D4. **Analizando el rendimiento con muchos POSIX threads y usando OpenBLAS***

A continuación se llevaron a cabo simulaciones aumentando el número de Posix threads y midiendo el tiempo de CPU requerido para cada configuración. Las configuraciones de variables de entornos para OpenBLAS que se mantuvieron fijas para cada simulación fueron las mismas que las que se muestran en II-D3 y los parámetros de simulación fueron los mismos que los mostrados en III. Ahora bien, teniendo en cuenta que mientras el paquete MCTDH utiliza paralelización

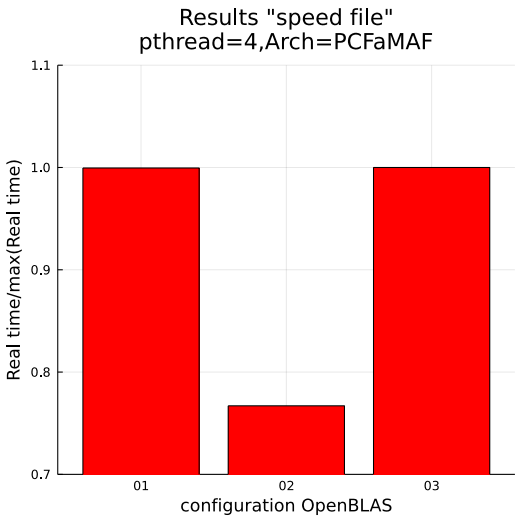

Fig. 12. Tiempo de CPU para distintas configuraciones de OpenBLAS

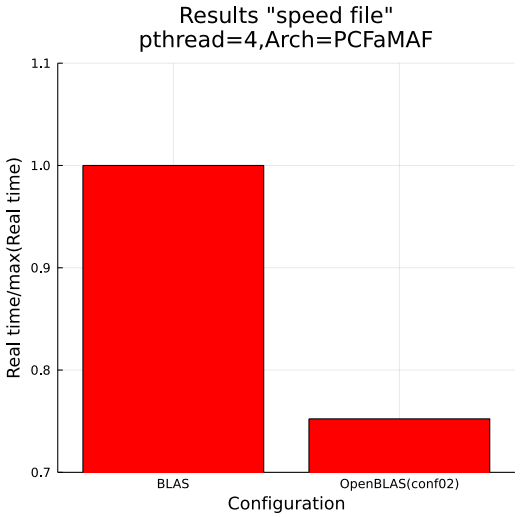

Fig. 13. Performance de OpenBLAS vs BLAS

vía Posix threads, OpenBLAS lo hace mediante OpenMP, se utilizó la configuración de ejecución para OpenBLAS de mayor performance obtenida en la sección II-D3, es decir, la configuración 02 según II-D3. Luego, se fue aumentando el valor de los threads lógicos para aumentar la paralelización tomando cuatro configuraciones diferentes, las cuales se detallan en II-D4.

Listing 7: Configuración de POSIX threads y OpenBLAS/OpenMP

```
# CONFIGURACIN 01
    # POSIX threads
        usepthreads=1
    # OpenBLAS y OpenMP
        export OMP_NUM_THREADS=1
        export OPENBLAS_NUM_THREADS=1
# CONFIGURACIN 02
    # POSIX threads
        usepthreads=2
    # OpenBLAS y OpenMP
        export OMP_NUM_THREADS=2
        export OPENBLAS_NUM_THREADS=1
# CONFIGURACIN 03
    # POSIX threads
        usepthreads=3
    # OpenBLAS y OpenMP
        export OMP_NUM_THREADS=3
        export OPENBLAS_NUM_THREADS=1
# CONFIGURACIN 04
    # POSIX threads
        usepthreads=4
    # OpenBLAS y OpenMP
        export OMP_NUM_THREADS=4
        export OPENBLAS_NUM_THREADS=1
```

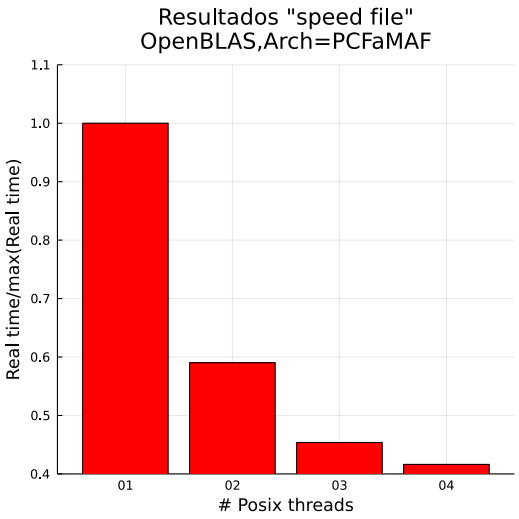

Fig. 14. Aumento de paralelismo con POSIX threads y OpenBLAS

Los resultados obtenidos se muestran en la figura 14 (donde hemos normalizado respecto al tiempo máximo obtenido de $t_{max} = 18821$ [s]), donde podemos observar que el tiempo de CPU disminuye a medida que aumentamos el número de POSIX threads, incluso hasta en $60\,\%$ aproximadamente. Además, para determinar que tan grande es la parte paralela del algoritmo de MCTDH (asociado al problema particular simulado) y partiendo de los resultados obtenidos en la figura 14 se calculó el Speedup en función del número de procesadores (número de POSIX threads) y los resultados discretos se ajustaron según la ley de Amdahl, la cual nos dice que:

$$S(N_p) = \frac{1}{(q + p/N_p)} \tag{10}$$

donde q y p denotan la parte serial y paralela del sistema tal que se cumpla $(q + p) = 1$, S es el speedup y N_p es el número de procesadores (que en nuestro caso sería el número de POSIX threads) entonces, con los datos experimentales calculamos el Speedup real de la siguiente manera:

$$S^{real}(N_p) = \frac{T_{start}}{T_{end}(N_p)} \tag{11}$$

donde T_{start} es el tiempo de CPU que tarda la simulación sin paralelización y $T_{end}(N_p)$ es el tiempo de CPU que tarda la simulación cuando paralelizamos usando N_p POSIX threads. El resultado se muestra en la figura 15 donde obtuvimos una parte serial del $30\,\%$ y una parte paralela del $70\,\%$.

Ahora bien, el resultado anterior es completamente válido para el recurso computacional II-A3 donde efectivamente explotamos los recursos disponibles, sin embargo, para analizar los límites de paralelización es necesario ejecutar el problema en un servidor, por ejemplo *JupiterAce*, donde se realizó un escalamiento del problema hasta un total de 28 threads obteniendo los resultados que se muestran en la figuras 16 (normalizando respecto al tiempo máximo de $t_{max} = 2721{,}33$ [s]) y 17, donde podemos apreciar que el sistema alcanza la aceleración máxima a partir de los 4 threads, lo que nos sugiere que para mayor numero de POSIX threads el

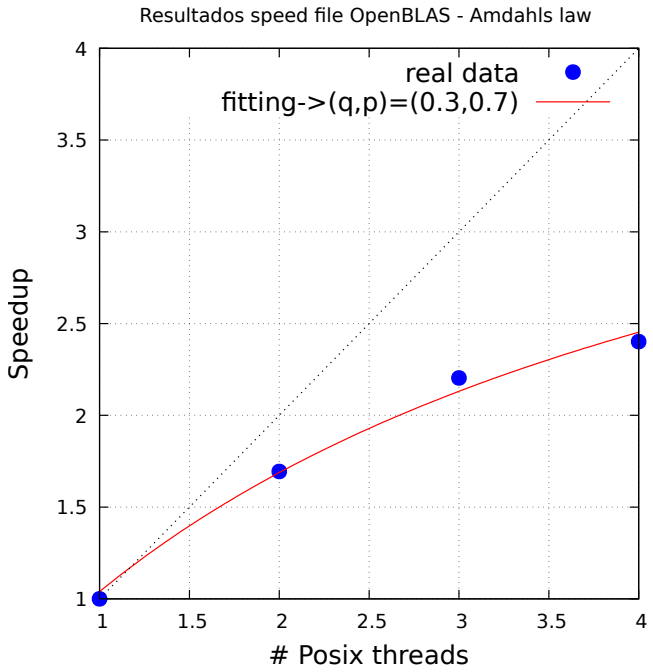

Fig. 15. Speedup vs POSIX threads

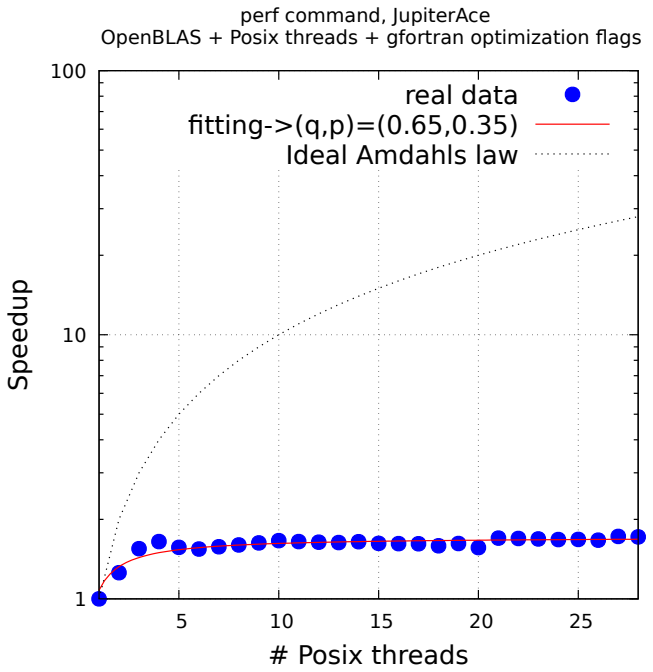

Fig. 17. Speedup vs POSIX threads

programa, no solo no se ejecutará más rápido sino que, al utilizar más threads estaríamos desperdiciando recursos que no mejorarían la performance de las simulaciones de forma significativa. Con este nuevo análisis a partir de un recurso computacional más potente, obtuvimos una parte serial del $65\,\%$ y una parte paralela del $35\,\%$, lo cual evidenciaría un margen acotado de paralelización del problema, notemos además que el tamaño de las partes serial y paralelas se han reducido a la mitad aproximadamente respecto al análisis en la computadora personal II-A3 (ver 15).

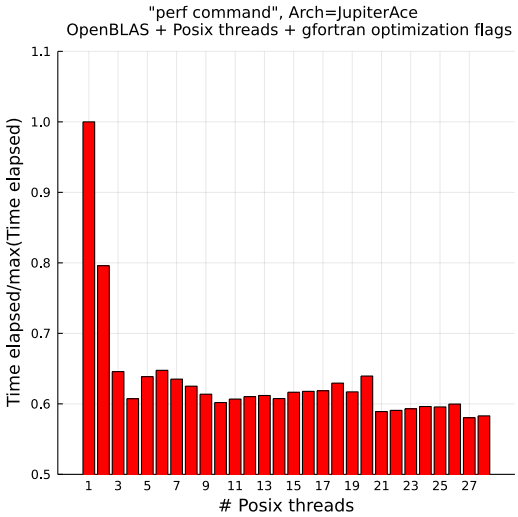

Fig. 16. Aumento de paralelismo con POSIX threads y OpenBLAS

II-D5. Modificación del schedule para paralelización de OpenBLAS con OpenMP

Como mencionamos anteriormente, OpenBLAS utiliza las variables de entorno propias de OpenMP para paralelizar sus algoritmos por lo que analizar el schedule dentro de OpenMP podría ayudar a definir la mejor configuración de distribución de threads y asignación de memoria y lograr así el menor tiempo de CPU posible, para ello se partió de la misma configuración de parámetros mostrados en la tabla III y las configuraciones de variables de entorno, para OpenBlas y POSIX threads, que se mantuvieron fijas para todas las simulaciones se muestran en II-D5.

Listing 8: Configuración de paralelización para OpenBLAS y POSIX threads

```
# OpenMP
    export OMP_NUM_THREADS=2
# OpenBLAS
    export OPENBLAS_NUM_THREADS=1
    export USE_THREAD=0
    export USE_LOCKING=1
# POSIX threads
    usepthreads=2
```

En primera instancia se modificó el schedule para OpenMP analizando los diferentes métodos sin especificar el tamaño de porción de código que se usaría para paralelizar, es decir, se utilizaron los tamaños por default (las tres configuraciones se pueden encontrar en II-D5). Esta elección de configuraciones se llevo a cabo para visualizar la distribución de threads dentro del paralelismo con OpenMP en OpenBLAS. Los resultados se pueden observar en la figura 18, los cuales se encuentran adimensionalizados con la peor condición ($t_{max} = 11792$ [s]). Vemos, en este caso que respecto al tiempo máximo de simulación todas ocuparon más del $90\,\%$ del mismo, lo que nos estaría diciendo que no existen diferencias significativas.

Listing 9: Configuración de schedule para paralelización de OpenBLAS con OpenMP

```
# configuracion 01
    export OMP_SCHEDULE="auto"
# configuracion 02
    export OMP_SCHEDULE="static,default"
# configuracion 03
    export OMP_SCHEDULE="guided"
```

En segunda instancia, sí se estudió cómo influía el tamaño de las porciones de código y, en particular, se utilizaron distribuciones de hilos dinámicas y estáticas. Los valores de chunksize simulados fueron potencias de dos como ser

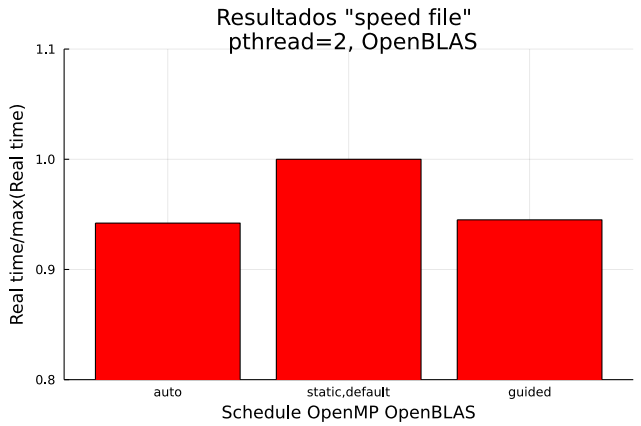

Fig. 18. Comparación de Performance para distintos Schedules en OpenBLAS/OpenMP

chunksize $= \{32, 64, 128, 256, 512, 1024\}$ y los resultados se muestran en las figuras 19 y 20, donde hemos adimensionalizado los resultados con el máximo tiempo de cpu obtenido ($t_{max} = 1609$ [s]), entonces, podemos notar que en todas las configuraciones la simulación ocupo más de un $80\,\%$ del tiempo máximo, lo que nos estaría diciendo que no hay diferencias significativas, sin embargo, si tuviésemos que elegir alguna configuración el menor tiempo se obtuvo para un $chunksize = 512$ con el schedule estático.

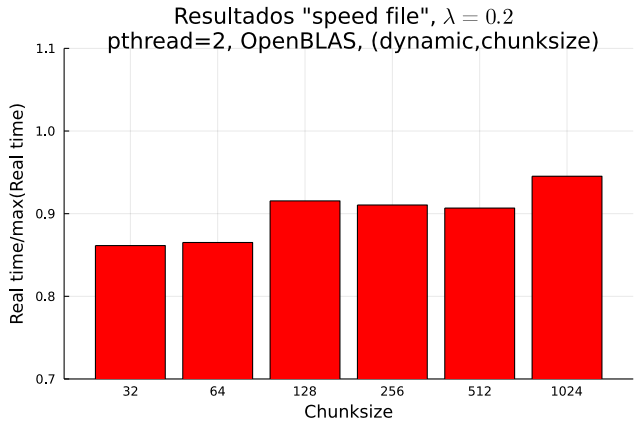

Fig. 19. Comparación de Performance para distintos chunksizes de dynamic schedules en OpenBLAS/OpenMP

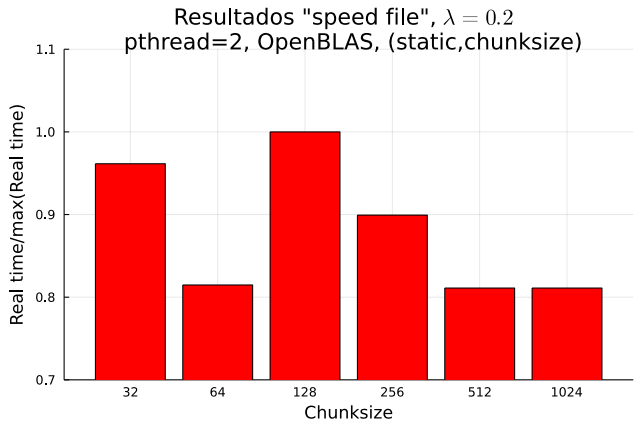

Fig. 20. Comparación de Performance para distintos chunksizes de static schedules en OpenBLAS/OpenMP

Por último, se analizó cómo influía la variable **OMP_STACKSIZE** que controla el tamaño en memoria de los stacks para cada OMP-threads, para ello se realizaron corridas con los siguientes valores del parámetro OMP_STACKSIZE $= \{\text{default}, 128\text{K}, 256\text{K}, 512\text{K}, 8\text{M}, 64\text{M}, 512\text{M}\}$ y los resultados se muestran en la figura 21 (donde los resultados se han normalizado respecto a la peor condición de tiempo máximo $t_{max} = 1403{,}09$ [s] y referencias a cache de $5{,}608\,\%$). Aquí, notamos que todas la simulaciones ocuparon más del $90\,\%$ del tiempo máximo de simulación y las diferencias no son significativas, sin embargo, al analizar las referencias a cache notamos que, para el tamaño de 512M se obtuvieron casi un $50\,\%$ menos de referencias perdidas respecto a las demás configuraciones.

Estos resultados, en principio, poco fructíferos podrían deberse a que OpenBLAS, al tener optimizaciones específicas para cada arquitectura dentro de su compilación, ya cuenta con tamaños de memoria específicos que permiten obtener los mejores rendimientos y cualquier mejora que realicemos, en tiempos de ejecución, no producirá resultados significativos.

II-D6. ¿Y si usamos OpenMP threads en MCTDH?¿Qué tan rápido puede ser respecto a usar POSIX threads?

Se realizaron simulaciones partiendo de las mejores configuraciones obtenidas hasta el momento como ser utilizando OpenBLAS y schedule de OpenMP "static,512"(más aquellas configuraciones fijas mostradas en II-D5), sin embargo, paralelizando MCTDH con POSIX threads y con OpenMP. En el primer caso tendremos una forma de paralelización híbrida pues MCTDH utilizará POSIX threads mientras que OpenBLAS utilizará OpenMP, y en el segundo caso tendremos una forma de paralelización usando OpenMP en su totalidad (es necesario mencionar que para ejecutar OpenMP, debemos compilar el paquete MCTDH de forma específica, esto puede consultarse en el manual del paquete). Los resultados se muestran en la figura 22, donde se han adimensionalizado los tiempos de simulación con el tiempo máximo obtenido ($t_{mx} = 1390$ [s]), y donde no se detectaron diferencias significativas, pues los dos enfoques demandan más del $90\,\%$ del tiempo de simulación, sin embargo, el mejor rendimiento se obtuvo para la paralelización con POSIX.

Por otro lado, para evidenciar esta mejora de performance de POSIX frente a OpenMP, se utilizó el hecho de que el programa no mejora su rendimiento para número de threads mayores a 4 threads (recordar figura 16) entonces, se compararon las simulaciones utilizando 2 y 4 threads para los casos en que se realiza paralelización con POSIX y OpenMP. Los resultados se muestran en la figura 23 (los cuales han sido normalizados respecto al tiempo máximo de CPU de $t_{mx} = 1413{,}49$ [s] y máximo porcentaje de referencias a cache fallidas $5{,}052\,\%$) donde vemos que a pesar de que en todos los casos el tiempo de simulación ocupa más de un $90\,\%$ del tiempo máximo de simulación, al estudiar la referencias a cache vemos que para el caso paralelización en 4 threads+POSIX estas referencias fallidas se reduce en casi un $40\,\%$ respecto a la peor condición.

II-D7. ¿Aprovechando el poder mágico de los compiladores?

Teniendo en cuenta que, siempre que sea posible, conviene aprovechar al máximo las ventajas que nos proporciona el compilador, pues con muy poco trabajo podríamos, en principio, lograr mejoras de rendimiento significativas. Entonces, bajo este enfoque se realizaron simulaciones agregando flags

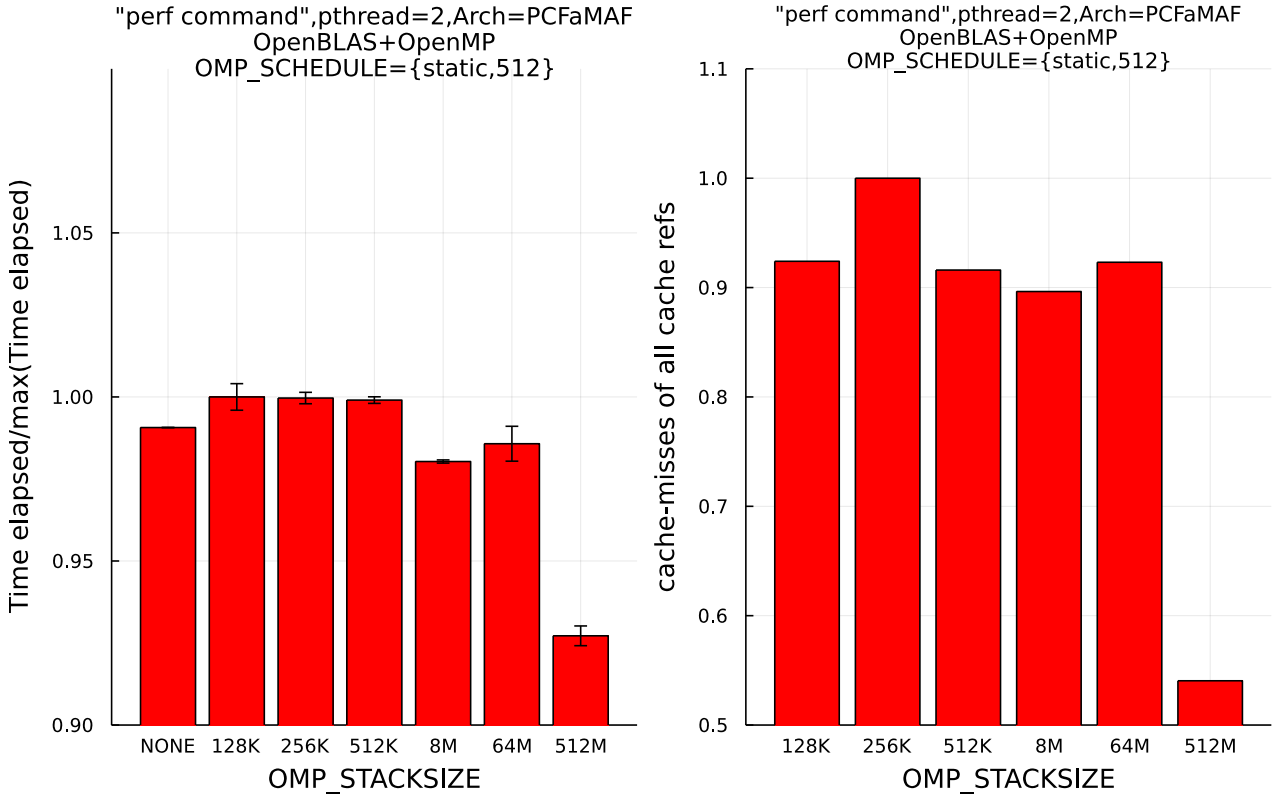

Fig. 21. Comparación de Performance para distintos OMP_STACKSIZE en OpenBLAS/OpenMP

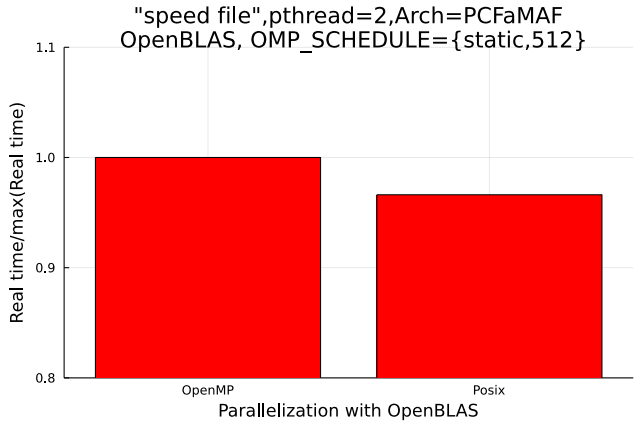

Fig. 22. Comparación de Performance paralelizando MCTDH con POSIX threads y con OpenMP

de compilación en los compiladores de FORTRAN y C, específicamente se utilizó la versión 12.2.0 de los compiladores gfortran y gcc. Las configuraciones de flags utilizadas se muestran en II-D7.

Listing 10: Configuración flags de compilación para gfortran y gcc

```
# FLAGS COMUNES A TODAS LAS CONFIGURACIONES
MCTDH_FFLAGS_OPT=${MCTDH_FFLAGS_0PT_01}" -fconvert=
    little-endian -frecord-marker=4 -funroll-loops
    -fno-align-commons"
# CONFIGURACIN original
MCTDH_FLAGS_OPT_01="-O2"
# CONFIGURACIN 01
MCTDH_FFLAGS_0PT_01="-O3 -fno-inline -ffast-math -
    fforce-addr -march=native -Ofast -msse4 -mavx2
    -ftree-vectorize"
# CONFIGURACIN 02
MCTDH_FFLAGS_0PT_01="-O3 -fno-inline -ffast-math -
    fforce-addr -march=native -Ofast -msse4 -mavx2
    -ftree-vectorize"
# CONFIGURACIN 03
MCTDH_FFLAGS_0PT_01="-O3 -fno-inline -fforce-addr -
    march=native -msse4 -mavx2 -ftree-vectorize"
```

Los resultados obtenidos se muestran en la figura 24, los cuales fueron adimensionalizados con el tiempo máximo de simulación. Entonces, podemos notar que todas las configuraciones demandaron más del $90\,\%$ del tiempo máximo de simulación mientras que en referencias a cache las configuraciones original y 01 permitieron reducir estas referencias a cache perdidas entre un 40-$50\,\%$. Las conclusiones que se pueden sacar de estos resultados es que no hay diferencias significativas de performance al agregar flags de optimización a los compiladores, esto podría deberse a que el paquete de MCTDH, además de los flags comunes mencionados en II-D7, cuenta con flags de compilacíon que optimizan notablemente el código, es decir, flags que chequean las unidades vectoriales disponibles y generan código específico para la arquitectura del recurso, esto permite que la compilación en su versión original ya esté optimizado, por lo menos en cuanto a flags de compilación. Sin embargo, es necesario notar que los flags que no aparecen en el paquete de MCTDH pero que, a criterio personal, deberían estar siempre son **-O3** (que a diferencia de -O1 y -O2, este flag de optimización prende la vectorización del algoritmo, entre otras propiedades de optimización interesantes) y **-march=native** (que permite generar un código binario haciendo uso de unidades específicas según la arquitectura donde se está compilando el programa).

II-D8. ¿Y si usamos procesos MPI en MCTDH?¿Qué tan rápido puede ser respecto a usar POSIX threads?

Teniendo en cuenta el artículo *Nagel, W.E., Kroner, D. and Resch, M., 2008* en donde se muestra que podrían obtenerse mejoras significativas de performance al utilizar paralelización vía OpenMPI, entonces, se decidió ir un paso más allá y lograr una compilación con MPI (Message Passing Interface). Los

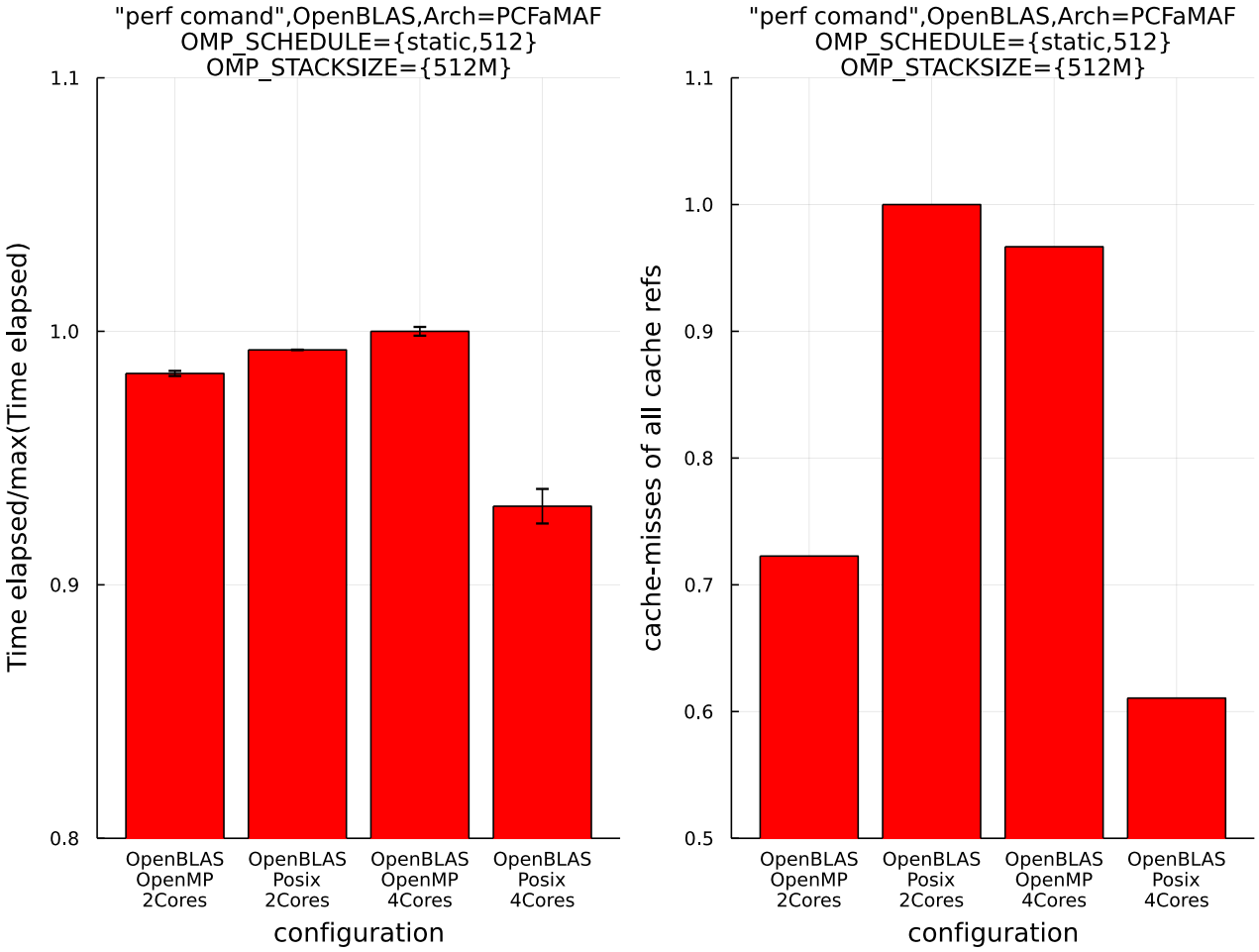

Fig. 23. Comparación de Performance paralelizando MCTDH con POSIX threads y con OpenMP, para distintos números de threads

resultados obtenidos fueron los que se muestran en la figura 25, donde se comparan los resultados de paralelización vía OpenMPI vs vía POSIX threads (con flags de optimización originales vs mejoradas). Entonces, se puede observar una reducción de aproximadamente un 30 % en el tiempo máximo de simulación ($t_{mx} = 1338{,}65$ [s]) utilizando MPI, lo cual evidencia efectivamente que hay mejorar algorítmicas dentro del paquete MCTDH que permiten un mejor rendimiento de las simulaciones al usar MPI.

Cabe mencionar que para correr con Posix threads hicimos uso de keywords y comandos específicos mostrados en 12, mientras que para correr con OpenMPI hicimos uso de aquellas keywords y comandos mostrados en 12. Además, al igual que comentamos para el caso de OpenMP, fue necesario compilar el paquete MCTDH de una forma específica para permitir el uso de procesos MPI.

Listing 11: Program: Keywords y comandos específicos para corridas de MCTDH con POSIX

```
RUN-SECTION
     usepthreads = 4,summf2 ,no-funkphi
     ptiming # Check ptiming file to control overload
          , and include
     rlxunit = au
     tfinal = 1 tout = 0.001 tpsi = 0.001
end-run-section

script_run.sh
     mctdh86 -mnd -w -p V_L 0.9,au -p lambda 0.2
          input_file.inp
```

Listing 12: Program: Keywords y comandos específicos para corridas de MCTDH con OpenMPI

```
RUN-SECTION
     usempi , no-dsyev
     ptiming # Check ptiming file to control overload
          , and include
     rlxunit = au
     tfinal = 1 tout = 0.001 tpsi = 0.001
end-run-section

script_run.sh
     mpirun -np 4 mctdh86.mpi-gfortran -mnd -w -p V_L
          0.9,au -p lambda 0.2 input_file.inp
```

II-D9. Un paso más... Incrementando paralelismo vía OpenMPI, POSIX y OpenBLAS

Bajo el mismo enfoque explicado en la sección II-D8 se llevó a cabo un análisis de las simulaciones aumentando el número de tareas vía OpenMPI y aumentando el número de threads de cada una de ellas utilizando POSIX threads, para ello fue necesario correr los programas en el servidor *Jupiterace*. Por otro lado, teniendo en cuenta que similar a las implementaciones con OpenMP, es necesario configurar el schedule de las corridas en OpenMPI, es decir, cuántas tareas (o procesos MPI) ejecutará el problema y por cada una de estas cuántos threads tendrá asociados, estos parámetros deben configurarse de forma específica según el administrador de colas del cluster, en nuestro caso el servidor utilizado (Jupiterace) implementa SLURM y las lineas dentro del script de corrida se muestran en 13, donde las variables index1 (número de tareas o procesos MPI) e index2 (número de POSIX threads) son índices que se barren desde 1 a 28. Es in-

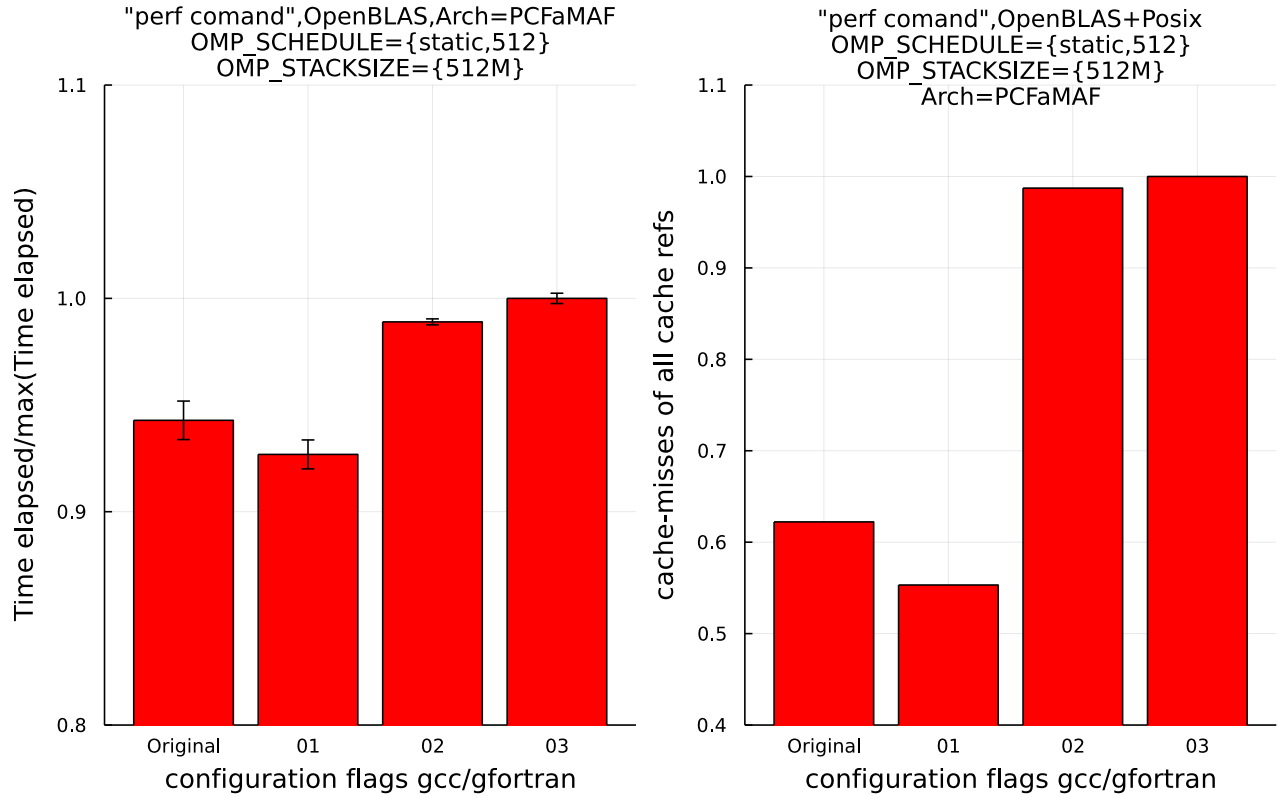

Fig. 24. Análisis de performance para distintas configuraciones de flags (gfotran/gcc)

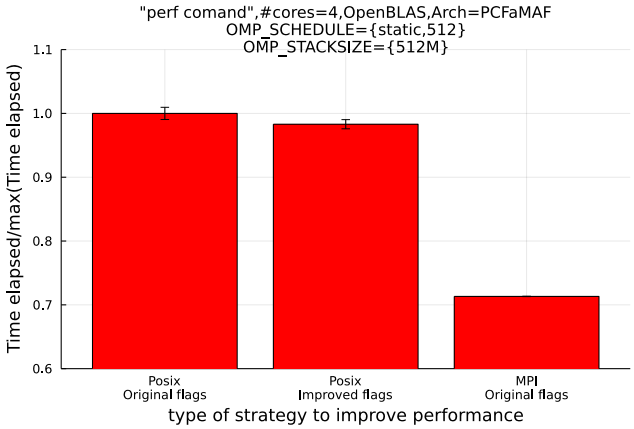

Fig. 25. Comparación de Performance entre Posix vs MPI

teresante notar que, bajo esta configuración los procesos MPI se mapean directamente con threads lógicos de procesamiento y los treads POSIX son abstracciones dentro de cada thread lógico dado por las tareas MPI, entonces, el paralelismo real se produce vía tareas MPI, para mayor información de este aspecto se puede observar la figura 31 donde hemos capturado la salida del comando *htop* en el servidor Jupiterace, para una ejecución con 2 procesos MPI y 14 Posix threads, donde vemos que la paralelización *real* es a través de los procesos de MPI ocupándose 2 cores reales del recurso computacional y sobre cada core tendremos una paralelización *abstracta* de 14 threads.

Por otro lado, un aspecto que requiere más atención es que para que no ocurran errores, es decir, para que SLURM no nos diga algo como "*no es posible asignar la cantidad de recursos solicitada, numero de slots, etc*" fue necesario cumplir con la condición de que $index_1 \cdot index_2 \leq 28$. Esto podría estar relacionado a que al momento de correr el comando **lscpu** en el servidor JupiterAce nos dice que **Thread(s) per core: 1** lo cual nos estaría diciendo que el multithreading no esta activado o con algún aspecto de particular relevancia en el uso de procesos MPI que no estamos considerando.

Listing 13: Program: Metadatos para ejecutar en SLURM y variables de entorno para ejecutar OpenMP, POSIX y OpenMPI

```
# +++ Metadatos para ejecutar con slurm
# +++ Nombre para identificar el trabajo.
#     Por defecto es el nombre del script
#SBATCH --job-name=MPI${index1}_POSIX${index2}
# +++ Solicita N cores, por defecto se asignan
#     consecutivamente dentro de un nodo. Si N
#     excede la cantidad de cores contin a en
#     otro nodo
#SBATCH --ntasks=${index1}
#SBATCH --ntasks-per-node=${index1}
# +++ Especifico en cuantos hilos correra cada
#     uno de los procesos anteriores
#SBATCH --cpus-per-task=${index2}
# +++ Especifico que no quiero a nadie m s en
#     el nodo cuando corra esta tarea
#SBATCH --exclusive
# +++ Env a un correo cuando el trabajo finaliza
#     correctamente o por alg n error
#SBATCH --mail-type=END
#SBATCH --mail-user=martinmendez@mi.unc.edu.ar
# +++ La linea siguiente es ignorada por Slurm
#     porque empieza con ##
##SBATCH --mem-per-cpu=4G # cantidad de memoria x
    core

# ++++++++++++++++++++++++++++++++++++++++++++
# set threads to use in OpenBLAS with OpenMP
export OMP_NUM_THREADS=${index2}

# schedule of OpenBLAS with OpenMP
export OMP_SCHEDULE="static,512"
export OMP_STACKSIZE="512M"

# threads distribution configuration to OpenBLAS
export OPENBLAS_NUM_THREADS=1
```

```
export USE_THREAD=0
export USE_LOCKING=1
```

Los resultados se muestran en la figura 26, la cual corresponde a un mapa de colores variando el número de POSIX threads y MPI processes, asignándole a cada par de estos valores un color correspondiente al tiempo de CPU demandado por la simulación normalizado con el tiempo máximo medido (en la figura aparecen algunos sitios sin color, que corresponden a simulaciones abortadas por algún problema particular). Como podemos ver los peores tiempos se obtuvieron utilizando un único proceso MPI, independientemente de los POSIX threads empleados, esto claramente es lo esperable pues como mencionamos el paralelismo real ocurre en la variable MPI y a medida que aumentamos el paralelismo (abstracción) vía POSIX threads simplemente estamos dividiendo aún más un único core de procesamiento ralentizando aún más la simulación. Por otro lado, el menor tiempo de CPU se obtuvo para 28 procesos MPI y un único POSIX thread, sin embargo, al analizar un poco más cuáles fueron los mejores tiempos (tiempos cercanos a esta configuración mencionada) obtuvimos los resultados mostrados en la figura 27, donde podemos ver que si bien el mejor tiempo se obtiene para 28 procesos MPI y 1 POSIX thread, un rendimiento similar (menos de un $1\,\%$ de diferencia) se obtiene con 4 procesos MPI y 7 POSIX threads, además, esta última configuración es preferible para realizar un uso más eficiente de los recursos computacionales pues, para el primer caso se requiere reservar (de forma exclusiva) 28 cores en servidor imposibilitando su uso por otro usuario, mientras que para la segunda configuración simplemente se requieren 4 cores, dejando 24 cores libres para que otro usuario los aproveche, mientras tanto, no perdemos eficiencia apreciable.

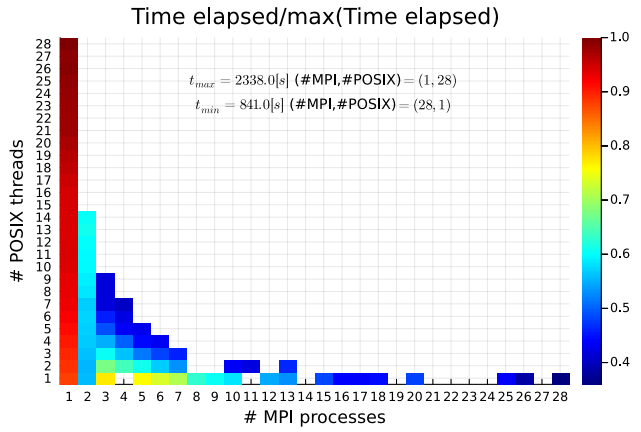

Fig. 26. Escalamiento de performance aumentando número de POSIX threads y número de procesos MPI

III. Comparación final: ¿Qué tanto jugo le sacamos al programa?

Finalmente, se comparó la performance del problema en su estado original, sin ningún tipo optimización, con la mejor configuración de optimizaciones encontradas, de tal forma de verificar cuánto rendimiento se ha podido extraer de las herramientas aprendidas en el curso. Para ello, se eligió simular un problema más demandante eligiendo un parámetro perturbativo de $\lambda = 1{,}0$ (ver figura 11), de esta forma el tiempo de simulación sería del orden de los días.

Los resultados se muestran en la figura 28, donde los mismos se han normalizado respecto al tiempo máximo obtenido

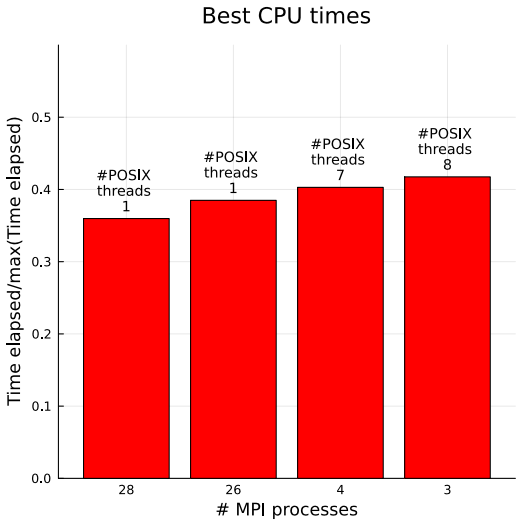

Fig. 27. Mejores 4 tiempos de CPU obtenidos para simulaciones vía POSIX threads y MPI processes

(correspondiente a la versión original con un tiempo máximo de aproximadamente 39 horas). Podemos notar que en la configuración con paralelismo vía POSIX threads, OpenBLAS y flags de optimización del compilador logramos disminuir el tiempo de CPU en aproximadamente un $50[\,\%]$ y para la configuración con paralelismo vía POSIX threads, OpenBLAS, MPI processes y flags de optimización del compilador logramos disminuir el tiempo de CPU en aproximadamente un $70[\,\%]$. Los resultados fueron muy prometedores, y se puede observar que la optimización gruesa se llevo a cabo con la utilización de OpenBLAS, luego optimizando detalles más finos se logró un rendimiento aún mejor.

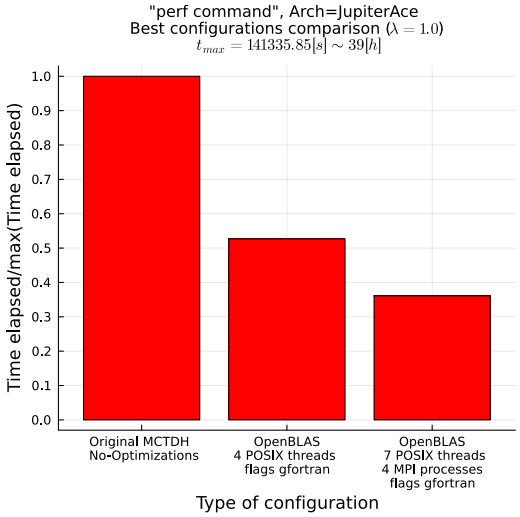

Fig. 28. Comparación final de performance entre configuración original de MCTDH vs mejor configuración con optimizaciones

IV. Anexo I: Tratando de utilizar MKL para optimizar operaciones algebraicas

Teniendo en cuenta que en el cluster Bandurria cuenta con la API MKL (software tipo free software) y al indagar sobre esta herramienta y su potencialidad frente a OpenBLAS para optimizar tanto BLAS como LAPACK se realizó un pequeño análisis de su performance para un problema reducido. En particular, se llevó a cabo el desarrollo de un algoritmo (código escrito en FORTRAN) que permita

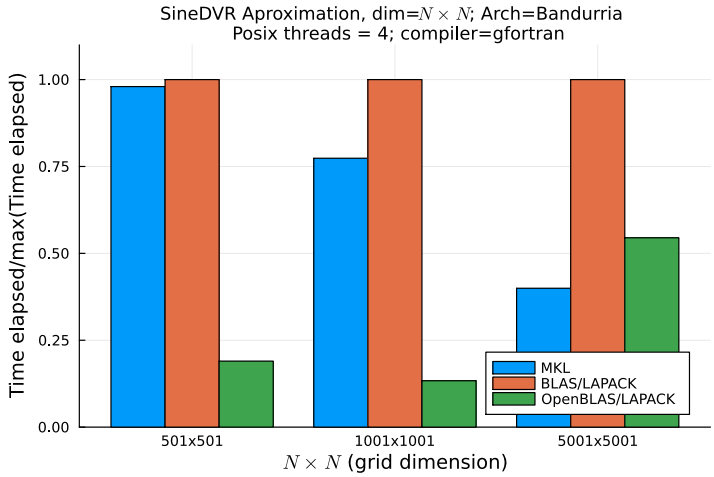

Fig. 29. Comparación de Performance entre BLAS+LAPACK, MKL y OpenBLAS+LAPACK

calcular la autoenergías y autofunciones de un electrón en un doble punto cuántico, utilizando aproximación tipo Sine-DVR (Discrete Variable Representation), para ello el algoritmo deberá encontrar autovalores y autovectores de la matriz Hamiltoniana del sistema utilizando subrutinas especificas de BLAS y LAPACK para la multiplicación y diagonalización de operadores. Entonces, para comparar el rendimiento de MKL, medido en tiempo de CPU, se realizaron simulaciones con distintos tamaños de grillas Sine-DVR y se calcularon autoenergías y autovalores con tres métodos, el primero es usando BLAS+LAPACK, el segundo es usando MKL y el tercero es usando OpenBLAS+LAPACK. Los resultados se muestran en la figura 29, donde para cada dimensión de grilla los resultados se han normalizado respecto al tiempo máximo requerido ($t_{mx} = \{14{,}35, 102{,}98, 3443{,}41\}$ [s] respectivamente).

Los resultados muestran que si usamos BLAS+LAPACK se obtienen los peores rendimientos y, lo interesante es que para dimensiones pequeñas de grillas SineDVR el rendimiento de OpenBLAS+LAPACK frente a MKL es muy superior (reduciendo el tiempo de CPU en hasta un $80\,\%$), sin embargo, al aumentar considerablemente la dimensión de la grilla el rendimiento de MKL aumenta considerablemente (reduciendo el tiempo de CPU en hasta un $50\,\%$), mejorando incluso el comportamiento para OpenBLAS+LAPACK. Esto sugeriría que las implementaciones de OpenBLAS+LAPACK tienen subrutinas optimizadas para dimensiones de grillas pequeñas, mientras que MKL tiene más optimizaciones para grillas grandes.

Este análisis básico de rendimiento es importante porque nos abre las puertas a considerar simulaciones de MCTDH utilizando MKL, sin embargo, al ser un software bajo licencia tiene la gran limitación de que dependemos de dicho acceso al software, no pudiendo explotar todos los recursos computacionales. Se ha intentado solucionar este aspecto creando entornos específicos de software utilizando CONDA, sin embargo, no se han obtenido resultados exitosos por el momento y solamente hemos podido realizar las simulaciones con versiones antiguas de MKL y gfortran (las cuales se encuentran instaladas en el cluster Bandurria), pero obviamente para obtener los tiempos de CPU más óptimos lo primero que deberíamos hacer es mantener actualizados los software.

V. Códigos

Repositorio de GitHub

- https://github.com/mendzmartin/mctdh_cp2021

Reporte de cómo usar CONDA

- https://github.com/mendzmartin/mctdh_cp2021/blob/master/double_quantum_dot_model/qdot_02/energies_vs_lambda/study_of_performance_04/UsingConda.ipynb

Reporte de cómo usar OpenBLAS

- https://github.com/mendzmartin/mctdh_cp2021/blob/master/double_quantum_dot_model/qdot_02/energies_vs_lambda/study_of_performance_03/running_whit_openblas/README.ipynb

Reporte de resultados

- https://github.com/mendzmartin/mctdh_cp2021/blob/master/double_quantum_dot_model/qdot_02/energies_vs_lambda/study_of_performance_03/running_comparison_performance/README.ipynb
- https://github.com/mendzmartin/mctdh_cp2021/blob/master/double_quantum_dot_model/qdot_02/energies_vs_lambda/study_of_performance_04/results.ipynb

Información de Hardware

- https://github.com/mendzmartin/mctdh_cp2021/blob/master/double_quantum_dot_model/qdot_02/energies_vs_lambda/study_of_performance_01/info_hardware.md

Reporte de problema librería pthread MCTDH

- https://github.com/mendzmartin/mctdh_cp2021/blob/master/MCTDH_Reports/libpthread_problem.ipynb

Referencias

[1] Daniel V. Schroeder. "The variational-relaxation algorithm for finding quantum bound states". En: *American Journal of Physics* 85.9 (sep. de 2017). Publisher: American Association of Physics Teachers, págs. 698-704. ISSN: 0002-9505. DOI: 10.1119/1.4997165. URL: https://aapt.scitation.org/doi/10.1119/1.4997165 (visitado 21-12-2021).

[2] Steven Chapra y Raymond Canale. *Numerical Methods for Engineers*. English. 7th edition. New York, NY: McGraw Hill, ene. de 2014. ISBN: 978-0-07-339792-4.

[3] Hans-Dieter Meyer, Fabien Gatti y Graham A. Worth. *Multidimensional Quantum Dynamics: MCTDH Theory and Applications*. en. Google-Books-ID: LMVLvheAeEAC. John Wiley & Sons, mayo de 2009. ISBN: 978-3-527-32018-9. URL: https://books.google.com.ar/books?id=LMVLvheAeEAC.

[4] R. Kosloff y H. Tal-Ezer. "A direct relaxation method for calculating eigenfunctions and eigenvalues of the schrödinger equation on a grid". en. En: *Chemical Physics Letters* 127.3 (jun. de 1986), págs. 223-230. ISSN: 0009-2614. DOI: 10.1016/0009-2614(86)80262-7. URL: https://www.sciencedirect.com/science/article/pii/0009261486802627 (visitado 20-12-2021).

[5] Andrew B. Lockhart, Alexandria Skinner, William Newman, Daniel B. Steinwachs y Shawn A. Hilbert. "An experimental demonstration of avoided crossings with masses on springs". En: *American Journal of Physics* 86.7 (jul. de 2018). Publisher: American Association of Physics Teachers, págs. 526-530. ISSN: 0002-9505. DOI: 10.1119/1.5036752. URL: https://aapt.scitation.org/doi/10.1119/1.5036752 (visitado 18-12-2021).

```
[mmendez@bandurria ~]$ qhost
HOSTNAME                ARCH         NCPU NSOC NCOR NTHR  LOAD  MEMTOT  MEMUSE  SWAPTO  SWAPUS
----------------------------------------------------------------------------------------------
global                  -               -    -    -    -     -       -       -       -       -
compute-0-0             lx-amd64        4    1    4    4     -    5.7G       - 1000.0M       -
compute-0-10            lx-amd64        6    1    6    6  0.01    5.7G  195.7M 1000.0M  223.0M
compute-0-11            lx-amd64       12    1    6   12  0.01    5.7G  241.5M 1024.0M  209.7M
compute-0-12            lx-amd64        6    1    6    6  0.01    7.6G  207.9M 1024.0M  210.0M
compute-0-13            lx-amd64       12    1    6   12  0.01    7.6G  288.6M 1000.0M  409.2M
compute-0-14            lx-amd64        8    1    4    8  0.02   19.3G  372.4M 1024.0M     0.0
compute-0-16            lx-amd64       20    1   10   20  0.01   15.2G  391.1M 1024.0M  464.8M
compute-0-17            lx-amd64       20    1   10   20  1.02   15.2G  453.5M 1024.0M  164.0M
compute-0-18            lx-amd64       16    1    8   16  1.01   31.1G  660.9M   11.2G     0.0
compute-0-19            lx-amd64       12    1    6   12  1.01   15.4G  501.7M    7.8G     0.0
compute-0-2             lx-amd64        4    1    4    4     -    5.7G       - 1000.0M       -
compute-0-20            lx-amd64        8    1    8    8  0.01   15.4G  349.9M    7.8G  232.2M
compute-0-21            lx-amd64       16    1    8   16  1.02   15.4G  530.2M 1024.0M     0.0
compute-0-22            lx-amd64       16    1    8   16  1.01   62.6G  545.1M 1024.0M  209.0M
compute-0-23            lx-amd64       20    1   10   20  4.01   31.1G  660.6M 1024.0M    7.2M
compute-0-24            lx-amd64       16    1    8   16  2.01   38.9G   13.9G 1024.0M  256.2M
compute-0-3             lx-amd64        4    1    4    4     -    5.7G       -  996.2M       -
compute-0-4             lx-amd64        8    1    4    8     -    5.7G       - 1000.0M       -
compute-0-5             lx-amd64        4    1    4    4  0.01    5.7G  192.7M 1024.0M  238.4M
compute-0-7             lx-amd64        4    1    4    4  0.01    5.7G  190.0M  996.2M  303.4M
compute-0-8             lx-amd64        8    2    8    8  0.01   23.3G  453.7M 1000.0M     0.0
compute-0-9             lx-amd64        4    1    4    4  0.01    5.7G  191.3M 1000.0M  313.4M
gpu-compute-0-0         lx-amd64        8    1    4    8  1.01    5.7G  207.9M 1000.0M   77.4M
gpu-compute-0-1         lx-amd64        4    1    4    4     -    5.7G       - 1000.0M       -
gpu-compute-0-2         lx-amd64        4    1    4    4  0.01    5.7G  184.5M 1000.0M  259.6M
```

Fig. 30. Salida del comando **qhost -q** en el cluster Bandurria

[6] K. R. Brown, C. Ospelkaus, Y. Colombe, A. C. Wilson, D. Leibfried y D. J. Wineland. *Coupled quantized mechanical oscillators | Nature*. URL: https://www.nature.com/articles/nature09721 (visitado 18-12-2021).

[7] Federico M. Pont, Axel Molle, Essam R. Berikaa, Sascha Bubeck y Annika Bande. *Predicting the performance of the inter-Coulombic electron capture from single-electron quantities - IOPscience*. English. URL: https://iopscience.iop.org/article/10.1088/1361-648X/ab41a9 (visitado 18-12-2021).

[8] *POSIX Threads for Windows – REFERENCE*. URL: http://sourceware.org/pthreads-win32/manual/index.html (visitado 10-08-2021).

[9] *POSIX Threads, Hilos o Hebras*. URL: http://profesores.elo.utfsm.cl/~agv/elo330/2s08/lectures/POSIX_Threads.html (visitado 10-08-2021).

[10] *POSIX Threads: Sincronización de hilos en POSIX: Mutex*. URL: http://profesores.elo.utfsm.cl/~agv/elo330/2s08/lectures/POSIX_Threads_Synchronization.html (visitado 10-08-2021).

[11] *Bash Reference Manual - Table of Contents*. URL: http://profesores.elo.utfsm.cl/~agv/doc/shell/bashref_toc.html (visitado 10-08-2021).

[12] *GNU Pth - The GNU Portable Threads*. URL: http://www.gnu.org/software/pth/pth-manual.html (visitado 10-08-2021).

[13] Wolfgang E. Nagel, ed. *High Performance Computing in Science and Engineering '09: Transactions of the High Performance Computing Center, Stuttgart (HLRS) 2009*. en. Berlin Heidelberg: Springer-Verlag, 2010. ISBN: 978-3-642-04664-3. DOI: 10.1007/978-3-642-04665-0. URL: https://www.springer.com/gp/book/9783642046643 (visitado 10-08-2021).

[14] Fabian Langkabel y Annika Bande. "Three-electron dynamics of the interparticle Coulombic decay with two-dimensional continuum confinement". En: *The Journal of Chemical Physics* 154.5 (feb. de 2021). Publisher: American Institute of Physics, pág. 054111. ISSN: 0021-9606. DOI: 10.1063/5.0037806. URL: https://aip.scitation.org/doi/10.1063/5.0037806 (visitado 23-12-2021).

[15] Laércio Lima Pilla. "Basics of Vectorization for Fortran Applications". en. En: (), pág. 13.

[16] *Optimize Options - Using and Porting GNU Fortran*. URL: https://gcc.gnu.org/onlinedocs/gcc-3.4.6/g77/Optimize-Options.html (visitado 02-11-2022).

[17] *Math Kernel Library*. en. Page Version ID: 1114693178. Oct. de 2022. URL: https://en.wikipedia.org/w/index.php?title=Math_Kernel_Library&oldid=1114693178 (visitado 18-11-2022).

[18] *Conda (package manager)*. en. Page Version ID: 1116828610. Oct. de 2022. URL: https://en.wikipedia.org/w/index.php?title=Conda_(package_manager)&oldid=1116828610 (visitado 18-11-2022).

[19] Michael Brill, Oriol Vendrell y Hans-Dieter Meyer. "Distributed Memory Parallelization of the Multi-Configuration Time-Dependent Hartree Method". En: ene. de 2010, págs. 147-163. ISBN: 978-3-642-04664-3. DOI: 10.1007/978-3-642-04665-0_11.

[20] *Open MPI: Open Source High Performance Computing*. URL: https://www.open-mpi.org/ (visitado 28-11-2022).

[21] Richard Friedman. *Reference Guides*. en-GB. URL: https://www.openmp.org/resources/refguides/ (visitado 28-11-2022).

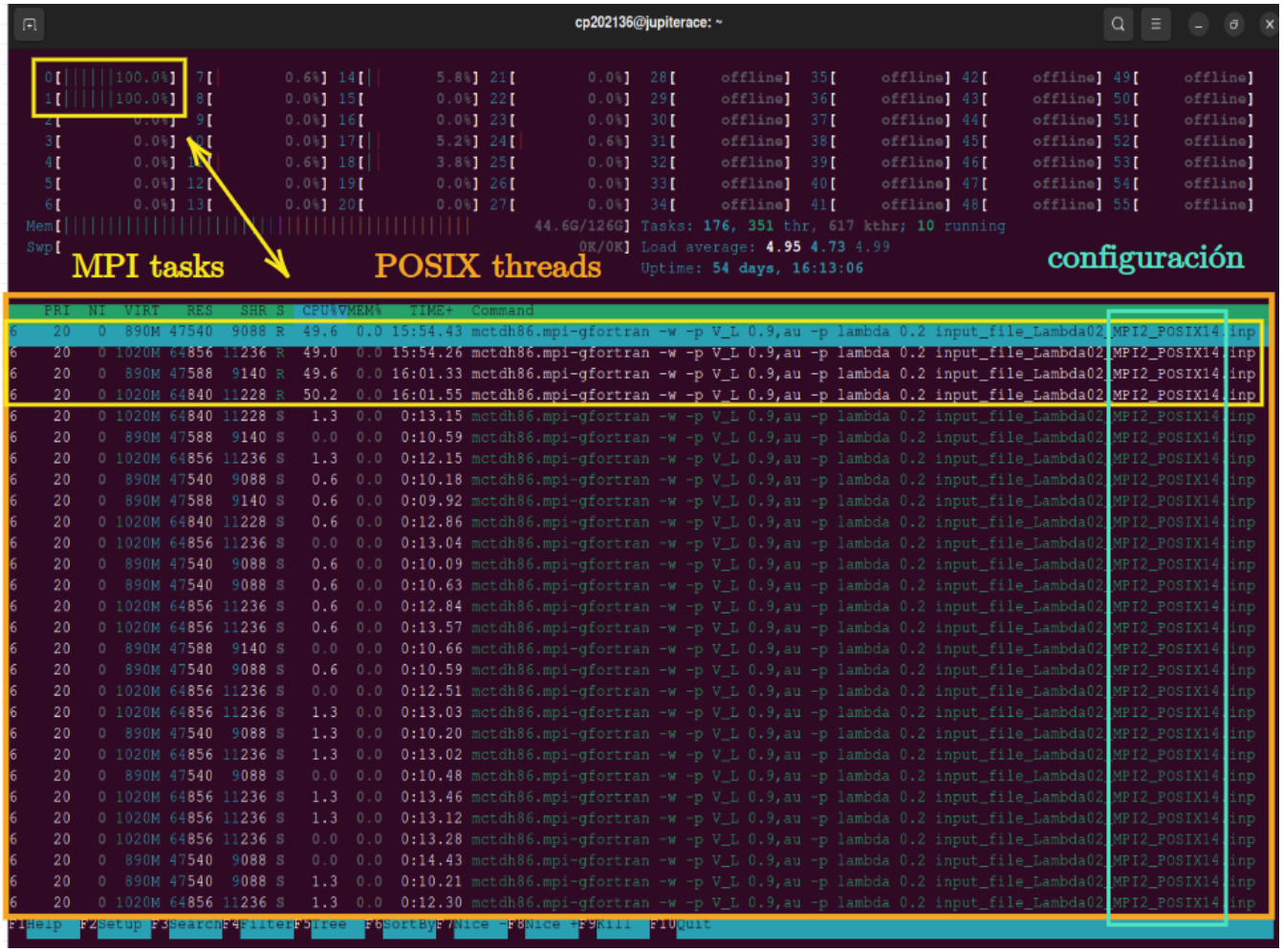

Fig. 31. Salida del comando **htop** en servidor Jupiterace