Nociones básicas de Computación de Altas Prestaciones

Rafael Mayo García – CIEMAT

Introducción

Los computadores han servido a lo largo de las últimas décadas para hacer simulaciones numéricas de fenómenos físicos, químicos, socioeconómicos, de tratamiento de datos, etc. Estas simulaciones sirven para realizar modelos, prever comportamientos, encontrar soluciones… por lo que han cambiado el modelo en el que se corroboraban teoría y experimentos. Actualmente y dada la interdisciplinaridad de la ciencia y la tecnología, es difícil encontrar investigaciones en las cuales no sea acuda en alguna fase de las mismas a realizar simulaciones con ordenador.

Diseño de nuevas instalaciones, predicción de proteínas, confirmación experimental de modelos complejos, meteorología y cambio climático, control de calidad, comportamiento de los mercados, defensa, etc. son sólo una breve mención de los resultados que son posibles gracias a la computación de altas prestaciones. Su impacto, además, se centra tanto en el sector público como en el privado, en donde ya nadie duda del famoso aforismo "*To compete, you must compute*" (Para competir, debes computar), o en otras palabras, computar/simular es una ventaja competitiva.

También es importante reseñar que actualmente no sólo los supercomputadores o los ordenadores personales, sino también los dispositivos móviles, tales como tabletas o teléfonos móviles, incorporan tecnología de supercomputación a partir de la integración de más de un procesador (*core*) en sus chips.

Gracias a ello y a las herramientas de software -sistema operativo y otras como compiladores, bibliotecas, servidores web, bases de datos, etc.- que gestionan la arquitectura de estos equipos, es posible realizar cálculos distribuidos o en paralelo, es decir, es posible realizar distintos cálculos a la vez usando los distintos procesadores con los que se cuenta.

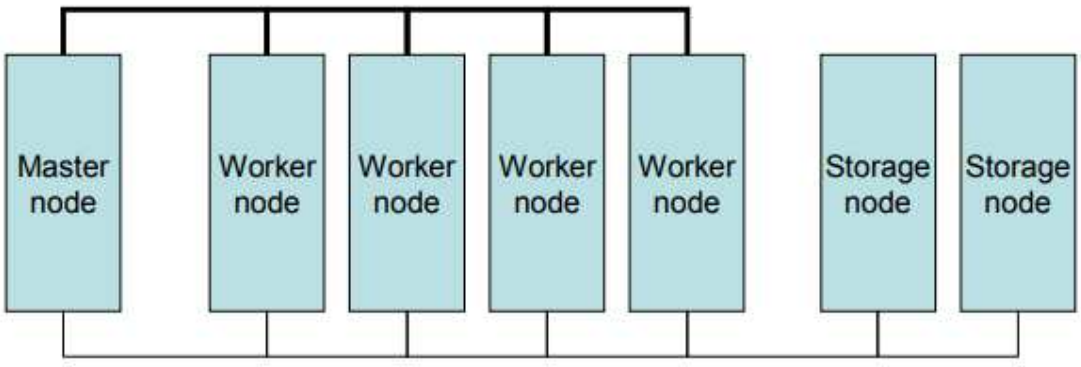


Figura 1. Arquitectura básica y simplificada de un clúster de computación

Así, la Computación de Altas Prestaciones se divide en Computación de Alta Productividad (*High Throughput Computing* o HTC por sus siglas en inglés) y

Computación de Alto Rendimiento (*High Performance Computing* o HPC por sus siglas en inglés). Los supercomputadores realizan estos cálculos bajo una arquitectura que básicamente se compone de unos nodos compuestos por distintos *cores* y que se emplean bien para acceder al clúster (nodo de acceso) o bien para realizar las simulaciones numéricas en sí (nodos de cálculo), interconectados entre sí mediante redes de baja latencia (Omni-Path, Infiniband, Ethernet) y conectados a su vez a nodos de almacenamiento. Un esquema de esta arquitectura que se desarrollará más adelante se puede ver en la Fig. 1.

Computación de Alta Productividad

El objetivo de la Computación de Alta Productividad es aumentar el número de ejecuciones por unidad de tiempo. Por ello, su rendimiento se mide en número de trabajos ejecutados por segundo, pues se busca que haya una gran cantidad de ejecuciones (independientes) a realizar. Así, se distribuyen tareas dentro del clúster para conseguir que *N* ejecuciones se hagan *M* veces más rápido (véase Fig. 2).

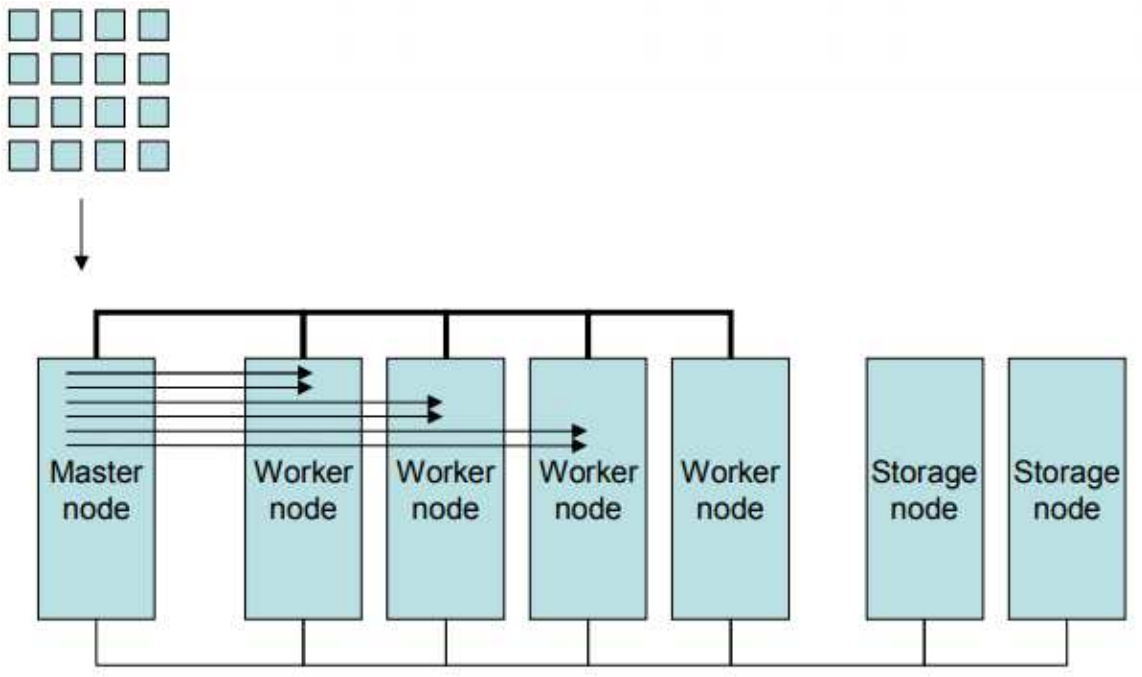


Figura 2. Esquema de una distribución de tareas HTC.

Los códigos de más aplicación en HTC son los Monte Carlo, Barrido de parámetros, Algoritmos genéticos, etc. Es decir, problemas en los que se hacen muchos cálculos independientes unos de otros. Los campos científicos que explotan este tipo de cálculos son muy numerosos: Radiofísica, Economía, Finanzas, Medioambiente, Física de Altas

Energías, Ingeniería, Estadística, Física de Plasmas, Química, Técnicas de optimización, etc.

Este tipo de cálculos se pueden ejecutar de tres maneras diferentes dependiendo de: la duración de los trabajos independientes o del esquema de cálculo final. En los modelos síncrono y asíncorono existe un trabajo (*job*) de pre-proceso y otro de post-proceso que son los encargados de realizar la distribución de los cálculos (etiquetados como trabajos *0, 1, 2… n,* siendo *n=N-1*) en un principio y de realizar el cálculo final con el resultado de todos ellos. El cálculo será síncrono si todos los trabajos independientes *0,1…n* terminan a la vez y asíncrono si no lo hacen (ver Figura 3).

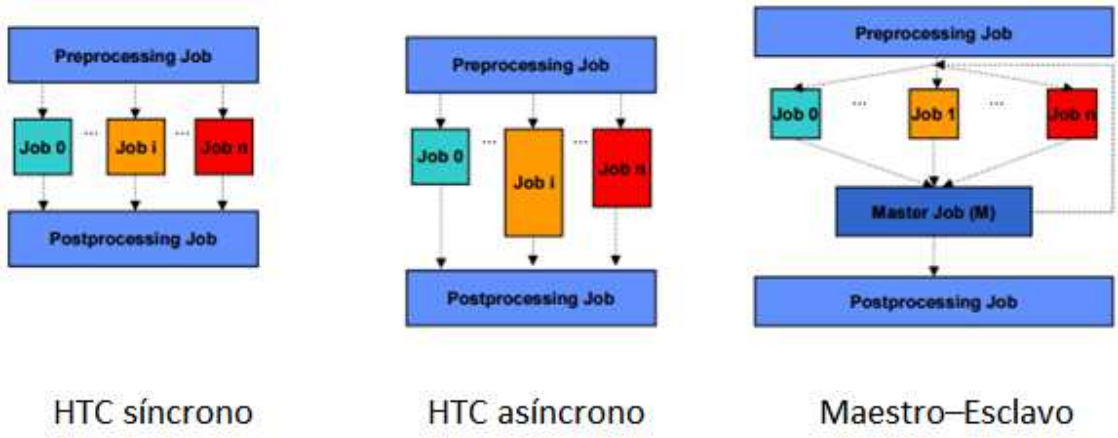


Figura 3. Ejemplos de computaciones HTC.

Por su parte, en el esquema de cálculo maestro-esclavo sigue existiendo los trabajos de pre-proceso y post-proceso, pero a ellos se añade un trabajo maestro que controla la ejecución de todos los trabajos independientes -muy común cuando se realizan tareas en bucle en las cuales un(os) cálculo(s) se repite(n) varias veces hasta que se alcanza una condición que indica cuándo terminar el bucle.

Un ejemplo muy sencillo de un cálculo HTC puede ser obtener la media aritmética de los resultados de varias operaciones (Fig. 4). Estos cálculos se pueden hacer todos a la vez independientemente unos de otros y que sea el trabajo de post-proceso el que calcule la media de las sumas independientes anteriores. Es decir, los trabajos independientes hacen las sumas que les indica el trabajo de pre-pre-proceso (40, 41, 39… en el ejemplo) y les van comunicando su resultado parcial al trabajo de post-proceso que será el encargado de calcular la media aritmética de las 7 sumas anteriores para obtener el valor final del cálculo de 40.

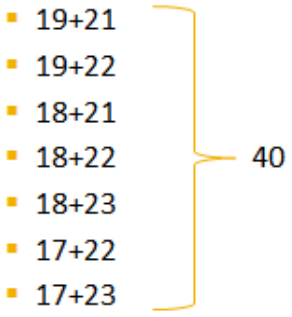


Figura 4. Ejemplo de HTC.

Como ejemplo científico del uso de la Computación de Alta Productividad se puede mencionar a la Física de Altas Energías, en donde la simulación de las colisiones originadas dentro del experimento LHC del CERN resultó vital para la medición del bosón de Higgs.

Computación de Alto Rendimiento

La Computación de Alto Rendimiento (o de Alto Desempeño como se suele decir en el ámbito latinoamericano) está más asociada a lo que solemos llamar supercomputación o HPC por sus siglas en inglés, aunque eso no implica que con supercomputadores no se realicen cálculos HTC.

Su objetivo es reducir el tiempo de ejecución de una única aplicación paralela, por lo que su rendimiento se mide en número de operaciones en coma flotante por segundo (flops). La representación de coma flotante (en inglés *floating point*)[1] es una forma de notación científica usada en los microprocesadores con la cual se pueden representar números racionales extremadamente grandes o pequeños de una manera muy eficiente y compacta en base 2 (sistema binario), y con la que se pueden realizar operaciones aritméticas. El lector puede averiguar más sobre esta notación en

En este tipo de computación, por tanto, se suele tener una única ejecución de gran complejidad, la cual se divide en partes interdependientes. De este modo, las tareas o cálculos interdependientes se distribuyen (o paralelizan) en los distintos nodos del clúster de tal forma que las dependencias entre ellos se comunican a través de la red de datos de baja latencia de la que disponga el clúster. En este caso, lo que se persigue es 1 ejecución se realice *M* veces más rápido (véase Fig. 5).

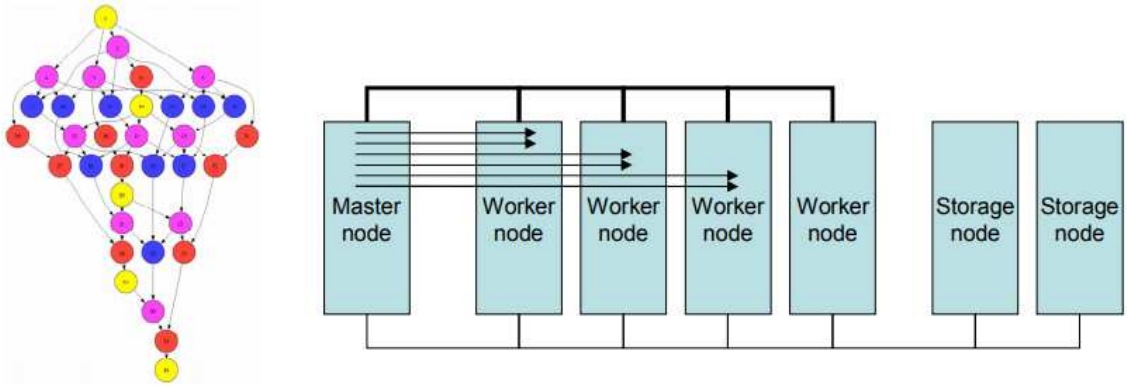


Figura 5. Esquema de una distribución de tareas HPC.

[1] https://www.uv.es/~diaz/mn/node10.html

Las áreas de aplicación de HPC son totalmente generalistas. Por ello, las podemos dividir en:

- Estudio de fenómenos a escala microscópica (dinámica de partículas, por ejemplo)
    - En ella, la resolución está limitada por la potencia de cálculo del computador
    - Cuantos más grados de libertad (puntos) se considere, mejor reflejo de la realidad se obtiene
- Estudio de fenómenos a escala macroscópica (por ejemplo, sistemas descritos por ecuaciones diferenciales fundamentales)
    - En este caso es la precisión la que está limitada por la potencia de cálculo del computador
    - Cuantos más puntos se consideren, más se acerca la solución discreta (aproximada mediante esos puntos) a la continua (la real en la cual, y por decirlo así, todos los puntos anteriores terminan formando una línea, un plano o un volumen continuo).

La Fig. 6 ilustra un ejemplo sencillo de lo que sería una ejecución con HPC. Como se puede ver, existen interdependencias entre los distintos trabajos que se van realizando de tal forma que el avance de la ejecución total depende que estas comunicaciones se vayan resolviendo de forma satisfactoria.

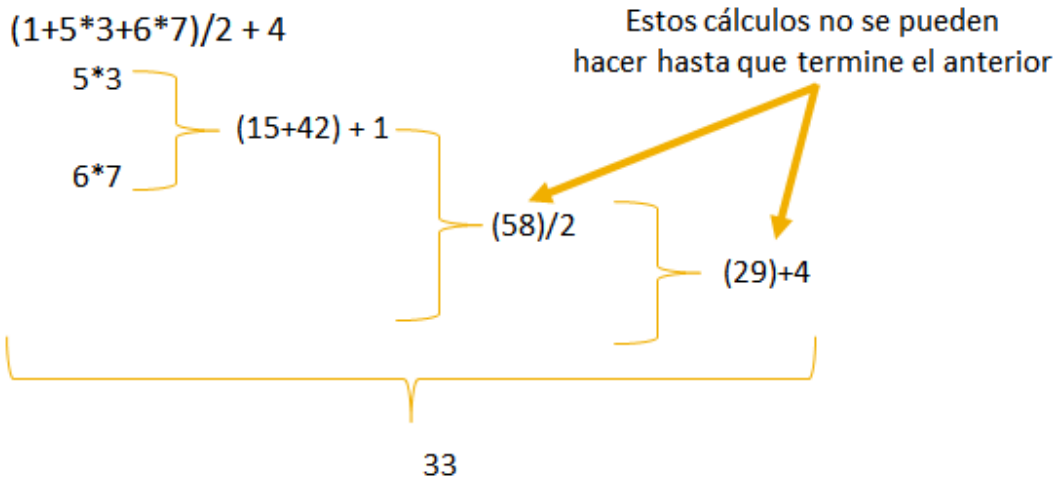


Figura 6. Ejemplo de HPC.

Ya hemos indicado anteriormente que uno de los casos más emblemáticos en los que se usa HPC es la simulación de fenómenos físicos descritos mediante ecuaciones diferenciales fundamentales (Ecuación no lineal de Schrödinger, Ecuaciones de Maxwell-Maxwell-Bloch, etc.) en los cuales esas derivadas parciales (u otras operaciones matemáticas como integrales) se discretizan mediante una representación por puntos que permite hacer operaciones sencillas en coma flotante al clúster de computación. Una representación de este proceso para derivadas parciales se puede encontrar en la Fig. 7 –

–las integrales se realizan mediante técnicas de cuadratura tales como Simpson, Lobatto o Gauss-Kronrod[2].

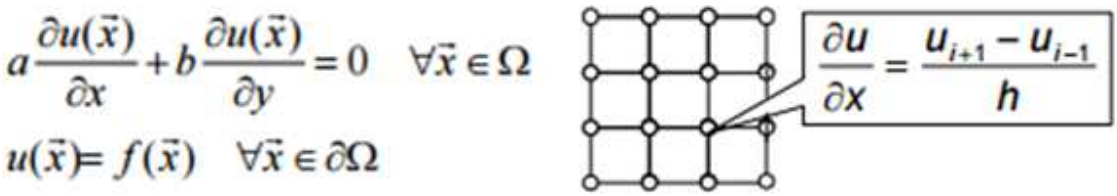


Figura 7. Representación de operaciones complejas mediante una discretización por puntos que permite al clúster realizar las primeras.

Igualmente, en la Fig. 8 se muestran algunos ejemplos de la necesidad de la potencia de cálculo en ámbitos científicos en los que el Nº de operaciones = Rendimiento * Tiempo.

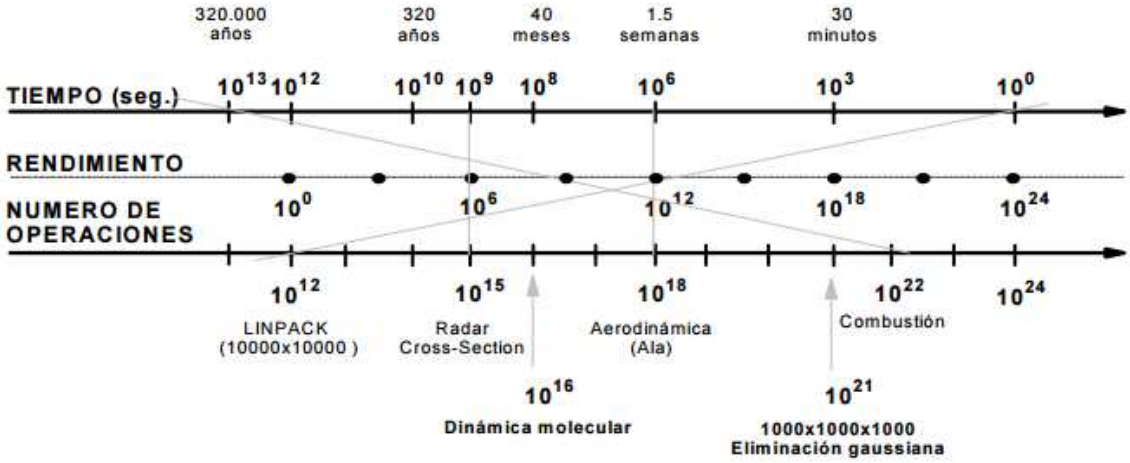


Figura 8. Potencia de cálculo necesaria en ámbitos científicos.

¿Pero qué significan realmente esas cantidades? Si se acude a la leyenda de los granos de trigo y del tablero de ajedrez[3], la situación es tal que en el primer escaque se colocaba un único grano de trigo el cual se iba doblando según se avanzaba al siguiente escaque y la suma total es la que debía entregarse, es decir, 1+2+4+… llegándose a que solamente en el último escaque debían entregarse 9.223.372.036.854.780.000 granos de trigo. Esta suma se puede también escribir como $2^0+2^1+2^2+\ldots+2^{63}$, la cual se puede calcular fácilmente mediante la fórmula siguiente:

[2] https://es.mathworks.com/content/dam/mathworks/mathworks-dot-com/moler/quad.pdf

[3] https://matematicascercanas.com/2014/03/10/la-leyenda-del-tablero-de-ajedrez-y-los-granos-de-trigo/

$$\sum_{i=0}^{63} 2^i = 2^{64}-1 = 18.446.744.073.709.551.615 \sim 18{,}5\cdot 10^{18}$$

Por curiosidad, esta suma que expresa la cantidad de granos de trigo que debían entregarse es aproximadamente $15{,}4\cdot 10^{12}$ Tm, o lo que es lo mismo, unas 22.000 veces la producción anual mundial de trigo.

Un ejemplo más actual y cotidiano puede ser la predicción meteorológica. Ésta se puede representar como una función de la longitud, latitud, altura y tiempo. Así:

- Para cada uno de estos puntos se debe calcular temperatura, presión, humedad y velocidad del viento (3 componentes). En total, 6 parámetros.
- Conocida la función clima(i,j,k,t), el simulador debe proporcionar el valor clima(i,j,k,t+Δt), es decir, el valor de la función transcurrido un intervalo de tiempo (Δt)
- Si queremos la predicción dentro de un minuto (Δt = 1 minuto) y dividimos Europa en celdas de 1 Km^2 de superficie y 10 Km de altura (es decir, columnas compuestas por 10 celdas verticales de altura 1 Km y superficie 1 Km^2)
    - Europa ~ $5\cdot 10^9$ celdas (cada celda requiere de 0,1 TB)
    - Δt = 1 minuto: 100 flops por celda (estimación optimista)
    - $100*5*10^9$ operaciones en menos de un minuto

Por último, indicar que con un mayor número de procesadores, se aumenta algunos de los parámetros que definen una simulación numérica:

- Resolución de problemas en menor tiempo de ejecución, usando más procesadores (crítico en *T*, productividad)
- Resolución de problemas con mayor precisión, usando más memoria (crítico en *P*, rendimiento)
- Resolución de problemas más ambiciosos, usando modelos matemáticos más complejos (crítico en *C*, computación)

Nociones Básicas de Arquitecturas Actuales

Los clústeres de computación se han venido dividiendo en tres grandes bloques acorde a cómo compartían la memoria los distintos procesadores de cálculo. Así, tenemos:

- Memoria compartida (denominados también UMA o SMP por *Uniform Memory Access* o *Symmetric Multi-Processing*)
- Memoria Distribuida (o MPP por sus siglas en inglés de *Massively Multi-Processing*)
- NUMA (por sus siglas en inglés *Non Uniform Memory Access*)

Como se puede ver en la Fig. 9, los distintos procesadores $P_1 \ldots P_n$ acceden a las distintas memorias $M_1 \ldots M_n$ a través de la red de interconexión de manera uniforme y homogénea de tal forma que todos los procesadores ven el mismo espacio de memoria compartido por todos ellos. Por el contrario, en las plataformas computacionales de memoria distribuida, cada procesador P_i ve su propia memoria M_i y sólo esa. Las arquitecturas NUMA se han demostrado como las más versátiles y eficientes por ofrecer una memoria virtualmente compartida y físicamente distribuida, es decir, cada procesador tiene su memoria pero están armados y configurados de tal forma que comparten la memoria los unos con los otros. Esta configuración reduce la eficiencia el acceso a memoria de cada procesador de forma individual, pero aumenta la versatilidad y eficiencia del clúster como un todo. Por tanto, en las arquitecturas NUMA, cada procesador tiene su propia memoria local pero también tiene acceso a la memoria de otros (de forma más lenta). Gracias a ello, ofrece la "escalabilidad" de MPP y la programación simple de SMP.

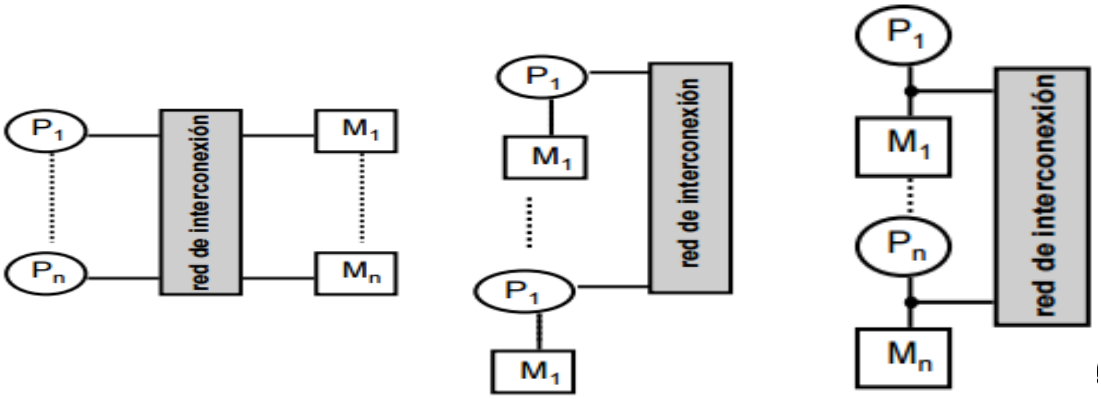


Figura 9. Esquemas de clústeres de memoria compartida (izquierda), distribuida (centro) y NUMA (derecha).

Un segundo avance ha sido las arquitecturas CC-NUMA (*Cache Coherent NUMA*). Suponen una extensión de SMP para soportar capacidades MPP con acceso por hardware a la memoria de otros procesadores mediantes canales de conexión de componentes (llamados *buses*) o también a través del Sistema Operativo. La memoria caché es la memoria de acceso rápido del procesador y es la que guarda temporalmente los datos procesados recientemente siendo además más pequeña que la memoria principal.

Generalizando, las infraestructuras HPC son lo que llamamos supercomputadores o clústeres locales que se hospedan en un único Centro de Proceso de Datos (CPD) y actualmente tienen arquitecturas NUMA. Se suelen definir bien por la cantidad de FLOPS que alcanzan en determinados tests (tales como el Linpack[4]) o por el número de *cores* (unidades de cálculo o CPU) que tienen. La arquitectura sigue el esquema: Cores que se agrupan en Procesadores que se agrupan en Nodos que se agrupan en

[4] https://www.top500.org/project/linpack/

Servidores/Chasis que finalmente forman el supercomputador. Todos estos componentes se conectan por redes de baja latencia, es decir, redes muy rápidas de intercomunicación, tales como Infiniband u Omni-Path.

En la actualidad, existen lo que se ha llamado aceleradores o co-procesadores, es decir, procesadores distintos a las tradicionales CPU que ofrecen un rendimiento mucho más alto para algunos cálculos específicos. Los principales son:

- GPU (Graphic Processing Units) especialmente indicados para cálculo vectorial y que son MPP en modo local
- Xeon Phi, especialmente indicados para *multithreading*, es decir, para paralelización dentro del procesador

Las GPU son los procesadores gráficos que tienen las consolas de juego tales como las Play Station. Por su parte, las Phi fueron la apuesta que hizo Intel para competir con Nvidia y sus GPU, pero retiraron el producto a finales de 2017 para relanzarlo con una arquitectura especialmente diseñada para cálculos de aprendizaje profundo (*Deep learning*), esto es, algoritmos para que los computadores vayan aprendiendo a reconocer objetos o representaciones gráficas.

Los supercomputadores actuales más potentes suelen tener una combinación de nodos con procesadores CPU tradicionales y nodos con co-procesadores.

Por su parte, las infraestructuras HTC generalmente son del tipo MPP –aunque no únicamente- y se suelen agrupar en federación de sitios (*sites*) heterogéneos distribuidos geográficamente, es decir, que cada uno cuenta con procesadores y nodos distintos en sitios muy alejados los unos de los otros. Por todo ello, se salen del dominio de administración de la institución (no son clústeres locales y, así, cada institución administra sus propios recursos de manera particular) y los sitios están conectados por fibra óptica o redes de datos (red Ethernet).

Las infraestructuras Grid son de tipo HTC, aunque ya están en desuso salvo en ciertos ámbitos tales como la física de altas energías donde el empleo de las mismas es óptimo para el tipo de cálculo que realizan.

Las infraestructuras Grid se han ido sustituyendo por las infraestructuras en la nube (*cloud*), las cuales ofrecen no ya recursos permanentes, sino bajo demanda. Las infraestructuras en la nube pueden ser públicas (tales como Amazon en las cuales cualquier persona puede acceder a sus recursos abonando una cantidad de dinero por ello) y privadas (bien para investigación o de empresas particulares para sus empleados).

Para poder ofrecer ese servicio bajo demanda se usa la técnica de la virtualización, en la cual sobre una máquina física se levanta una máquina virtual con las características requeridas por el usuario.

Por último, indicar que una tendencia de investigación muy potente hoy en día es el uso de memorias NVRAM (*Non-Volatile Random Access Memory*), las cuales son un tipo de memoria de acceso aleatorio que no pierde la información almacenada al cortar la alimentación eléctrica y que ofrece grandes posibilidades para aumentar la eficiencia de los cálculos y la recuperación de posibles errores.

Nociones básicas para la paralelización de aplicaciones

En computación científica, y no solamente en ella, el objetivo es realizar cálculos lo más eficientemente posible para con ello poder resolver nuestro problema de interés de forma satisfactoria y rápida. Con ese aumento de la eficiencia conseguimos tanto obtener nuestro resultado antes como poder abordar problemas más ambiciosos.

Una de las técnicas básicas de optimización en la ejecución de un código es su paralelización. Con ella y como se ha visto, el cálculo se divide en partes que se ejecutan de forma paralela o distribuida en contraposición con la ejecución secuencial en la cual el código se ejecuta en un solo *core* siguiendo los pasos definido dentro del mismo por orden secuencial.

Para esta metodología, se dispone de una mayor potencia de cálculo y almacenamiento, redes de interconexión cada vez más rápidas y problemas que se pueden dividir en partes más pequeñas de tal forma que cada una de ellas se ejecute en un sitio distinto de la infraestructura computacional. El problema por tanto es saber cómo se divide ese problema, esto es, cómo se diseña esa paralelización o distribución de las tareas que lo componen y una vez que se ha hecho ese diseño, cómo se ejecutan esas tareas distribuidas para que así se optimice la eficiencia computacional.

El quid de la cuestión es saber qué tipo de infraestructura sirve para resolver el problema de interés. Esto es así dado que no todo sirve para todo (aunque se pueda conseguir el objetivo final, no es bueno matar moscas a cañonazos) y a veces hay que ser consciente de que una solución aproximada con un pequeño margen de error es más que suficiente para el propósito perseguido. O en otras palabras, no es imprescindible la solución exacta en caso de que ésta sea posible obtenerse si con ello el cálculo va a durar mucho más tiempo y la solución aproximada es suficientemente correcta a pesar del error que pueda albergar.

Otro aspecto a tener en cuenta es que hay que tener la infraestructura computacional ocupada con problemas que realmente lo requieran, pues el mayor supercomputador se puede llenar fácilmente con códigos que buscan soluciones por “fuerza bruta”, pero hacer esto sería desperdiciar los recursos con los que se cuenta.

Es importante igualmente tener en cuenta para realizar la paralelización ciertas leyes que han regido en la computación de altas prestaciones:

- Ley de Moore (1965 / 1975), por la cual la densidad de circuitos en un chip (*cores* dentro de un procesador) se dobla cada 12 / 24 meses[5]
- Ley de Amdahl (1967), por la cual la eficiencia obtenida en una implementación paralela viene limitada por la parte secuencial, por ello, el límite superior de mejora obtenida es independiente del número de procesadores
- Respuesta de Gustafson (1988), en la práctica, la mayoría de las aplicaciones son críticas en precisión, por lo que el tamaño de problema crece con el número de procesadores

Teniendo estas leyes en cuenta, hasta esta década se han ido obteniendo mejoras con ganancias en un factor:

- ~80 cada 10 años a partir de las metodologías computacionales (software)
- ~50 cada 10 años a partir de la arquitectura (hardware)

Sin embargo y al igual que pasa con la Ley de Moore, estas ganancias se van estancando poco a poco. Por todo ello y ante el reto de la Exaescala (realizar 10^{18} operaciones en punto flotante en un segundo), la tendencia en investigación a día de hoy es el co-diseño, es decir, diseñar las nuevas arquitecturas teniendo en cuenta todos los factores implicados la ejecución de aplicaciones muy demandantes computacionalmente.

Teniendo en cuenta todo lo expresado hasta el momento, hay que realizar un análisis previo a la paralelización antes de implementarla. Algunos factores a considerar son:

- La complejidad numérica del código/algoritmo sabiendo su escalabilidad (se habla de ella un poco más adelante) y los recursos que serán necesarios
- El grado de paralelismo del código/algoritmo a partir de la Ley de Amdahl
- La granularidad de la implementación paralela (en cuantos trocitos se dividirá el cálculo y si este número puede variar) observando el grado de acoplamiento requerido en la arquitectura subyacente
- El propio análisis del código/algoritmo para responder a preguntas tales como ¿por qué paralelizarlo?, ¿dónde hacerlo?, ¿cómo hacerlo?...
    - Este punto implica identificar las partes críticas del código, las que son idóneas para paralelizar, la consideración del principio de Pareto o Regla 80/20[6]
- Tratar de prever qué ganancia (*speedup*) final se espera, esto es, cuánto más rápido y eficiente va a ser el código/algoritmo tras su paralelización y si va a compensar el esfuerzo que se le dedique

[5] Esta ley ya no se cumple en los últimos procesadores, esto es, ya no se es capaz de doblar la capacidad en períodos de tiempo análogos dado que la miniaturización tiene un límite físico

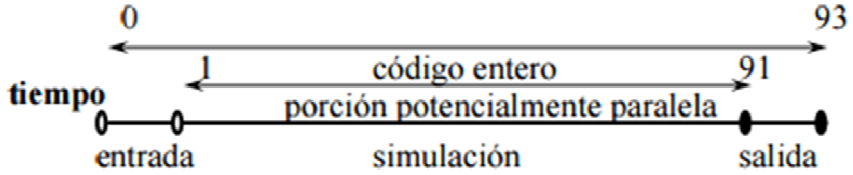
[6] The application of the Pareto principle in software engineering. Ankunda R. Kiremire 19th October, 2011

Con este último punto, se llega a la escalabilidad anteriormente comentada. Una ganancia de 2 quiere decir que, manteniendo fijo el tamaño de problema que queremos resolver, doblando el número de *cores*, el mismo código se ejecuta dos veces más rápido. Si esta tendencia se mantiene según se aumenta el número de *cores*, se dice que el código tiene muy buena escalabilidad (fuerte en este caso, en contraposición a la escalabilidad débil en la que el número de *cores* se mantiene constante y lo que se varía es el tamaño del problema).

Para ver esta característica más detenidamente, hay que tener en cuenta que un código científico se compone básicamente de una parte de entrada (prólogo), una parte principal de ejecución y una parte de salida (epílogo). Habiendo identificado estas partes, se mide:

- el tiempo de ejecución (*wall-time*) de la parte del programa que se va a paralelizar (código potencialmente paralelo) usando las rutinas de tiempos que nos aporte el S. Operativo
    - para esta medida se quitan las partes secuenciales, como las sumas globales o el tiempo requerido en la entrada y/o salida de datos
- El tiempo del programa entero, desde la primera sentencia hasta la última

Tendiendo estos valores, es posible calcular el contenido paralelo *c* que se define como la razón entre el tiempo de ejecución de la parte paralela y el tiempo de ejecución total. En la Fig. 10 se describe un ejemplo de cálculo de este parámetro.



$$c = \frac{Tiempo\_parte\_paralela}{Tiempo\_total} = \frac{90}{93} = 0.9667$$

$$speedup\_teorico = \frac{1}{(1-c) + \frac{c}{p}} = \frac{1}{0.0323 + \frac{0.9677}{p}}$$

Figura 10.Ejemplo de cálculo del contenido paralelo; p representa el número de *cores*.

Realizando el cálculo de *c* se puede prever cómo de escalable será el código de interés. A mayor *c* mejor escalabilidad fuerte habrá puesto que menos penalización habrá de la parte secuencial tal y como indicaba la Ley de Amdhal. Un ejemplo de este fenómeno se puede ver en la Fig. 11 en la que se representa cómo la ganancia o *speedup* generalmente se va reduciendo según aumenta el número de *cores*.

Nº de CPU	*Speedup* teórico
1	1,00
2	1,94
3	2,82
4	3,65
5	4,43
6	5,17
7	5,86
8	6,52
9	7,15
10	7,75

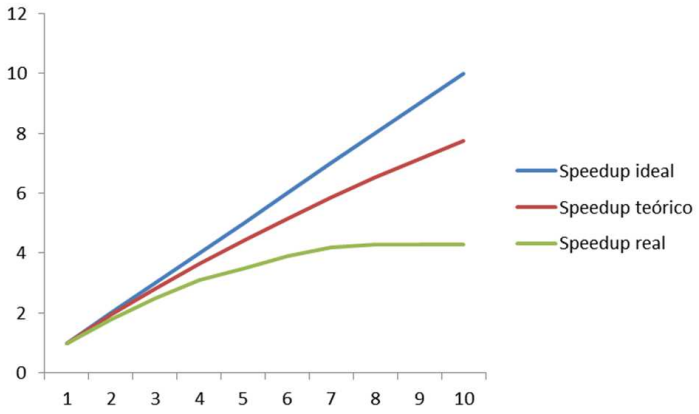


Figura 11.Ejemplo de ganancias ideal, teórica y real.

Se ve que la diferencia creciente entre el *speedup* ideal (que tiene pendiente igual a 1) y el teórico (marcado por el cálculo de *c*) es debida al tanto por ciento secuencial que cada vez se hace más dominante. Por su parte, el *speedup* real se debe a factores tales como el tiempo consumido en creación, sincronización y comunicación, la competencia por recursos o la desigualdad en la carga de los procesadores o los nodos.

Planificación y gestión de recursos

El siguiente paso a considerar una vez que se ha implementado la paralelización de un código es determinar cómo se van a ejecutar las tareas paralelas o distribuidas en un clúster determinado para que así la ejecución sea lo más eficiente posible. Esto es, hay que gestionar de forma óptimo los recursos computacionales con los que se cuenta planificando dónde y cómo se van a ejecutar las tareas y qué tamaño han de tener estas últimas para el caso específico de un clúster determinado.

Empezando por el último punto, para paralelizar una aplicación científica acoplada (sus partes no son independientes, por lo que las tareas paralelas implementadas dependen las unas de las otras), hay que definir la granularidad de la división, es decir, cuán pequeñas hacemos las particiones. Por su parte, en un cálculo HTC, la división es de todas y cada una de las partes del cálculo independientes

La granularidad es una medida de la cantidad de computación de un proceso software. Por ello, se considera como el segmento de código escogido para su procesamiento paralelo. Así, esta granularidad define la arquitectura óptima en la cual ha de ejecutarse, por lo que el proceso se puede idear al revés: escoger una granularidad en el código -si esto fuera posible- que conlleve el uso óptimo de la infraestructura computacional.

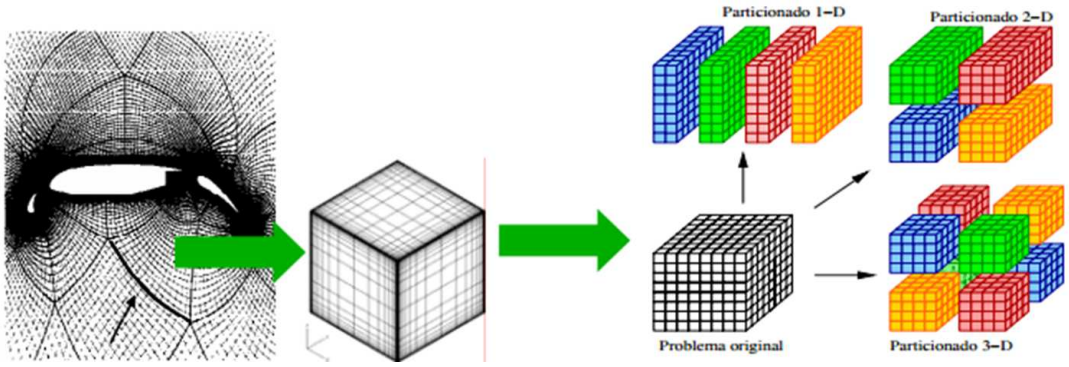


Figura 12.Ejemplos de granularidad en 1, 2 y 3 dimensiones.

Una vez que se ha terminado de implementar el código y de definir la granularidad que se ha estimado como óptima, se requiere de distintas pruebas de ejecución de ese código en la arquitectura de la que se dispone para corroborar si las estimaciones teóricas se cumplen o no. En otras palabras, hay que analizar cómo reducir el tiempo de ejecución si observamos que la ejecución final no es lo suficientemente eficiente.

Este proceso es una optimización tanto a nivel del código como de su ejecución propiamente dicha. La optimización se produce a partir de varios factores: tipo de procesador; procesos de Entrada/Salida en el código; uso y acceso a la memoria de la infraestructura; llamadas al sistema; o, el propio conocimiento del código. Por todo ello, es muy común el uso de herramientas para saber cómo se comporta el código (*profiling*), pues indican qué partes del código consumen más recursos.

Un análisis más detallado de los puntos anteriores se recoge en el siguiente esquema:

- Procesador
  - Opciones de compilación para optimización secuencial y paralelización automática
  - Técnicas de optimización para mejorar la explotación de la arquitectura y de la memoria caché
  - Inclusión de directivas de paralelización en el código
- Entrada/Salida
  - Reorganizar la E/S para evitar muchas peticiones cortas
  - Funciones para mapear ficheros en memoria
  - Funciones para indicar al sistema el uso de las páginas de fichero en memoria
- Memoria virtual
  - Técnicas de optimización para mejorar la explotación de la memoria virtual
  - Funciones para indicar al sistema el uso de las páginas de fichero en memoria
- Conocimiento del código
  - Aplicación de distintos modelos para un mismo dato de entrada
  - Álgebra aplicada a los bucles

- Determinación de los cálculos más pesados dentro del código

El último punto a tener en cuenta para la ejecución de un código paralelizado en una plataforma HPC o HTC específica es adaptar la paralelización y la planificación a las políticas de uso que la rigen. Esto es así porque un usuario comparte los recursos con otros usuarios (generalmente, si se accede a un supercomputador con 100.000 *cores* no se puede disponer de todos ellos, sino de una fracción de los mismos).

El escenario por tanto es aquel en el que las infraestructuras computacionales a partir de cierto tamaño están compartidas por varios usuarios y gestionadas por administradores que disponen de permisos para el uso de los recursos superiores a los de los usuarios. Esto es así para que los administradores implementen reglas de uso (AUP) que delimitan el software que está instalado, el nº de recursos al que cada usuario puede acceder, el empleo de esos recursos para que la máquina esté operativa de la forma más eficiente posible, etc. Ejemplos típicos de políticas de uso de un supercomputador son: el nº máximo de procesadores permitidos a un único usuario; el nº máximo de trabajos permitidos a un único usuario; el nº máximo de procesadores y trabajos permitidos a un grupo de usuarios; el tiempo máximo que puede durar un trabajo ejecutándose; el porcentaje reservado para la ejecución de trabajos paralelos; etc.

Como consecuencia, el usuario ha de planificar cómo se distribuyen sus tareas paralelas porque de nada, por ejemplo, implementar una granularidad 3D para 1.000 procesos paralelos si luego sólo se va a poder disponer de 500 *cores*.

Esa planificación se implementa con unas herramientas informáticas que se configuran por los administradores, pero que también se usan por los usuarios finales para mandar sus trabajos a ejecutarse en el supercomputador. Estos programas son los gestores de recursos o planificadores (*scheduler*) y están destinados a la gestión de los recursos computacionales con el fin de ofrecer una buena calidad de servicio (QoS). Básicamente cuentan con:

- Sistemas de colas *batch*, en los que las peticiones de trabajos a ejecutarse (con un nº de recursos solicitados) se encolan para irles dando prioridad
- Sistema de gestión de carga para entornos distribuidos HTC
- Librerías que permiten la coordinación e intercomunicación de los trabajos paralelos o distribuidos (MPI, OpenMP, DRMAA…)
- Herramientas de monitorización de recursos y sistemas de información de *sites*

Algunos ejemplos de gestores de recursos para supercomputación HPC son Maui/Moab/PBS[7] o Slurm[8], siendo este último el más utilizado en la actualidad en los supercomputadores más potentes del mundo. Tanto los administradores como los

[7] http://www.adaptivecomputing.com/products/open-source/maui/

[8] https://slurm.schedmd.com/

usuarios finales disponen gracias a ellos de una serie de comandos con los que se ejecuta el envío de trabajos y se administra el clúster.

Los gestores de recursos disponen de algoritmos de planificación que gestionan automáticamente la carga del supercomputador acorde a las políticas AUP y a otras directrices que se puedan definir. Estas directrices pueden referirse a tiempos de espera de trabajos encolados antes de ejecutarse o a la utilidad o no de compactar tareas.

La compactación o distribución de tareas viene dada porque actualmente los nodos pueden contar, por ejemplo, con dos procesadores que cuentan a su vez con 48 *cores* cada uno. Parece evidente que si un usuario manda un trabajo con 96 tareas paralelas, éstas se planifiquen de tal forma que se ejecuten dentro de un mismo nodo, ¿pero que pasa si un usuario manda un número que no es múltiplo de 96? ¿O si lo hace en un número menor? ¿Cómo se gestiona el clúster cuando hay multitud de usuarios cada uno con sus requerimientos particulares? ¿Qué hace el planificador con los nodos parcialmente ocupados que tienen *cores* libres? La complejidad de la planificación de tareas puede llegar a ser muy elevada y es una línea de investigación por sí misma

Como resumen, se puede indicar que en ciertas ocasiones compensa compactar tareas de distintos usuarios en un mismo nodo para liberar recursos que requieran de trabajos paralelos más demandantes o para dejar parte del supercomputador libre con el fin de realizar trabajos de configuración, actualización, apagado, etc. Igualmente, también puede ser interesante para aumentar la localidad para reducir el tiempo empleado en la gestión de los recursos (*overhead*), por ejemplo, el ancho de banda de procesos con pasos de mensajes MPI.

Sin embargo, a veces y por el contrario, se aumentará el rendimiento general del clúster planificando tareas en distintos nodos dadas las características de las mismas (un ejemplo puede ser tareas que requieran mucha memoria).

Todo este movimiento de compactación o distribución de tareas puede realizarse cuando los trabajos están encolados antes de ejecutarse (planificación) o una vez que ya se ha iniciado la ejecución de las mismas (migración) gracias a técnicas adicionales llamadas puntos de chequeo (se guarda en memoria el estado intermedio de una tarea y se reinicia en un nodo distinto).

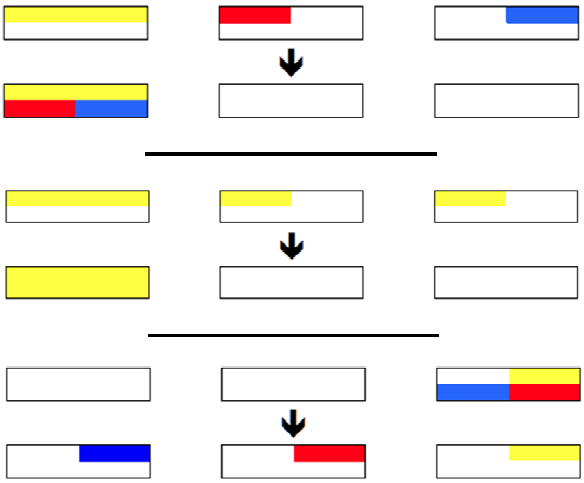


Figura 13.Esquemas de .compactación de tareas (arriba), aumento de localidad (medio) y distribución de tareas (debajo)

En relación a la ejecución de trabajos en entornos distribuidos HTC, simplemente mencionar que ésta es mucho más complicada. En la planificación, aprovisionamiento y gestión en HTC siguen siendo válidos algunos de los conceptos anteriormente citados, pero el escenario es distinto. Así, la infraestructura:

- Dispone de *sites* heterogéneos distribuidos geográficamente
  - Distintos recursos (potencia de cálculo, memoria, etc.)
  - Distintas configuraciones
  - Distinto ancho de banda para acceder a ellos
- El movimiento de grandes volúmenes de datos es aún más problemático
- Mayor posibilidad de fallos y errores de hardware y software
- El problema de ajustar carga de trabajo y recursos disponibles es por definición NP-Completo
  - Se define como Nondeterministic Polynomial time, es decir, no tiene soluciones exactas en un período de tiempo finito
  - Únicamente se pueden hallar soluciones sub-óptimas
- La infraestructura es dinámica (la oferta de recursos disponibles no es constante al contrario que en un supercomputador salvo errores por rotura)