# UNIVERSIDAD DE BURGOS

## Área de Tecnología Electrónica

# Programación paralela e híbrida.

José María Cámara Nebreda, César Represa Pérez, Pedro Luis Sánchez Ortega
***Programación Paralela e Híbrida***. 2015
Área de Tecnología Electrónica
Departamento de Ingeniería Electromecánica
Universidad de Burgos

Introducción ..... 5
LA PROGRAMACIÓN EN PARALELO ..... 5
Práctica 1: Multiplicación de matrices en MPI. ..... 9
OBJETIVOS ..... 9
CONCEPTOS TEÓRICOS ..... 9
CUESTIONES ..... 11
DIAGRAMA DE FLUJO ..... 11
Práctica 2: Medida del rendimiento ..... 13
OBJETIVOS ..... 13
CONCEPTOS TEÓRICOS ..... 13
REALIZACIÓN PRÁCTICA ..... 17
CUESTIONES ..... 18
Práctica 3: Introducción a la Programación Híbrida ..... 19
OBJETIVOS ..... 19
CONCEPTOS TEÓRICOS ..... 19
REALIZACIÓN PRÁCTICA ..... 22
CUESTIONES ..... 22
Práctica 4: Programación Híbrida ..... 23
OBJETIVOS ..... 23
CONCEPTOS TEÓRICOS ..... 23
EJERCICIO PRÁCTICO ..... 24
Práctica 5: MPI vs OpenMP ..... 25
OBJETIVOS ..... 25
CONCEPTOS TEÓRICOS ..... 25
EJERCICIO PRÁCTICO ..... 25
Práctica 6: Envío de trabajos al cluster ..... 26
OBJETIVOS ..... 26
CONCEPTOS TEÓRICOS ..... 26
EJERCICIO PRÁCTICO ..... 31
Práctica 7: Planificación de trabajos ..... 32
OBJETIVOS ..... 32
CONCEPTOS TEÓRICOS ..... 32
EJERCICIO PRÁCTICO ..... 35
Práctica 8: Planificación de trabajos II. ..... 36
OBJETIVOS ..... 36
CONCEPTOS TEÓRICOS ..... 36
EJERCICIO PRÁCTICO ..... 36
Práctica 9: Planifiación de trabjos III ..... 37
OBJETIVOS ..... 37
CONCEPTOS TEÓRICOS ..... 37
EJERCICIO PRÁCTICO ..... 37
Práctica 10: Batalla de rendimiento. ..... 38
OBJETIVOS ..... 38
CONCEPTOS TEÓRICOS ..... 38
EJERCICIO PRÁCTICO ..... 38

Apéndice A: Instalación de DeinoMPI ..... 39
Instalación ..... 39
Configuración ..... 39
Ejecutar Aplicaciones ..... 40
Entorno Gráfico ..... 40
Apéndice B: Configuración de un Proyecto en Visual Studio 2010 y versiones sucesivas. ..... 44
Apéndice C: Configuración de MS-MPI. ..... 49

# Introducción

## *LA PROGRAMACIÓN EN PARALELO*

En este apartado introductorio vamos a contemplar las diferentes alternativas a la hora de programar sistemas que emplean paralelismo explícito, ya que aquellos que optan por el paralelismo implícito no requieren de una programación especializada.

Como sabemos, las máquinas que admiten este tipo de programación son las de **arquitectura MIMD**, tanto multiprocesadores como multicomputadores.

Las **arquitecturas multiprocesador** hemos visto que emplean un espacio de memoria compartida por los diferentes procesadores a la que se conectan a través de un bus común con el fin de intercambiar información entre ellos para completar una labor común: el programa paralelo. El desarrollo de algoritmos adecuados para este tipo de máquinas exige partir de una abstracción del hardware sobre la que se puedan implementar algoritmos. A este efecto se crea un modelo teórico de máquina multiprocesador que no tiene reflejo en ninguna arquitectura real, pero que sirve al objetivo que se busca. Se trata de la máquina PRAM (parallel random access machine) o máquina paralela de acceso aleatorio. Este modelo presupone la existencia de una memoria compartida por todos los procesadores a la que pueden acceder en un tiempo unitario, es decir, el mismo tiempo de acceso que a su memoria local en caso de que esta exista.

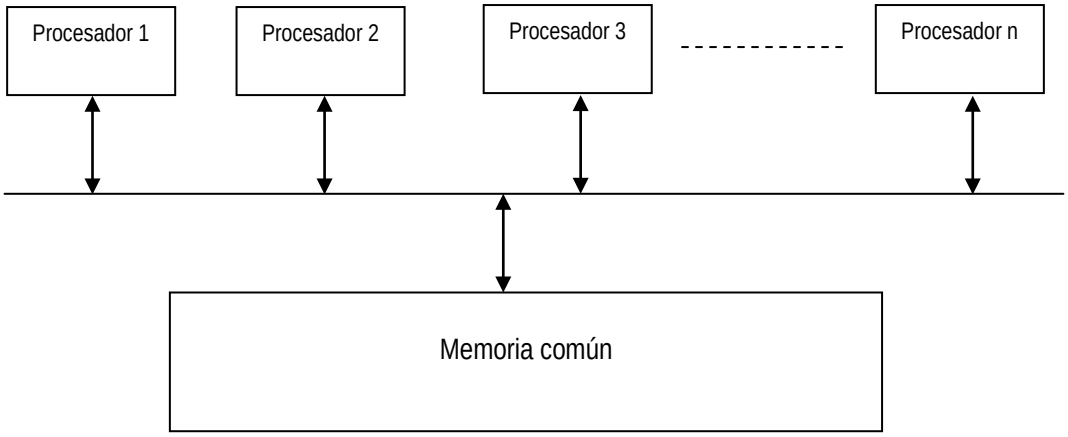



*Figura 1.1. Uso de memoria común.*

Este supuesto es en sí mismo imposible, pero se encuentra a medio camino entre la posibilidad real de que la memoria común sea un dispositivo independiente de la memoria local de los procesadores y la situación también real en la que la memoria común se implementa en las memorias locales de los procesadores.

Nuestro "cluster" tiene estructura de *sistema multicomputador*, independientemente de que alguno de sus nodos pueda ser un multiprocesador. En esta situación, no vamos a hacer más abstracción del hardware que la de olvidarnos del sistema de conexión que tengamos. Lo que sí vamos a hacer es crear un modelo de programación apropiado. Consistirá en considerar los programas estructurados en procesos comunicados entre sí por canales. De esta forma, dividiremos la programación en

procesos independientes, cada uno de los cuales dispone de su flujo de instrucciones secuencial, sus datos de entrada, salida e intermedios y una serie de puertos de entrada-salida que le permiten comunicarse con otros procesos a través de canales creados a tal efecto por los que se intercambian mensajes.

La mayoría de los problemas que abordemos tendrán varias posibles soluciones paralelas. Trataremos de obtener la más ventajosa, para lo cual tendremos en cuenta dos aspectos:

- Buscaremos incrementar al máximo el **rendimiento**, entendiendo éste como ejecución más rápida posible del programa completo. Para ello deberemos hacer especial hincapié en el concepto de "localidad", consistente en procurar que se realice la mayor cantidad de trabajo dentro de un proceso con datos existentes en la memoria local del computador, disminuyendo en intercambio de mensajes con otros nodos para acceder a los datos. Siempre es más rápido encontrar un dato en la propia memoria que en la de otro nodo, especialmente si está conectado a través de una red como es el caso.

- No debemos dejar de lado el concepto de **escalabilidad**, especialmente si estamos desarrollando software que habrá de correr en una arquitectura dinámica en la que el número de nodos pueda variar. En este caso, la división en procesos debe permitir aprovechar al máximo la capacidad del sistema independientemente del número de nodos disponibles en cada momento.

Vamos a explicar un posible procedimiento para desarrollar aplicaciones paralelas de forma ordenada. Debe entenderse que la programación paralela, al igual que la programación secuencial, es una tarea fundamentalmente creativa, por lo que lo que se va a exponer a continuación no pretende ser sino una secuencia de etapas que se considera interesante cubrir ordenadamente para obtener buenos resultados. El trabajo que implica cada tarea es una labor esencialmente creativa cuyos resultados dependen de las aptitudes y experiencia del programador. El procedimiento consta de cuatro etapas:

- **Fragmentación**: en esta fase inicial se intenta localizar las máximas posibilidades de paralelismo a base de descomponer la programación en tareas tan pequeñas como sea posible. No se debe plantear en esta fase la conveniencia o no de crear tareas tan simples, dado que es el punto de partida para estudiar las posibilidades de paralelización. Existen dos grandes criterios que sirven de guía para esta división:

  - El criterio funcional: contempla la naturaleza del trabajo que debe realizar el programa buscando posibles divisiones en él.

  - El criterio de datos: analiza la naturaleza de los datos que va a manejar el programa para estructurarlos en grupos de tamaño mínimo.

- **Comunicación**: partiendo de las tareas identificadas en la fase anterior, se analizan las necesidades de comunicación entre ellas.

- **Aglomeración**: dado que el coste de las comunicaciones en cuanto a rendimiento es alto, se tratará de agrupar las tareas descritas con anterioridad en otras de tamaño mayor que

busquen minimizar la necesidad de comunicación. De esta forma se generarán los procesos que se van a terminar programando.

- **Mapeo**: una vez establecida la estructura del programa, falta asignar los procesos creados a los computadores disponibles. La estrategia es diferente según se haya empleado un criterio funcional o de datos a la hora realizar la fragmentación. Una condición necesaria es que existan al menos tantos procesos como máquinas, ya que de lo contrario alguna quedaría sin trabajo. Si las máquinas son iguales, sería aconsejable igualar el número de procesos al número de máquinas; si no es así, se puede asignar más procesos a las máquinas más potentes. También existe la posibilidad de que los procesos se vayan asignando de forma dinámica, analizando qué máquinas tienen una menor carga de trabajo para asignarles más. Esta situación se da especialmente en las denominadas estructuras SPMD (un solo programa con flujo de datos múltiple). En ellas todas las máquinas realizan procesos idénticos sobre datos diferentes. Cuando una de ellas termina se le asignan nuevos datos para que siga trabajando.

Siguiendo este procedimiento se puede llegar a múltiples posibilidades de programación. La experiencia orientará al programador para que pueda llegar a la más conveniente.

Volviendo a la estructura de nuestro hardware debemos recordar que en esencia se trata de un entorno de memoria distribuida. En este caso, la programación paralela se ha de explicitar en torno a un sistema de paso de mensajes. A pesar de ello, cada nodo del cluster va a estar integrado por un procesador multinúcleo que en esencia es un pequeño sistema de memoria compartida por lo que admitiría una programación basada en este principio pero que no sería extensible al conjunto del sistema. Dado que este tipo de estructura es la habitual en los supercomputadores actuales y de cara a explotar al máximo las posibilidades del hardware disponible, se plantea un paradigma de programación que combine los dos mencionados. Se trata de la denominada programación híbrida consistente en la explicitación de procesos independiente que se comunican mediante paso de mensajes y que se adaptan perfectamente a una estructura de memoria distribuida, pero que internamente se desintegran en múltiples hilos que se comunican mendiante variables compartidas en memoria común. Los hilos pertenecientes a un mismo proceso no pueden residir en nodos diferentes sino que se van a ejecutar dentro de los diferentes procesadores o núcleos de un mismo nodo.

En este curso se van a dar por conocidos los conceptos de programación paralela de paso de mensajes asociados al entorno MPI, que ya fueron vistos en el Grado de Ingeniería Informática. En caso contrario se recomienda al alumno la realización de la primera parte del curso de Programación MPI, de las práctica 0 a la 5. Enlazaremos con ese curso no obstante en lo que fue la práctica 9 del mismo y que se convertirá en nuestra práctica 1.

# Práctica 1: Multiplicación de matrices en MPI.

## *OBJETIVOS*

- Demostrar los conocimientos adquiridos desarrollando una aplicación un poco más compleja que permita posteriormente evaluar el rendimiento del sistema.

## *CONCEPTOS TEÓRICOS*

En esta práctica no se van a incluir conceptos nuevos ya que se trata de aprovechar los disponibles. Por supuesto, en este punto no se han visto todas las posibilidades de MPI y la aplicación desarrollada en ningún momento va a ser la mejor de las posibles, pero sí se dispone de conocimientos suficientes para hacer un buen trabajo.

No obstante, puede resultar útil para el desarrollo de la aplicación comentar alguna posibilidad adicional de las funciones que ya conocemos. Concretamente, nos vamos a centrar en la función **`MPI_Recv`**:

```
int MPI_Recv(void *buf, int count, MPI_Datatype dtype, int source, int
   tag, MPI_Comm comm, MPI_Status *status)
```

Esta función proporciona la posibilidad recibir un mensaje sin tener que conocer previamente la fuente ni la etiqueta. Para ello es necesario utilizar el valor `MPI_ANY_SOURCE` como parámetro `source` y el valor `MPI_ANY_TAG` como parámetro `tag`. Posteriormente podemos comprobar el valor de dichos parámetros a través de la estructura del tipo `MPI_Status` que hasta el momento no hemos utilizado. Esta estructura está formada por tres elementos: `MPI_SOURCE`, `MPI_TAG` y `MPI_ERROR`. El primero proporciona la fuente del mensaje que se ha recibido y el segundo proporciona la etiqueta que traía el mensaje; en este caso, si se recibió con el valor `MPI_ANY_TAG`, puede ser interesante conocer la etiqueta que puso el emisor del mensaje ya que se puede codificar algún mensaje en ella. El tercer parámetro proporciona el código de error que se haya podido producir. No vamos a entrar en qué posibles errores se codifican. El interés por el parámetro de estado se encuentra en este momento en la utilidad que puedan proporcionar los dos primeros parámetros y que pueden ayudar a mejorar el código de nuestro programa de aplicación.

## *REALIZACIÓN PRÁCTICA*

Se trata de desarrollar un programa de multiplicación de matrices en paralelo. La forma de realizar el reparto de trabajo entre los procesos es libre. Simplemente se pide que el tamaño de las matrices (cuadradas) pueda ser definido mediante argumento en línea de comando o bien pidiendo su valor por teclado, con el fin de poder realizar pruebas con matrices de diferentes tamaños. Asimismo se debe procurar no limitar el tamaño de las matrices en la medida de lo posible, por lo que se recomienda la reserva dinámica de memoria.

El proceso 0 inicializará las matrices con valores aleatorios de tipo `float` y repartirá la carga de forma adecuada, según el criterio del programador. En una primera fase, se imprimirá el tiempo empleado en la ejecución junto con el resultado en la salida estándar para comprobar que es correcto. Posteriormente se eliminará para permitir que las matrices puedan crecer en tamaño sin hacer esperar por una impresión de datos interminable.

***NOTA:***

Con el fin de realizar una reserva dinámica de memoria para matrices de un tamaño elevado teniendo la posibilidad de doble indexación es necesario recurrir a los dobles punteros, es decir, definir un array de punteros cada uno de los cuales apunta a una fila de la matriz:

```
// Declaramos un doble puntero para almacenar la matriz
// Esto nos permite referenciar los elementos de la matriz mediante [fila][columna]
float **Matriz;
// Incializamos el doble puntero para almacenar punteros a cada una de las filas de la matriz
Matriz = (float **) malloc(FILAS*sizeof(float *));
// Inicializamos cada puntero de "Matriz" con las posiciones de comienzo de cada fila
for (int i=0; i< FILAS; i++)
{
	Matriz[i] = (float *) malloc(COLUMNAS*sizeof(float));
}
// Ahora ya podemos utilizar la notacion [fila][columna] para nuestra matriz:
for (int i=0; i<FILAS; i++)
{
	for (int j=0; j<COLUMNAS; j++)
	{
		Matriz[i][j] = 0.0;
	}
}
```

Sin embargo, la reserva dinámica de memoria realizada de este modo no garantiza que los datos correspondientes a filas consecutivas sean almacenados de forma consecutiva en memoria, lo cual puede ser imprescindible para algunas funciones de envío. La solución estaría en enviar la matriz fila a fila. Si queremos conseguir un espacio de direccionamiento contiguo manteniendo la posibilidad de la doble indexación para las matrices va a ser necesario inicializar los punteros a las distintas filas de la matriz con valores consecutivos. Esto lo conseguimos del siguiente modo:

```
// Declaramos un doble puntero para almacenar la matriz
// Esto nos permite referenciar los elementos de la matriz mediante [fila][columna]
float **Matriz;
// Incializamos el doble puntero para almacenar punteros a cada una de las filas de la matriz
Matriz = (float **) malloc(FILAS*sizeof(float *));
// Declaramos un puntero para reservar espacio consecutivo para toda la matriz
float *Mf;
// Incializamos el puntero con el tamaño necesario para toda la matriz
// Esto nos garantiza un espacio de memoria en posiciones consecutivas
Mf = (float *) malloc(FILAS*COLUMNAS*sizeof(float));
// Inicializamos cada puntero de "Matriz" con las posiciones de comienzo de cada fila
for (int i=0; i< FILAS; i++)
{
	Matriz[i] = Mf + i* COLUMNAS;
}
// Ahora, ya podemos utilizar la notacion  [fila][columna] para nuestra matriz:
for (int i=0; i<FILAS; i++)
{
	for (int j=0; j<COLUMNAS; j++)
	{
		Matriz[i][j] = 0.0;
	}
}
```

Es muy importante notar que con esta última alternativa, en las funciones de envío se debe utilizar la dirección `Matriz[0]` como dirección de comienzo de los datos almacenados.

## *CUESTIONES*

- Para multiplicar A×B se puede pasar la matriz A completa a todos los procesos y la matriz B se puede repartir por columnas. Pensar otra alternativa.
- ¿Sería posible aprovechar la potencia de la topología cartesiana para facilitar la resolución de este problema?
- El hecho de pasar una de las matrices completa ralentiza notablemente el funcionamiento del programa. Plantear alguna posibilidad en la que no sea necesario trasladar tanta información. Comparar su rendimiento previsible con el de la aplicación que se ha desarrollado.

## *DIAGRAMA DE FLUJO*

# Práctica 2: Medida del rendimiento

## OBJETIVOS

- Evaluar el rendimiento del sistema de paso de mensajes en diferentes situaciones.
- Adquirir la capacidad de prever la potencia del sistema y extraerla logrando un compromiso entre el esfuerzo de programación y la optimización del código.

## CONCEPTOS TEÓRICOS

### *Rendimiento de un sistema de tamaño variable*

A continuación se van a definir una serie de conceptos manejados de forma habitual en la literatura sobre procesamiento en paralelo que permiten evaluar diferentes características de los sistemas y prever su rendimiento. Estos conceptos son los siguientes y se aplican a situaciones donde el número de procesadores utilizados para la ejecución de un programa varía a lo largo del tiempo de ejecución:

- **Grado de paralelismo (DOP)***:* Número de procesadores utilizados para ejecutar un programa en un instante de tiempo determinado. Esta función, **DOP** = *P(t)*, que nos da el grado de paralelismo en cada momento durante el tiempo de ejecución de un programa, se llama **perfil de paralelismo** de ese programa. No tiene porqué corresponder con el número de procesadores disponibles en el sistema (*n*). En las siguientes definiciones supondremos que se dispone de más procesadores que los necesarios para alcanzar el grado de paralelismo máximo de las aplicaciones, **máx**{*P(t)*} = *m* < *n*.

- **Trabajo total realizado**: Si $\Delta$ es el rendimiento, velocidad o potencia de cómputo de un solo procesador, medida en MIPS o MFLOPS, y suponemos que todos los procesadores son iguales, es posible evaluar el trabajo total realizado entre los instantes de tiempo $t_A$ y $t_B$ a partir de área bajo el perfil de paralelismo como:

$$W = \Delta \cdot \int_{t_A}^{t_B} P(t) \cdot dt \,.$$

  Normalmente el perfil de paralelismo es una gráfica definida a tramos (figura 10.1), por lo que se puede expresar el trabajo total realizado como:

$$W = \Delta \cdot \sum_{i=1}^{m} i \cdot t_i \,.$$

  En esta expresión $t_i$ es el intervalo de tiempo durante el cual el grado de paralelismo es *i*, siendo *m* el máximo grado de paralelismo durante todo el tiempo de ejecución del programa.

  Según esto, se cumple que la suma de los diferentes intervalos de tiempo es igual al tiempo de ejecución del programa:

$$\sum_{i=1}^{m} t_i = t_B - t_A .$$

- **Paralelismo medio**: Es la media aritmética del grado de paralelismo a lo largo del tiempo ejecución. Se expresa como:

$$\overline{P} = \frac{1}{t_B - t_A} \int_{t_A}^{t_B} P(t) \cdot dt = \frac{\sum_{i=1}^{m} i \cdot t_i}{\sum_{i=1}^{m} t_i} .$$

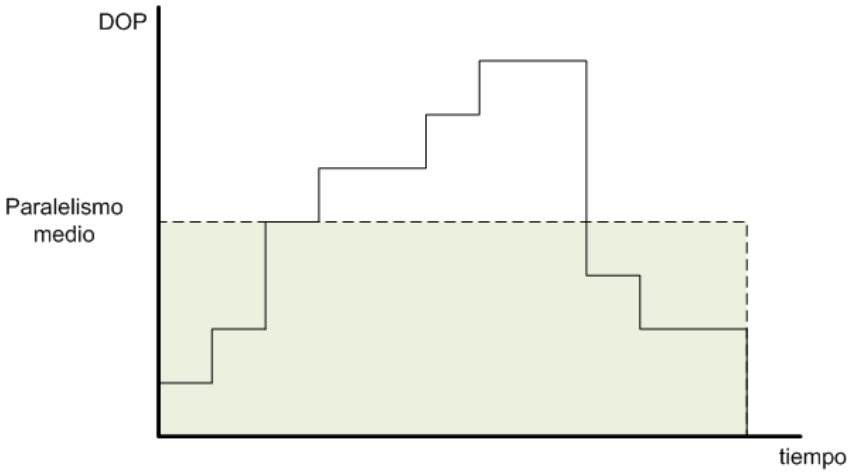



*Figura 10.1. Perfil de paralelismo: representación gráfica de P(t).*

- **Paralelismo disponible**: Máximo grado de paralelismo extraíble de un programa o aplicación, independientemente de las restricciones del hardware.

- **Máximo incremento de rendimiento alcanzable**: Sea $W_i = i \cdot \Delta \cdot t_i$ el trabajo realizado cuando **DOP** = *i*, de manera que $W = \sum_{i=1}^{m} W_i$ .

  En estas condiciones, el tiempo empleado por un solo procesador para realizar un trabajo $W_i$ es $t_i(1) = \frac{W_i}{\Delta}$; para *k* procesadores es $t_i(k) = \frac{W_i}{k \cdot \Delta}$, y para infinitos procesadores es $t_i(\infty) = \frac{W_i}{i \cdot \Delta}$ (sólo trabajan "*i*" procesadores ya que el grado de paralelismo es *i*).

  A partir de aquí se puede definir el **tiempo de respuesta** como:

$$T(1) = \sum_{i=1}^{m} t_i(1) = \sum_{i=1}^{m} \frac{W_i}{\Delta}$$

$$T(\infty) = \sum_{i=1}^{m} t_i(\infty) = \sum_{i=1}^{m} \frac{W_i}{i \cdot \Delta} .$$

El máximo rendimiento que puede proporcionar un sistema paralelo se da cuando en número de procesadores disponibles es ilimitado, por lo que el máximo incremento de rendimiento alcanzable (también denominado *speed-up asintótico*) se obtendrá del cociente entre éste y el rendimiento que proporciona un solo procesador. En términos de tiempos de respuesta se expresa como:

$$S_\infty = \frac{T(1)}{T(\infty)}.$$

Utilizando la definición de $W_i = i \cdot \Delta \cdot t_i$, la expresión del *speed-up asintótico* se puede poner como:

$$S_\infty = \frac{T(1)}{T(\infty)} = \frac{\sum_{i=1}^{m} \frac{W_i}{\Delta}}{\sum_{i=1}^{m} \frac{W_i}{i \cdot \Delta}} = \frac{\sum_{i=1}^{m} \frac{i \cdot \Delta \cdot t_i}{\Delta}}{\sum_{i=1}^{m} \frac{i \cdot \Delta \cdot t_i}{i \cdot \Delta}} = \frac{\sum_{i=1}^{m} i \cdot t_i}{\sum_{i=1}^{m} t_i} = \overline{P}.$$

Esto se puede simplificar diciendo que el máximo incremento de rendimiento alcanzable por un sistema paralelo que disponga de un número ilimitado de procesadores equivale al paralelismo medio intrínseco de la aplicación que se va a paralelizar. Lógicamente lo complicado es averiguar ese paralelismo intrínseco y lograrlo mediante la programación.

### *Rendimiento de un sistema con carga de trabajo fija*

Existen otras formas de evaluar la ganancia en rendimiento de un sistema. Una de ellas es considerar situaciones donde una carga de trabajo fija se puede repartir entre un diferente número de procesadores. Por ejemplo, decimos que un trabajo *W*, ya sea un programa o un conjunto de ellos, se va a ejecutar en "modo *i*" si para ello se van a emplear *i* procesadores, siendo $R_i$ el rendimiento o velocidad colectiva de todos ellos medida en MIPS o MFLOPS y $T(i) = W/R_i$ el tiempo de ejecución. Así, $R_1$ sería la velocidad de uno solo y $T(1) = W/R_1$ su correspondiente tiempo de ejecución. Supongamos que el trabajo *W* se realiza en *n* modos diferentes con una carga de trabajo diferente para cada modo, lo cual se refleja en un peso relativo $f_i$ que se le asigna a cada uno:

$$W = W_1 + W_2 + \cdots + W_n = f_1 \cdot W + f_2 \cdot W + \cdots + f_n \cdot W$$

En estas condiciones podemos definir la ganancia en rendimiento como el cociente entre el tiempo $T(1)$ empleado por un único procesador (modo 1) y el tiempo total *T* empleado por los *n* modos (modo 1, modo2,…, modo *n*):

$$S = \frac{T(1)}{T}.$$

El tiempo total de ejecución se calcula a partir del rendimiento y de la carga de trabajo de cada uno de los modos de ejecución:

$$T = \sum_{i=1}^{n} \frac{f_i \cdot W}{R_i} = W \cdot \sum_{i=1}^{n} \frac{f_i}{R_i}.$$

De esta manera, la ganancia en rendimiento se puede poner como:

$$S = \frac{T(1)}{T} = \frac{1/R_1}{\sum_{i=1}^{n} f_i/R_i}.$$

En una situación ideal en la que no existen retardos por comunicaciones o escasez de recursos, considerando un trabajo unidad ($W$ = 1) tendremos que si $R_1$ = 1, $R_i$ = $i$ :

$$S = \frac{1}{\sum_{i=1}^{n} f_i/i}.$$

En este contexto se enuncia la ley de Amdahl, donde suponemos que el trabajo se realiza en dos modos con pesos relativos $\alpha$ y $1-\alpha$. Una parte se realiza utilizando un procesador tal que $f_1 = \alpha$ y otra en "modo $n$" con $n$ procesadores tal que $f_n = 1-\alpha$, lo que quiere decir que una parte del trabajo se va a realizar en modo secuencial y el resto empleando la potencia máxima del sistema. En estas condiciones:

$$S = \frac{1}{\frac{\alpha}{1} + \frac{1-\alpha}{n}} = \frac{n}{1+(n-1)\cdot\alpha}.$$

Esto implica que si $\alpha$ = 0, es decir, si idealmente todo el trabajo se puede realizar en "modo $n$" utilizando la potencia máxima del sistema, entonces:

$$S = n.$$

Sin embargo, si esto no es posible, el máximo incremento de rendimiento alcanzable (*speed-up asintótico*) cuando $n \to \infty$ es:

$$S_\infty = \lim_{n\to\infty} S = \frac{1}{\alpha}.$$

Dicho de otro modo, el rendimiento del sistema se encuentra limitado por el peso de la parte secuencial del trabajo.

Existen algunos parámetros que son de utilidad para evaluar las diferentes características de un sistema paralelo con $n$ procesadores y son los siguientes:

- **Ganancia (*Speed-up*)**: Determina el grado de mejora del sistema:

$$S = \frac{T(1)}{T(n)} \le n.$$

  Se usa para indicar el grado de ganancia de velocidad de una computación paralela.

- **Eficiencia**: Determina el grado de aprovechamiento del sistema:

$$E = \frac{S}{n} = \frac{T(1)}{n \cdot T(n)} \leq 1.$$

La eficiencia mide la porción útil del trabajo total realizado por *n* procesadores. A partir de la eficiencia se puede definir la **escalabilidad**. Así, se dice que un sistema es escalable si la eficiencia *E*(*n*) del sistema se mantiene constante y cercana a la unidad.

- **Redundancia**: Es la relación entre el número total de operaciones que realiza el sistema completo con *n* procesadores y las que realizaría un solo procesador para realizar el mismo trabajo:

$$R = \frac{O(n)}{O(1)} \geq 1.$$

La redundancia mide el grado del incremento de la carga.

- **Utilización**: Refleja el grado de utilización del sistema completo:

$$U = R \cdot E \leq 1.$$

La utilización indica el porcentaje de recursos (procesadores, memoria, recursos, etc.) que se utilizan durante la ejecución de un programa paralelo

- **Calidad del sistema**:

$$Q = \frac{S \cdot E}{R}.$$

La calidad combina el efecto del speed-up, eficiencia y redundancia en una única expresión para indicar el mérito relativo de un cálculo paralelo sobre un sistema.

## *REALIZACIÓN PRÁCTICA*

Se va a evaluar el rendimiento del sistema a partir de la aplicación desarrollada en la práctica anterior. La multiplicación de matrices es un problema de orden cúbico que obliga a una elevada carga de cálculo en cuanto se incrementa un poco el tamaño de las matrices. La medida del rendimiento la vamos a realizar evolucionando en dos sentidos:

- Variando el volumen de cálculo.
- Jugando con el tamaño del sistema.

En lo que al volumen de cálculo se refiere, debemos escoger una serie de valores de tamaño para las matrices que nos resulten representativos. El punto de partida será un valor que provoque un tiempo de ejecución similar para una o varias máquinas en paralelo. Esto depende principalmente de la capacidad de las máquinas disponibles.

A partir de este punto de inicio, se irá incrementando el tamaño de las matrices y con cada nuevo valor se medirá el tiempo de cálculo sobre una máquina, dos, tres, etc. Una representación gráfica de los resultados nos debería llevar a la conclusión de que para un volumen de trabajo muy elevado, el

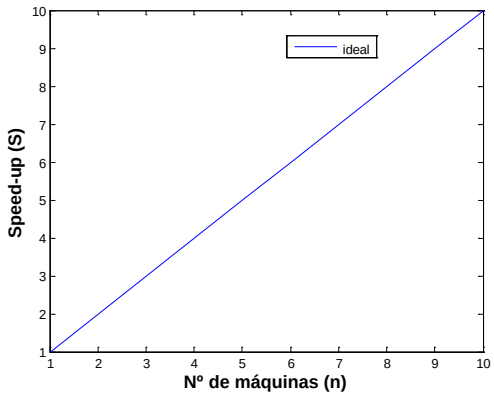
tiempo de proceso debería reducirse proporcionalmente al número de máquinas empleadas si éstas son de similar potencia.

## *CUESTIONES*

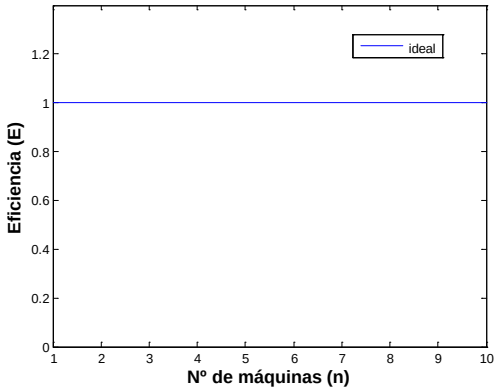
- ¿Está muy lejos el rendimiento obtenido del máximo teórico que se puede alcanzar?
- Evaluar el grado de acercamiento de la aplicación desarrollada a los máximos posibles, calculando para ello de forma aproximada la eficiencia, redundancia, utilización y la calidad del sistema.
- Estimar las causas de la desviación observada.
- Describir qué aspectos se deberían optimizar para obtener un mayor rendimiento.

### *Gráficas de rendimiento*

Representar gráficamente la evolución del speed-up (*S*) y de la eficiencia (*E*) frente al número de máquinas y compararla con la evolución ideal (figura a y figura b, respectivamente).




(a)




(b)

# Práctica 3: Introducción a la Programación Híbrida

## *OBJETIVOS*

- Entender las limitaciones de la programación monohilo.
- Entender los beneficios de la programación multihilo.
- Aprender cómo incrustar hilos que comparten memoria dentro de procesos de memoria distribuida.

## *CONCEPTOS TEÓRICOS*

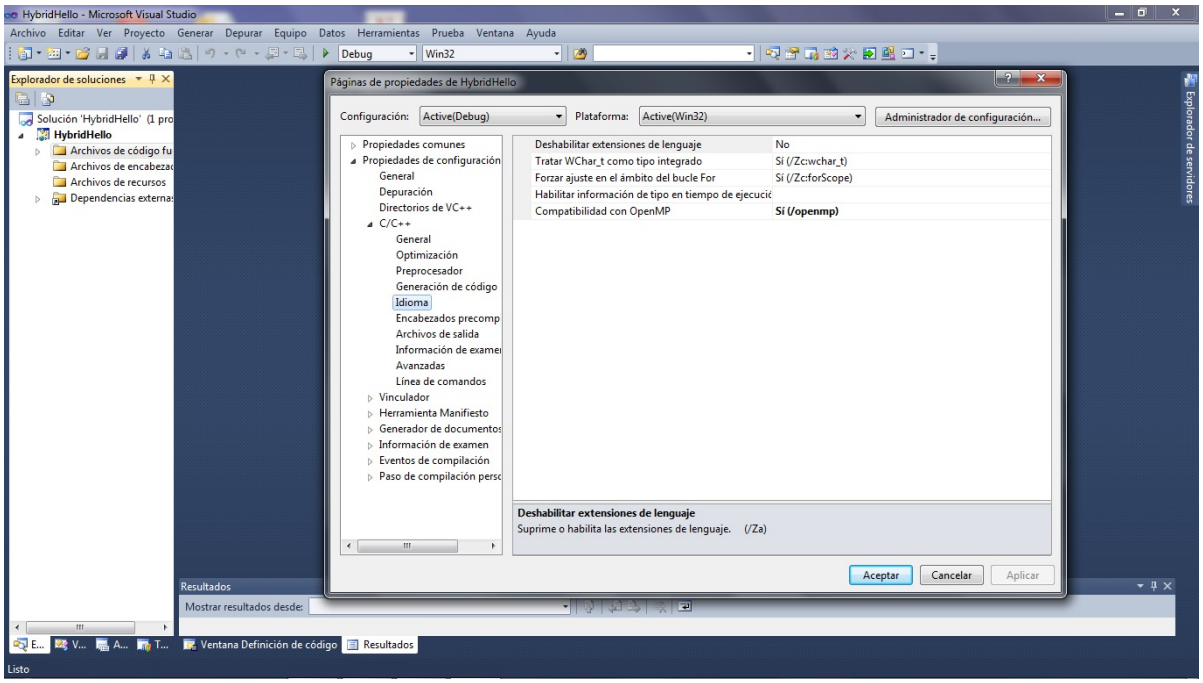
### *Limitaciones de la programación monohilo.*

Hasta ahora hemos desarrollado programas de un solo hilo. Aunque hubiera varios hilos lanzados, cada proceso contenía solamente uno. Esto plantea notables limitaciones cuando el programa ha de correr en sistemas multinúcleo, los más habituales en la actualidad. Si lanzamos tantos procesos como procesadores haya disponibles, solamente uno de los núcleos del procesador va a estar ocupado. Se puede incrementar la eficiencia simplemente lanzando tantos procesos como núcleos. Incluso así, un análisis más profundo del uso de los recursos probablemente revelará que se mantiene un cierto grado de ineficiencia.

En los entornos de paso de mensajes como MPI, la información se intercambia mediante mensajes que se envían de un proceso fuente a otro destino. Estos mensajes contienen, aparte de la información en sí misma, algunos elementos de información adicionales: fuente, destino, etiquete, cuenta, etc. Cuando los procesos fuente y destino se encuentran físicamente en procesadores distintos conectados a través de una red de área local, esto es lo correcto y prácticamente la única opción. Sin embargo, cuando los procesos se encuentran ubicados en el mismo procesador, no necesitan recurrir a un procedimiento tan complejo, ya que comparten una memoria común en la que ambos pueden leer y escribir. En este escenario, el uso de mensajes genera un considerable sobrecarga. Un sistema de memoria compartida ha de ser habilitado para optimizar el rendimiento.

### *Cómo incrustar hilos de memoria compartida en m procesos de paso de mensajes.*

Hasta ahora hemos desarrollado programas de un solo hilo. Cada proceso MPI es monohilo. Podemos crear múltiples hilos de diferentes maneras pero vamos a seleccionar el entorno de programación de OpenMP por resultar una manera muy simple de llevarlo a cabo. Para habilitar la programación multihilo se deben realizar algunos ajustes en las propiedades del proyecto:




Así el proyecto está preparado para aceptar directivas de OpenMP, pero será necesario comentar algo sobre su funcionamiento para poder seguir avanzando.

REGIONES PARALELAS EN OpenMP

Los programs OpenMP son monohilo por defecto. Disponen de un hilomaestro que, en ciertas ocasiones se divide en varios hilos para llevar a cabo un conjunto de operaciones en paralelo. Estos trozos de código son denominados “regiones paralelas”.

```
#pragma omp parallel
{
	Parallel region
}
```

La función omp_get_thread_num() devuelve la identificación del hilo actual de 0 a N-1 pero, ¿cómo se establece el valor de N? Hay también varias opciones, pero aquí vamos a utilizar únicamente la definición estática:

```
#pragma omp parallel num_threads (N)
{
	Parallel region
}
```

Hasta aquí correcto, pero aún nos resta establecer los contenidos de la región paralela. Puede estar integrada por cualquier conjunto de instrucciones váildas, si bien en la mayoría de los casos se van a emplear para paralelizar bucles. Los bucles “for” son los candidatos ideaales:

```
#pragma omp parallel num_threads (N)
{
	#pragma omp for
		for(i=0;i<n;i++){
			Operations to be performed
		}
```

```
}
```

Las “n” operaciones a realizar se distribuirán entre los N hilo. Se espera con ello lograr una reducción del tiempo de ejecución en procesadores multinúcleo y/o multihilo.

Esto es un entorno de variable compartida pero, ¿dónde están las variables compartidas? Las variables declaradas fuera de la región paralela son compartidas. Las variables declaradas dentro son privadas de cada hilo. De todas formas, es posible hacer que una variable compartida se convierta en privada:

```
#pragma omp parallel num_threads (N) private (j)
{
      #pragma omp for
            for(i=0;i<n;i++){
                  Operations to be performed on variable j
            }
}
```

En este caso cada hilo va a tener su propia copia de “j” aunque haya sido declarada fuera de la región pero, ¿cuál será el valor de “j” en cada hilo? En el ejemplo anterior “j” no ha sido inicializada, independientemente del valor que pueda tener antes de entrar en la región. Si se desea empezar con el valor que tenía previamente a la región paralela en cada hilo, se debe modificar el código:

```
#pragma omp parallel num_threads (N) firstprivate (j)
{
      #pragma omp for
            for(i=0;i<n;i++){
                  Operations to be performed on variable j
            }
}
```

De la misma forma, podemos necesitar que el hilo maestro tenga constancia de los cambios que ha sufrido “j” dentro de la región. Podemos conseguir que, al finalizar la región “j” tome el último valor que adquirió dentro:

```
#pragma omp parallel num_threads (N) firstprivate (j) lastprivate (j)
{
      #pragma omp for
            for(i=0;i<n;i++){
                  Operations to be performed on variable j
            }
}
```

Para concluir esta breve introducción vamos a estudiar una capacidad adicional de OpenMP. No será difícil de entender ya que existe una equivalente en MPI que ya ha sido utilizada. Se trata de la operación de reducción. Se aplica al caso de que una variable compartida es modificada hacia valores diferentes en los distintos hilos de ejecución. A veces, el valor final de esta variable tiene que ser obtenido como combinación de los valores generados por cada uno de los hilos. Vamos a ver un ejemplo sencillo:

```
#pragma omp parallel num_threads (N)
{
        #pragma omp for reduction(+:sum)
              for (i=0;i<n;i++){
```

```
                    sum=sum+(a[i]);
                }
}
```

Resulta obvio que lo que se pretende es obtener el valor final de la variable “sum” que deberá ser el resultado de las “n” sumas realizadas sobre ella. La reducción va a tomar el último valor generado por cada hilo y va a calcular la suma de todos ellos. Para que esto sea posible, se genera una copia privada de la variable en cada uno.

## *REALIZACIÓN PRÁCTICA*

Tomar de nuevo el programa de multiplicación de matrices. Y realizar los experimentos siguientes:

1. Lanzar dos procesos sobre matrices 5000 x 5000.
2. Hacer lo mismo con tantos procesos como núcleos por procesador estén disponibles.
3. Repetir la experiencia con más procesos que núcleos.
4. Ahora adaptar el programa al paradigma de programación híbrida lanzando dos procesos y dividiendo el proceso que calcula en tantos hilos como núcleos menos uno.
5. Volver a ejecutar el programa híbrido con tantos hilos como núcleos.

## *CUESTIONES*

- ¿Qué paradigma de programación proporciona el rendimiento más alto?
- ¿Es más óptimo ejecutar solo un proceso/hilo en cada núcleo o resulta que el proceso 0 debe compartir núcleo con otro?
- ¿Son estos resultados los esperados?¿Por qué?

# Práctica 4: Programación Híbrida

## OBJETIVOS

- Comprender algunas alternativas de planificación encaminadas a optimizar el tiempo de ejecución.

## CONCEPTOS TEÓRICOS

### *Sincronización.*

El procedimiento de sincronización por defecto introduce una barrera al final de las regiones paralelas de manera que la ejecución no continúa hasta que todos los hilos alcanzan ese punto. Esto es razonable pero en ciertos casos puede ser interesante evitar esta restricción. Se puede hacer así mediante la cláusula "nowait":

```
#pragma omp parallel num_threads (N)
{
      #pragma omp for nowait
            for(i=0;i<n;i++){
                  Operations to be performed on variable j
            }
}
```

En este ejemplo concreto, no plantea ninguna diferencia práctica pero en caso de que hubiera un nuevo bucle paralelo a continuación permitiría que los hilos pudieran entrar en él con la mayor rapidez.

### *Planificación.*

Hasta el momento se ha asumido que el trabajo a realizar se reparte entre los diferentes hilo participantes de manera equitativa. Esto es correcto pero, incluso siendo así puede haber diferentes posibilidades en el reparto que pueden tener consecuencias significativas en el rendimiento.

La política de planificación por defecto divide el número de iteraciones entre el número de hilos proporcionando a todos la misma cantidad de trabajo si es posible. Este reparto se realiza previamente a la ejecución; no se realizan cambios en tiempo de ejecución. OpenMP permite definir el tamaño de los bloques de trabajo (número de iteraciones en este caso); cada implementación de OpenMP decide cómo asignar estos bloques a cada hilo.

```
#pragma omp parallel num_threads (N)
{
      #pragma omp for schedule(static,10)

            for(i=0;i<n;i++){
                  Operations to be performed on variable j
            }
```

```
}
```

En este ejemplo se reparten bloques de 10 iteraciones. Los últimos bloques se hacen más pequeños si la división no es exacta.

Las políticas estáticas no permiten asignar trabajo dinámicamente a los hilos a medida que terminan el que ya tenían asignado. Esto supone una pérdida de eficiencia que debería ser evitada. Para ello es posible aplicar políticas dinámicas.

```
#pragma omp parallel num_threads (N)
{
      #pragma omp for schedule(dynamic,10)

            for(i=0;i<n;i++){
                  Operations to be performed on variable j
            }
}
```

En este ejemplo los hilos obtienen nuevos bloques tan pronto como finalizan el cálculo en curso.

## *EJERCICIO PRÁCTICO*

Se va a continuar el trabajo realizado en la práctica anterior. Disponemos ya de los resultados que proporciona la planificación estática, por lo que vamos a añadir resultados nuevos:

- Utilizar de nuevo la planificación estática pero especificando bloques de 10.
- Repetir el experimento utilizando bloques de 100.
- Cambiar ahora a planificación dinámica con bloques de 10.
- Utilizar de nuevo la planificación dinámica con bloques de 100.

Comparar todos los resultados para determinar cuál es la política más adecuada y tratar de explicar por qué. Trabajar con el número de hilos que demostraron ser la mejor opción en el caso anterior para ello.

REFERENCIAS:

OPENMP APPLICATION PROGRAM INTERFACE. Disponible en:
http://www.openmp.org/mp-documents/spec30.pdf

# Práctica 5: MPI vs OpenMP

## *OBJETIVOS*

- En programación híbrida se puede recurrir a múltiples combinaciones de número de procesos e hilos. Vamos a intentar descubrir cuál es la mejor.
- El paso de mensajes y la memoria compartida implican diferentes modelos de programación y un uso distinto de los recursos hardware. Debemos saber entones cuál es la técnica más eficiente y adecuada.

## *CONCEPTOS TEÓRICOS*

No se van a introducir nuevos conceptos teóricos en esta práctica.

## *EJERCICIO PRÁCTICO*

Vamos a lanzar una bacteria de pruebas para tratar de alcanzar el primero de los objetivos:

- Repetir la multiplicación de matrices 5000x5000 con dos procesos MPI en la máquina local.
- Lanzar tantos procesos MPI como núcleos haya disponibles en la máquina local.
- Lanzar tantos procesos como núcleos más uno.
- Volviendo a dos procesos, dividir el que trabaja (rango 1) en tantos hilos como núcleos disponibles menos uno.
- Dividir el rango uno ahora en tantos hilos como núcleos haya disponibles.

Comparar los resultados para ver cuál es la major política y explicar por qué.

Ahora vamos a abordar el Segundo objetivo Usamos la multiplicación 5000 x 5000 de nuevo:

- Lanzar tantos procesos como procesadores haya disponibles más uno y dividir el proceso que trabaja en tantos hilos como núcleos haya por procesador.
- Lanzar tantos procesos como núcleos haya disponibles en el cluster más uno (no usamos memoria compartida en este caso).

Comparar los resultados y tratar de explicarlos Consultar las referencias para encontrar respuestas.

REFERENCIAS:

Comparing the OpenMP, MPI, and Hybrid Programming Paradigms on an SMP Cluster
Gabriele Jost and Haoqiang Jin and Dieter An Mey and Ferhat F. Hatay
NAS Technical Report NAS-03-019, November 2003.

# Práctica 6: Envío de trabajos al cluster

## *OBJETIVOS*

- Conocer el proceso de envío de trabajos a un cluster de computación.
- Analizar las diferencias entre el trabajo a nivel local y la inclusión en un entorno de computación paralela.

## *CONCEPTOS TEÓRICOS*

Los trabajos que vamos a lanzar al cluster no se van a diferencia de los que hemos venido desarrollando hasta ahora. Se tratará de aplicaciones MPI derivadas de la multiplicación de matrices que estamos empleando como banco de pruebas habitual. Trabajaremos en Windows 8.1 empleando roles de usuario que previamente han sido dados de alta en el dominio ARAVAN y en el cluster de computación, de manera que se encuentran autorizados para lanzar trabajos. Asimismo vamos a utilizar la implementación MPI de Microsoft: MS-MPI.

La herramienta que nos permite lanzar estos trabajos se denomina “Job Manager” y forma parte de las utilidades de cliente que incluye el HPC PACK 2012 R2 que se ha instalado en las máquinas. Antes de ver cómo se envía un trabajo al cluster, es necesario tener en cuenta una serie de consideraciones que afectan a nuestros programas:

1. No vamos a disponer de interfaz gráfica. La información de inicialización se va a pasar a las aplicaciones en línea de comando; la información en tiempo de ejecución se ha de leer de fichero. Será necesario modificar el programa en cuiertos casos para que acepte y lea la información en línea de comando. En el caso de la multiplicación de matrices, el tamaño de éstas se va a suministrar de esta manera. Mediante una sentencia como: “size = atoi (argv[1]);” podemos introducir el tamaño de la matriz en línea de comando y recuperarlo para operar dentro del programa.
2. La salida estándar se va a redirigir a fichero, que tendremos que abrir y analizar una vez haya finalizado la aplicación.
3. Para que el gestor de trabajos no nos genere un error en su realización, deberemos finalizarlos con código de salida cero. Esto se logra mediante la función “exit (0)” que se encuentra definida en <stdlib.h>.

### *Generación de trabajos en Job Manger para la máquina local.*

Genéricamente un trabajo está formado por una serie de tareas, siendo éstas las aplicaciones que el usuario quiere ejecutar. En nuestro caso vamos a lanzar trabajos con una sola tarea: nuestra aplicación MPI. Para este caso particular existe la opción de crear trabajos con una tarea (“Single Task Job”).

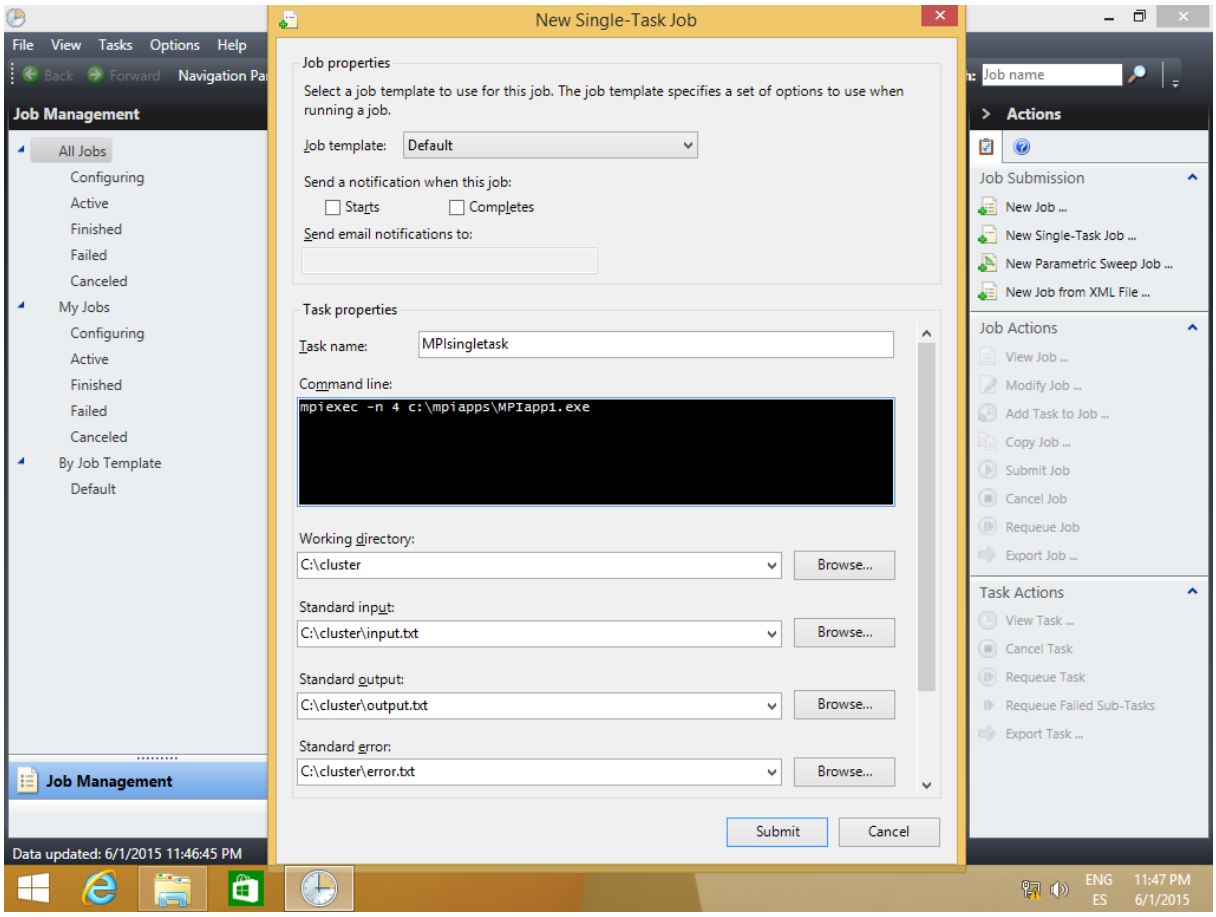



*Figura 6.1. Configuración de trabajos con una sola tarea.*

Vemos que se ha de configurar el directorio de trabajo por defecto que en nuestro caso coincide con la carpeta en la que vamos a ubicar los ficheros de texto asociados a la tarea. Configuramos cada uno de estos ficheros. Si no se van a emplear datos de entrada, no es necesario configurar la redirección de la entrada estándar.

En la línea de comando se define la tarea a llevar a cabo, que en este caso es una aplicación paralela MPI: "mpiexec –n 4 c:\mpiapps\MPIapp1.exe". La aplicación no tiene porqué encontrarse en el directorio de trabajo. El parámetro "-n 4" indica que queremos lanzar 4 procesos.

### *Trabajos de barrido paramétrico.*

En muchos ámbitos de trabajo las tareas no se realizan de forma aislada, sino que se llevan a cabo tareas relacionadas de las que se extraen resultados que se analizan de forma comparativa, complementaria, etc. Por ejemplo, nuestra multiplicación de matrices la ejecutamos sobre diferente número de nodos y con matrices de tamaños diferentes. Es posible lanzar un trabajo para cada caso, obviamente, pero es más eficiente configurar una batería de pruebas y lanzarla de una sola vez. Esto es lo que nos permite hacer la opción de gereración de trabajos de barrido paramétrico ("Parametric sweep job").

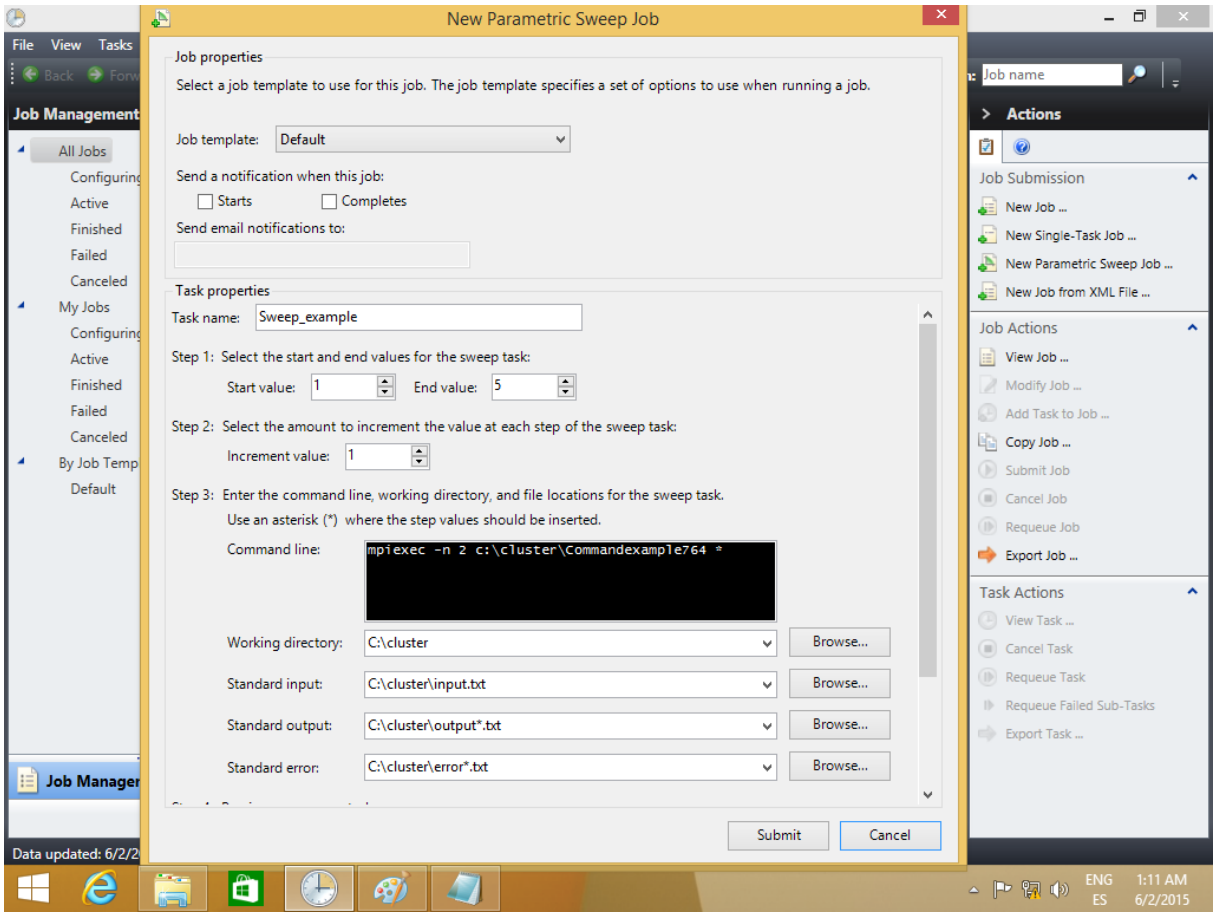



*Figura 6.2. Configuración de trabajos de barrido paramétrico.*

En el ejemplo hemos configurado que el parámetro va a variar de 1 a 5 en incrementos unitarios. Se realizarán, por lo tanto, 5 tareas, una para cada valor del parámetro. Es interesante observar que el asterisco que ubica el parámtro en la línea de comando, también se ha añadido a los nombres de los ficheros de salida. Esto permite que cada tarea imprima en su propio fichero sin sobreescribir la salida de otras.

En este caso hemos asociado el argumento introducido en línea de comando al parámetro de variación pero en realidad se puede asociar a cualquier aspecto. Por ejemplo, si quisiéramos modificar el número de procesos que ejecutan la tarea en cada caso, como hacemos en las pruebas de rendimiento sobre la multiplicación de matrices, podríamos escribir la siguiente línea de comando: “mpiexec –n * c:\ruta\multimatriz 5000”. También es posible emplear más de un asterisco pero no suele ser interesante porque no suele suceder que sean necesarios los mismos valores en dos posiciones diferentes de la línea de comando. Lo que no es posible es anidar barridos. En el caso que nos ocupa, podemos variar el número de procesos o el tamaño de las matrices pero no ambas cosas de manera anidada.

### *Generación de trabajos en Job Manger para el cluster.*

Conceptualmente no existe diferencia entre lanzar un trabajo en la máquina local y en el cluster, ya que una es parte del otro. Sin embargo hay algunas cuestiones de configuración que merece la pena destacar en este aprtado:

- Compartición de carpetas.
- Configuración de rutas.
- Selección de nodos.

Es necesario que la carpeta y subcarpetas en las que se va a trabajar seencuentren compartidas. La compartición la podemos realizar mediante el Explorador de Windows, editando las propiedades de la carpeta correspondiente al directorio de trabajo.

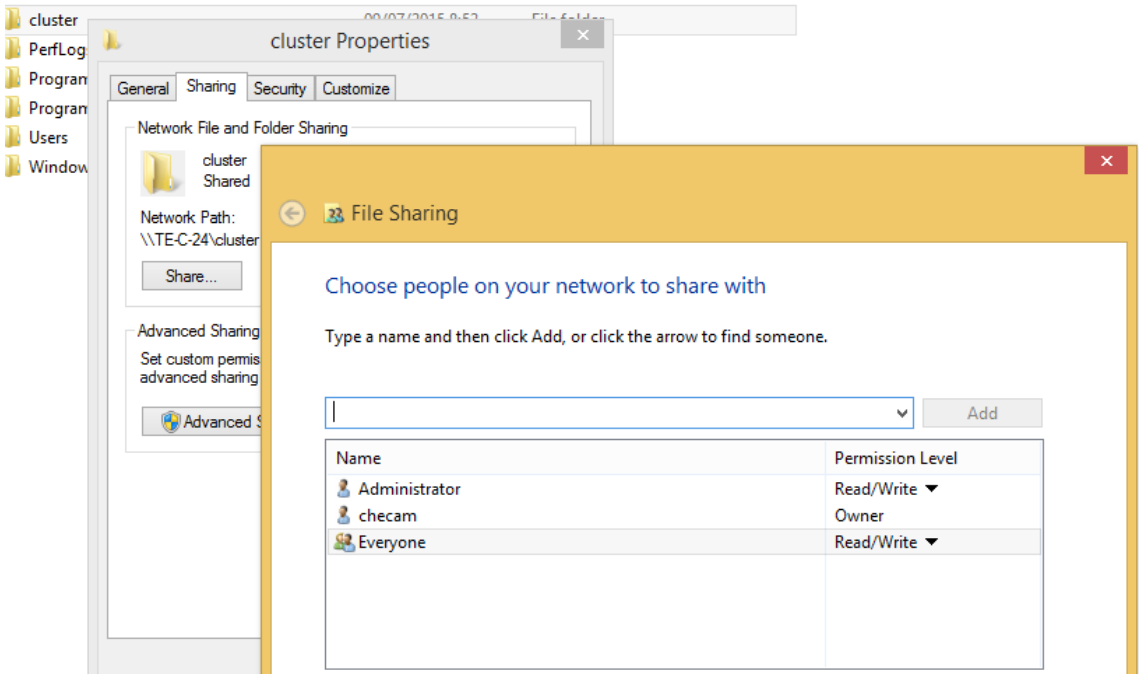



*Figura 6.3. Compartición del directorio de trabajo.*

Los usuarios que tengan que poder ejecutar el trabajo deberán disponer de los permisos correspondientes.

Si el trabajo ha de ser ejecutado por otros nodos, la ruta debe ser conocida por todos ellos bajo una denominación común. Para ello se utiliza el formato UNC (https://msdn.microsoft.com/en-us/library/gg465305.aspx). Bajo este formato se configura la ruta del directorio de trabajo; el resto de rutas: ficheros de entrada y salida y aplicación a ejecutar en la tarea, se consideran referidos a dicha ruta. La figura 6.4 muestra cómo llevar a cabo esta configuración.

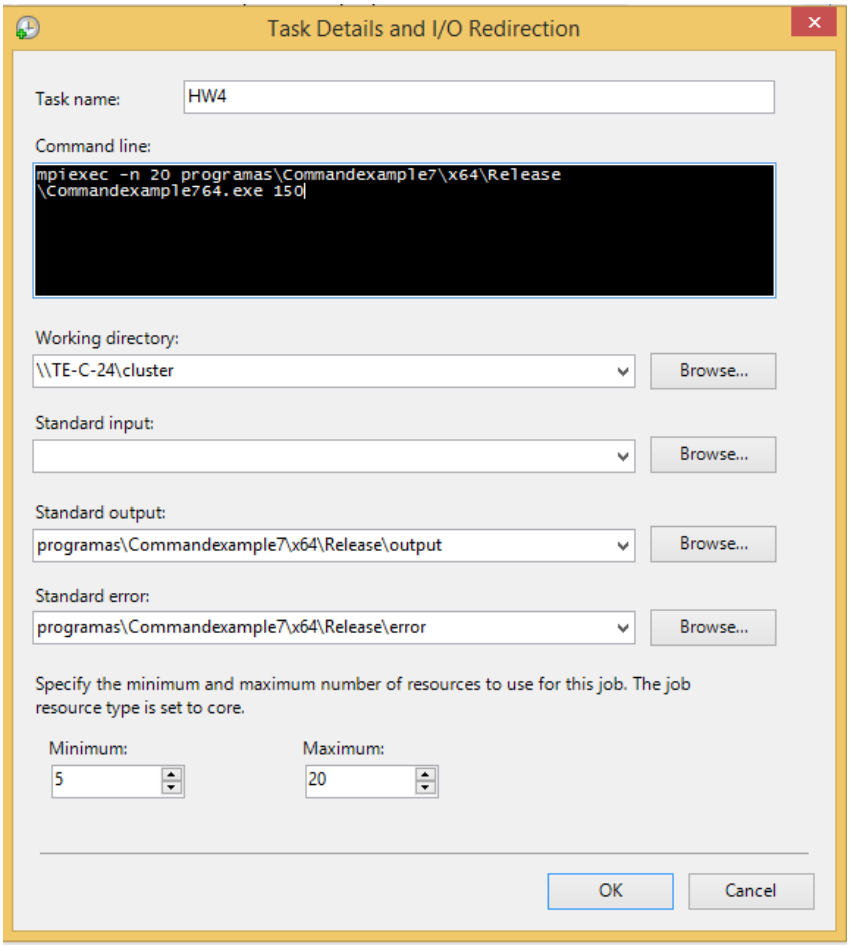



*Figura 6.4. Configuración del directorio de trabajo compartido.*

En este caso, la ruta completa de la aplicación sería:

*TE-C-24\c:\cluster\programas\Commandexample7\x64\Release\Commandexample764.exe*, siendo "150" un argumento en línea de comando de la aplicación.

Finalmente, se ha de realizar la selección de los nodos en los que se pretende correr la aplicación. Obviamente, se trata de un máximo, ya que si la aplicación no crea suficientes procesos/hilos de ejecución para llenarlos, no se van a utilizar algunos.

En la configuración del trabajo, acudiremos a la opción "Resource Selection" para definir los nodos que queremos habilitar.

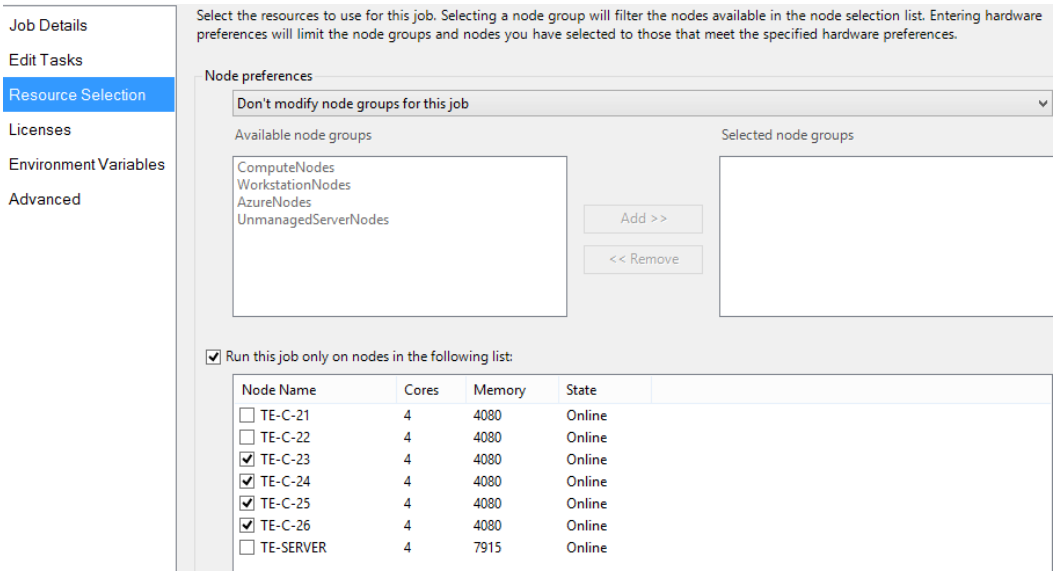



*Figura 6.5. Selección de nodos.*

## *EJERCICIO PRÁCTICO*

Lanzaremos en primer lugar la clásica aplicación HolaMundo de manera que cada proceso imprima en la salida estándar el mensaje de saludo, su rango y el total de procesos lanzados.

Sobre la aplicación de multiplicación de matrices, una vez adaptada para que el tamaño sea introducido en línea de comando, vamos a lanzar 2 trabajos paramétricos: uno variando el número de procesos de 1 a 8 y otro, con 8 procesos variando el tamaño de la matriz de 3000 a 5000 en incrementos de 1000.

- Incluir en el informe las salidas obtenidas en cada caso.
- Reflejar además los recursos asignados por el sistema para la ejecución de las mismas. Esto se puede obtener analizando la información asociada a la tarea una vez finalizada ésta.
- Comprobar que el tiempo que ha tardado la tarea en ejecutarse es coherente con el resultado de tiempo proporcionado en el archivo de salida.

# Práctica 7: Planificación de trabajos

## *OBJETIVOS*

- Conocer las políticas de planificación disponibles.
- Analizar su repercusión en el rendimiento global del sistema.

## *CONCEPTOS TEÓRICOS*

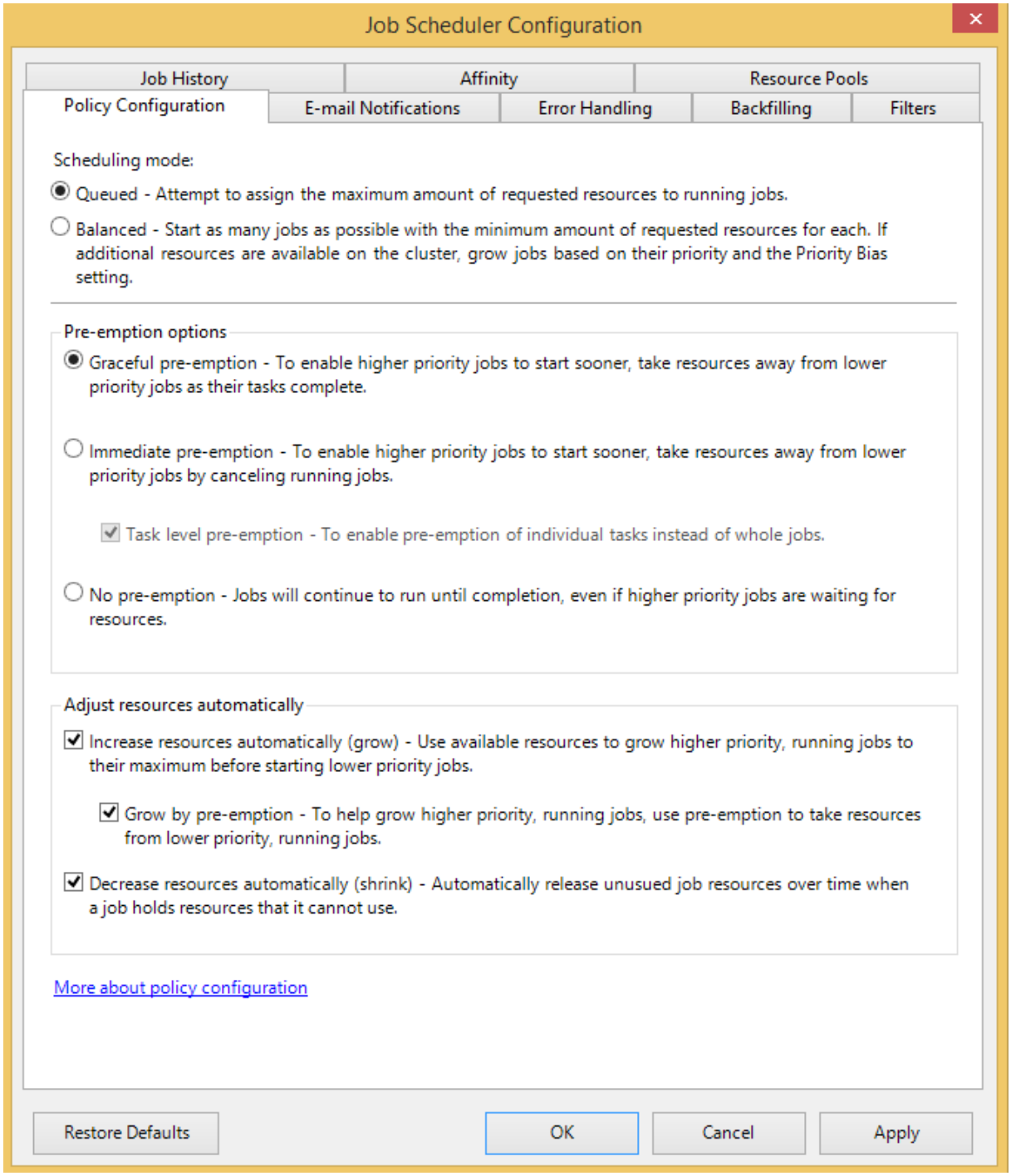
El administrador del Sistema se encarga de configurar políticas de planificación eficientes. Su objetivo es optimizer el rendimiento del Sistema, lo cual no es un concepto obvio De hecho existen diferentes formas de interpretar el rendimiento, lo cual da lugar a objetivos diferentes también:

- Maximizar la utilización del sistema.
- Minimizar el tiempo de ejecución de los trabajos (wall time).
- Maximizar el throughput (trabajos realizados por unidad de tiempo).

Desde el punto de vista del usuario, el "wal time" es lo que suele se más importante pero el administrador no va a ocuparse de un único usuario priviliegiado, sino que tratará de obtener el máximo partido del conjunto del sistema.

El planificador de trabajos de Windows HPC proporciona diversas opciones de planificación. Lo primero que se ha de decider es si se quiere seguir una política encolada o balanceada en la planificación de los trabajos:

- Encolada (queued): el planificador va a intentar otorgar el máximo de los recursos solicitados por los trabajos y sus tareas. Cuando todos los recursos estén agotados, los trabajos siguientes tendrán que esperar en la cola. Como se muestra en la figura, varias sub-opciones acompañan a esta decisión.




*Figua 7.1.Política de planificación balanceada.*

Una de ellas es cómo manejar la precedencia. Si seleccionamos la opción elegante (graceful), los trabajos más prioritarios van a tomar recursos de otros, pero sólo cuando sus tareas vayan finalizando. La precedencia inmediata permite cancelar trabajos menos prioritarios en curso para dar servicio a otros más prioritarios. También es posible deshabilitar la precedencia. Se puede automatizar la Gestión de los recursos. Así, los trabajos más prioritarios recibirían más recursos, de acuerdo con la política de predecencia adoptada, hasta alcanzar el máximo solicitado, antes de que trabajos menos prioritarios puedan comenzar. Podrían incluso crecer tomando recursos de trabajos de menor prioridad que ya

estén en curso. Finalmente, se puede establecer que se retiren recursos a trabajos que ya no vayan a tener opción de utilizarlos.

- Balanceada (balanced): el planificador intentará arrancar tantos trabajos como sea posible. Para ello reducirá el número de recursos asignados a cada uno sin sobrepasar el mínimo solicitado por el usuario. De esta forma, siempre que haya recursos disponibles, se lanzarán nuevos trabajos en lugar de incrementar los recursos asignados a trabajos ya en curso. Como en el caso anterior, aparecen una serie de sub-opciones.

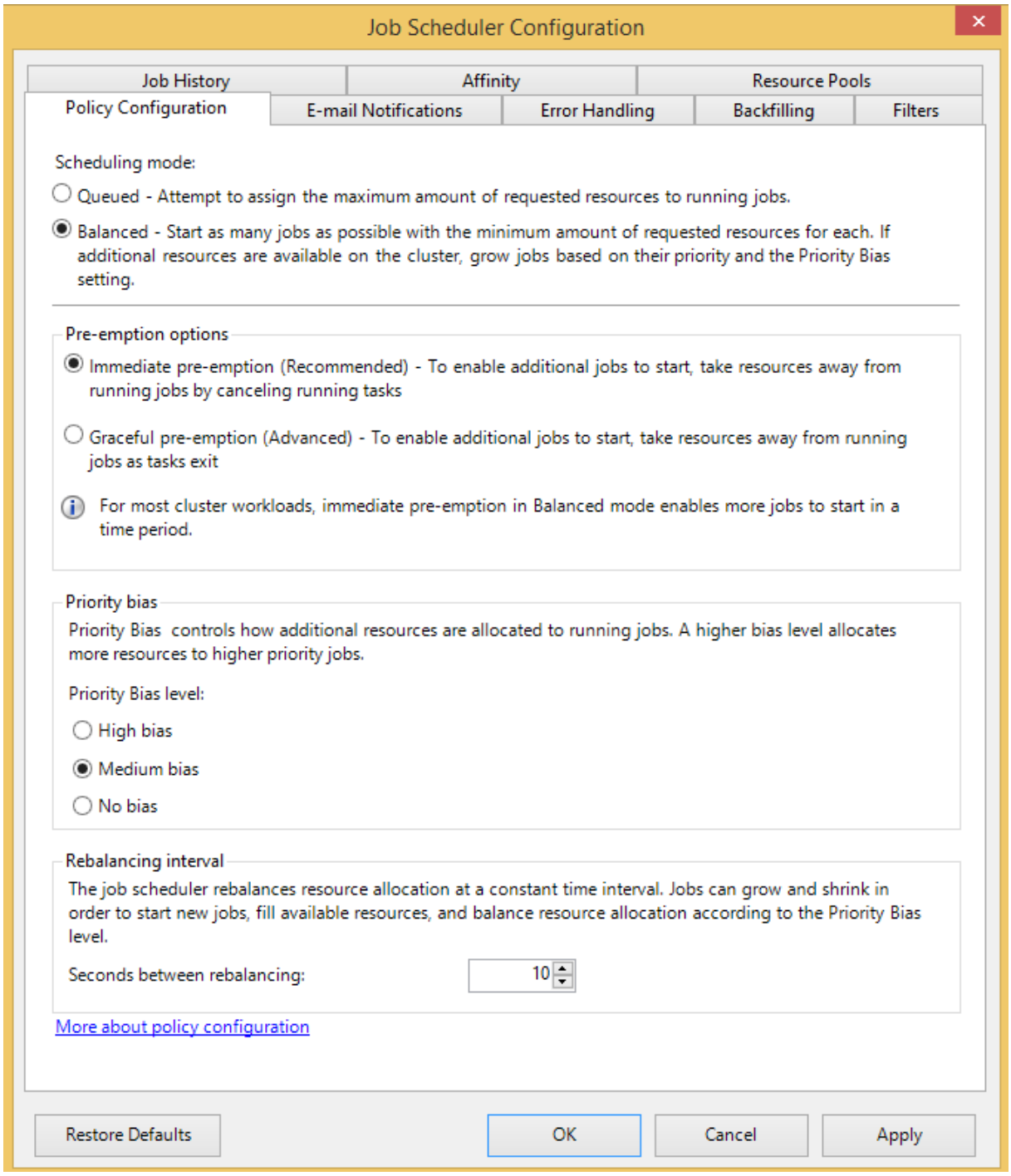



*Figure 7.2. Balanced scheduling policy.*

De Nuevo, la precedencia se puede seleccionar como elegante o inmediata, pero no se puede deshabilitar. Cuando se trata de asignar recursos adicionales a trabajos en curso, la decision

de cómo hacerlo está condicionada por la prioridad de los mismos. La influencia de la prioridad en la decision es ajustable. También es possible ajustar el interval de rebalanceo de recursos.

El usuario asigna la prioridad a cada trabajo en el momento de su configuración. Dentro de un mismo trabajo todas las tareas comparten el mismo nivel de prioridad. En esta práctica vamos a trabajar con trabajos únicos, por lo que ignoraremos todo lo que tiene que ver con la prioridad por el momento.

## *EJERCICIO PRÁCTICO*

Vamos a configurar un trabajo integrado por multiples tareas. Cada tarea consistirá en la ejecución de una multiplicación de matrices 8000 x 8000. El número de procesos se incrementará desde 4 hasta el número de núcleos disponibles en el Sistema, en increment de 4 en 4.

El trabajo se lanzará bajo una política encolada y bajo una política balanceada. En ambos casos se lanzará ordenando las tareas por número de procesos, de menor a mayor y de mayor a menor.

Para cada caso, se mostrará en una table los recursos asignados a cada tarea, su tiempo de ejecución y el tiempo de ejecución de todo el trabajo.

¿Qué politica de planificación resulta ser major para este tipo de carga? Intentar explicar por qué a partir de los contenidos de las tablas.

# Práctica 8: Planificación de trabajos II

## *OBJETIVOS*

- Comprender la diferencia entre tareas y trabajos.
- Trabajar con difierentes niveles de prioridad y distintas políticas de planificación.

## *CONCEPTOS TEÓRICOS*

En esta práctica vamos a trbajar con niveles de prioridad distintos, por lo que aparecerán disponibles nuevas opciones de configuración. Los niveles de prioridad se pueden asignar a los trabajos, ya sea porque el usuario los considera de distinta importancia o para buscar una mayor eficiencia en su ejecución. En esta práctica la motivación va a ser la segunda. Los niveles de prioridad asignables a cada trabajo son:

- Muy alta (Highest).
- Alta (Above normal).
- Normal.
- Baja (Below normal).
- Muy baja (Lowest).

En este momento aparece una nueva cuestión: es posible que se exceda el máximo número de conexiones permitidas a la carpeta compartida (hasta 20). La solución se encuentra en el uso de carpetas compartidas en el servidor, cuyo sistema opertivo permite un número de conexiones prácticamente ilimitado. Para ello se debe ajustar la carpeta compartida proporcionada por el admisitrador como “directorio de trabajo” y copiar el ejecutable de la aplicación a dicha carpeta.

## *EJERCICIO PRÁCTICO*

En primer lugar vamos a repetir los experimentos realizados en la práctica anterior pero lanzando multiples trabajos de una sola tarea en lugar de un solo trabajo con multiples tareas. Por el momento mantendremos el nivel de prioridad normal que adquieren por defecto. Reconstruir las tablas de resultados de la práctica anterior con los nuevos datos y comparar ambas.

¿Cómo son los tiempos de ejecución respecto al caso anterior? ¿Qué otras circunstancias se producen ahora? ¿Cómo se pueden superar?

Manteniendo los ajustes de planificación por defecto, modificar los niveles de prioridad de los trabajos a los que se considere más adecuados y repetir el procedmiento anterior.

¿Se ha conseguido mejorar el rendimiento? ¿Por qué puede ser?

# Práctica 9: Planifiación de trabjos III

## *OBJETIVOS*

- Comprender el concepto de precedencia.
- Comprobar la influencia de la precedencia en el rendimiento del sistema.

## *CONCEPTOS TEÓRICOS*

La precedencia permite que trabajos más prioritarios interrumpan a otros menos. Como ya se ha comentado, esto se puede llevar a cabo de diferentes formas. Dado que el objetivo es el rendimiento del sistema, la prioridad se asignará de manera que se minimice el tiempo de ejecución global.

## *EJERCICIO PRÁCTICO*

Agrupar las tareas lanzadas en las practices anteriores en 2 trabajos. En uno de ellos se situarán las tareas que demandan menos recursos y se le dará mayor prioridad. En el otro trabajo, con prioridad menor, se encontrarán el resto de las tareas.

De Nuevo bajo las políticas encolada y balanceada, repetir los experimentos habituales variando las políticas de precedencia.

Construir de Nuevo las tablas de resultados y compararlas.

Decidir qué política de precedencia es la más aconsejable para este tipo de carga de trabajo.

Comparar los resultados obtenidos mediante la política de planificación encolada con las opciones “Adjust resources automatically” activadas y sin activar.

Comparar los resultados obtenidos mediante la política de precedencia balanceada variando la influencia de la prioridad en la precedencia.

# Práctica 10: Batalla de rendimiento.

## *OBJETIVOS*

- Tomas decisiones de planificación acertadas.

## *CONCEPTOS TEÓRICOS*

No se van a introducer nuevos conceptos teóricos en esta práctica.

## *EJERCICIO PRÁCTICO*

Para una determinada aplicación de multiplicación de matrices (la misma para todos los participantes), cada un ova a optar por las opciones de planificación que crea le van a dar mejores resultados. Esto incluye el uso de trabajos, tareas o ambos. Con estas opciones se van a llevar a cabo los experimientos habituales y se van a comparar los tiempos de ejecución totales para detereminar cuáles, de las planteadas, eran las mejores opciones.

En el informe se ha de incluir las decisions tomadas, los resultados obtenidos y la comparación de los mismos con los mejores. Explicar en su caso porqué no se han obtenido los mejores resultados. En caso de haberlos obtenido, enhorabuena, no va a ser necesario incluir estas explicaciones en el informe.

# Apéndice A: Instalación de DeinoMPI

DeinoMPI es una implementación del estándar MPI-2 para Microsoft Windows derivado de la distribución MPICH2 de Argonne Nacional Laboratory.

Requisitos del sistema:

- Windows 2000/XP/Server 2003/Windows 7
- .NET Framework 2.0

## *Instalación*

Para instalar DeinoMPI hay que descargarlo e instalarlo en todas las máquinas del cluster, la instalación es idéntica en todas las máquinas. Es necesario ser administrador para instalar DeinoMPI, pero una vez instalado cualquier usuario puede usar DeinoMPI aunque no tenga privilegios de administrador.

Después de instalar la aplicación es necesario añadir al path la ruta hasta el directorio \bin de la instalación de DeinoMPI.

**Nota:** asegurarse de que la versión de Deino coincide con la del sistema operativo (32 o 64 bits).

## *Configuración*

Después de instalar el software en todas las máquinas cada usuario necesitará crear un "Credential Store". Este registro se utilizará para lanzar rutinas de forma segura. Mpiexec no ejecutará ninguna rutina sin un "Credential Store". El entorno gráfico mostrará al usuario la opción para crear un "Credential Store" en la primera ejecución. Se puede crear este "Credential Store" en el entorno gráfico o mediante línea de comandos:

```
create_credential_store.exe
```

Por defecto todos los usuarios tienen acceso a todas las máquinas en las que se ha instalado DeinoMPI y pueden lanzar trabajos en todas las máquinas en las que tienen cuentas de usuario. Si se quiere restringir el acceso a las máquinas se puede utilizar:

```
manage_public_keys.exe
```

En línea de comandos. El administrador debe ejecutar los siguientes comandos en cada máquina a la que quiera restringir el acceso:

- `manage_public_keys.exe /auto_keys false`
- `manage_public_keys.exe /clear_all`

En este momento ningún usuario tiene acceso a las máquinas. El administrador puede añadir usuarios uno a uno utilizando la "public key" del "Credential Store" de cada usuario. Se puede exportar la "public key" del "Credential Store" de cada usuario utilizando el entorno gráfico o mediante línea de comandos, por ejemplo:

```
manage_public_keys.exe /export mypubkey.txt
```

El administrador puede añadir acceso al usuario mediante el siguiente comando:

```
manage_public_keys.exe /import mypubkey.txt
```

Este comando es local a cada máquina y lo debe ejecutar un usuario con privilegios de administrador. Como el comando es local se debe ejecutar en cada máquina del cluster a la que se quiere que el usuario tenga acceso. Una vez que la "public key" del usuario se ha importado en una o más máquinas del cluster, el usuario podrá ejecutar trabajos en estas máquinas usando el entorno gráfico o mediante el comando:

```
mpiexec
```

## *Ejecutar Aplicaciones*

Se puede utilizar el entorno gráfico o la línea de comandos para lanzar trabajos MPI.

## *Entorno Gráfico*

Esta herramienta puede ser utilizada para arrancar procesos MPI, administrar las "Credential Store" de los usuarios, buscar las máquinas con DeinoMPI instalado en la red local, verificar las entradas mpiexec para diagnosticar problemas comunes, y ver la web de DeinoMPI para consultar información como páginas de ayuda y todas las funciones MPI.

### Pestaña "Mpiexec"

Es la página principal, se utiliza para arrancar e interactuar con los procesos MPI.

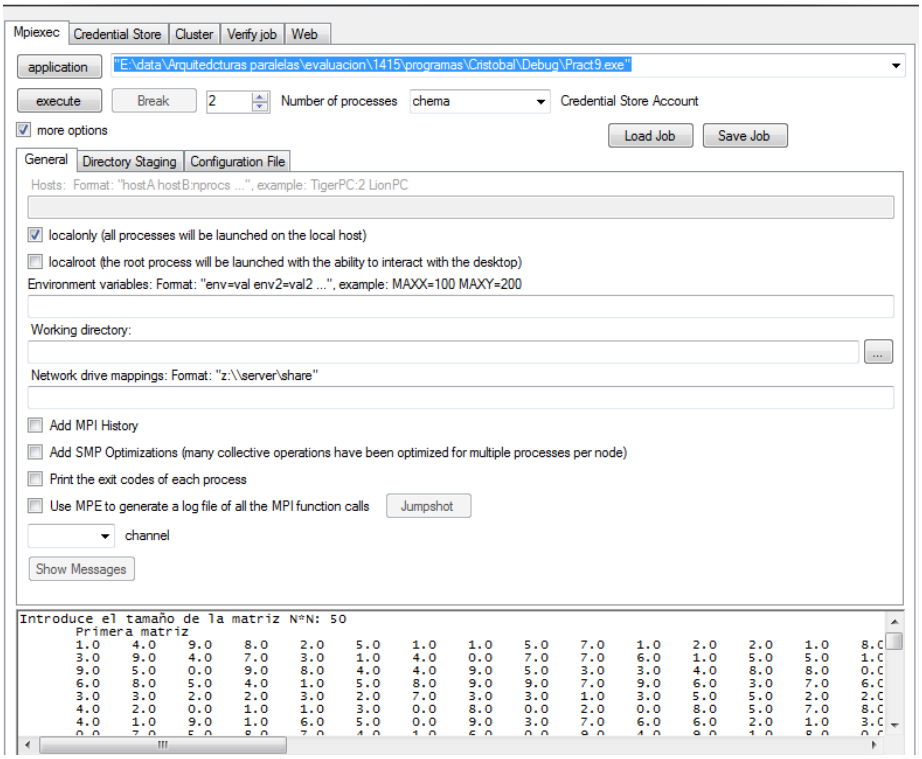



*Figura A1. Pestaña "Mpiexec".*

Los principales elementos de esta pestaña son:

- **Botón y cuadro de texto "application":**
  - Introduce aquí la ruta completa de la aplicación MPI. La ruta debe ser la misma para todas las máquinas del trabajo. Previamente la aplicación tiene que encontrarse en ellas.
  - Si se introduce una ruta de red se debe estar seguro de tener suficientes privilegios en el servidor para albergar el ejecutable.
  - Se puede pulsar sobre el botón "application" para buscar la localización del ejecutable.
- **Botón "execute":** al pulsar aquí se ejecutará el programa.
- **Botón "Break":** permite abortar un trabajo en curso.
- **"Number of processes":** permite especificar el número de procesos que se quieren lanzar independientemente del número de máquinas en las que se vayan a ejecutar.
- **Check box "more options":** Esta opción expande o contrae el area de controles extra.
- **Cuadro de texto "Hosts":** permite especificar en qué máquinas se quiere ejecutar el trabajo. Para lanzarlo en la máquina local solamente podemos poner su nombre de hosto o simplemente mantener la opción por defecto "localonly". Los nombres de las máquinas se separan simplemente por espacios.

### Pestaña "Credential Store"

Esta pestaña se utiliza para administrar los "Credential Store" de los usuarios.

Si no se ha creado un "Credential Store" hay que seleccionar el check box "enable create store options" para que las demás opciones estén disponibles. Estas opciones están ocultas porque normalmente solo son necesarias la primera vez que se ejecuta DeinoMPI.

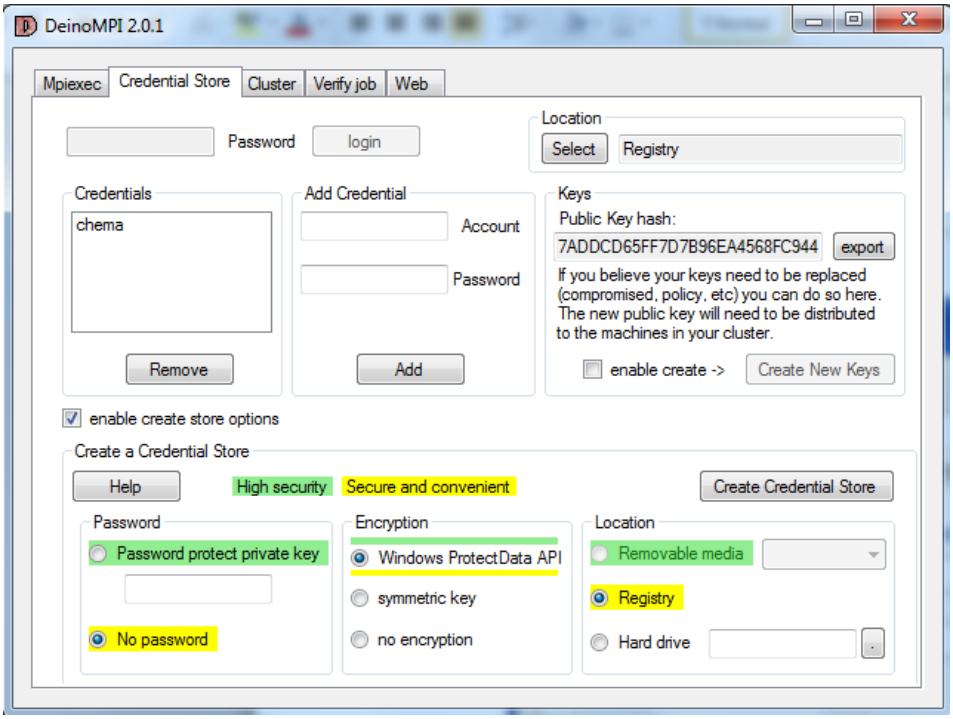



*Figura A2. Pestaña "Credential Store" con todas las opciones disponibles.*

Para crear un "Credential Store" hay que seleccionar el check box "enable create store options" para poder acceder a los campos de creación. Hay tres secciones:

- **"Password":**
  - Si se ha seleccionado una contraseña el "Credential Store" sólo es accesible introduciéndola, ésta es la opción más segura pero es necesario introducir la contraseña cada vez que se lanza una rutina MPI.
  - Si se selecciona "No password", utilizar MPI es más fácil pero más vulnerable, sin contraseña cualquier programa ejecutado por el usuario puede acceder al "Credential Store", lo que normalmente no supone ningún problema si se sabe que no se está ejecutando un software malicioso.
  - Aunque no se ponga contraseña el "Credential Store" no estará disponible para otros usuarios si se selecciona la opción de encriptación.
- **"Encryption":**
  - "Windows ProtectData API" sirve para encriptar el "Credential Store" utilizando el esquema de encriptación del usuario actual proporcionado por Windows. Esto asegura que el "Credential Store" solo podrá ser accesible cuando el usuario esté validado.
  - Si se selecciona una contraseña se podrá seleccionar el formato de encriptación "symmetric key", esta encriptación utiliza la contraseña para crear una clave simétrica para encriptar el "Credential Store". Esta encriptación no es específica para el usuario, por lo que podrá acceder al registro cualquier usuario que conozca la contraseña.

- No se recomienda la opción “no encryption” ya que almacena los datos del “Credential Store” en un archivo de texto plano accesible a todo el mundo. Es responsabilidad del usuario proteger el fichero del “Credential Store”.

- **“Location”:**

  - Seleccionar la opción “Removable media” para salvar el registro en un dispositivo extraíble como una memoria USB. Si se selecciona esta opción las rutinas MPI sólo se podrán arrancar por el usuario actual cuando su dispositivo esté conectado a la máquina. Esto puede ser una mejora de seguridad porque el usuario controla cuando el “Credential Store” está diponible. Si se combina con una contraseña y encriptación el dispositivo estará seguro en caso de pérdida.

  - Seleccionar la opción “Registry” para salvar el “Credential Store” en el registro de Windows.

  - Se puede usar la opción “Hard drive” para salvar el “Credential Store” en el disco duro de la máquina. El botón de búsqueda permite navegar gráficamente hasta el directorio deseado, incluyendo dispositivos extraíbles.

### *Pestaña “Cluster”*

En esta pestaña se muestran las máquinas de la red local y se puede ver la versión de DeinoMPI que tienen instalada.

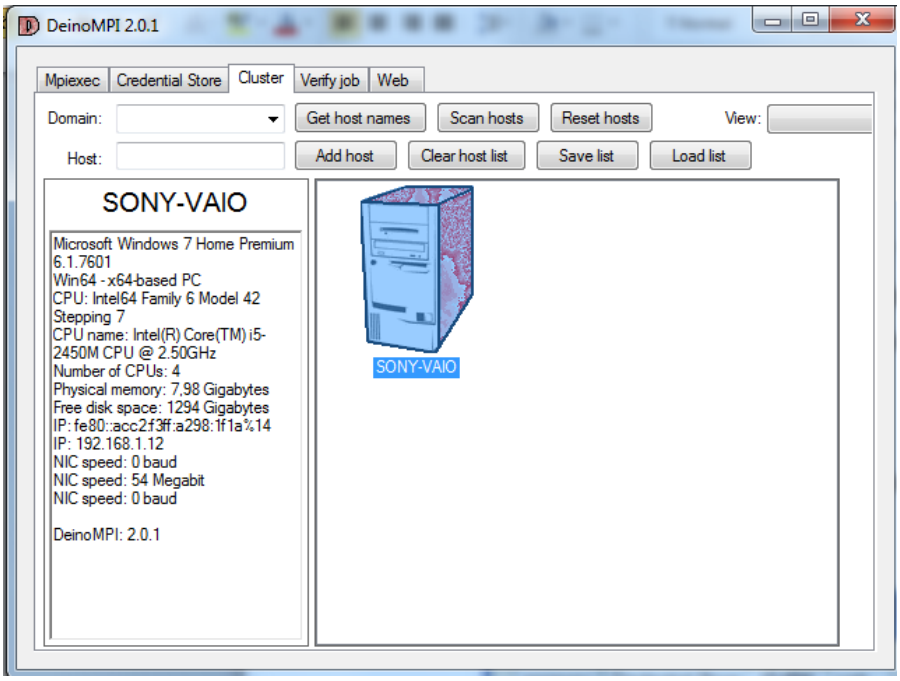



*Figura A3. Pestaña “Cluster” – Vista Iconos Grandes.*

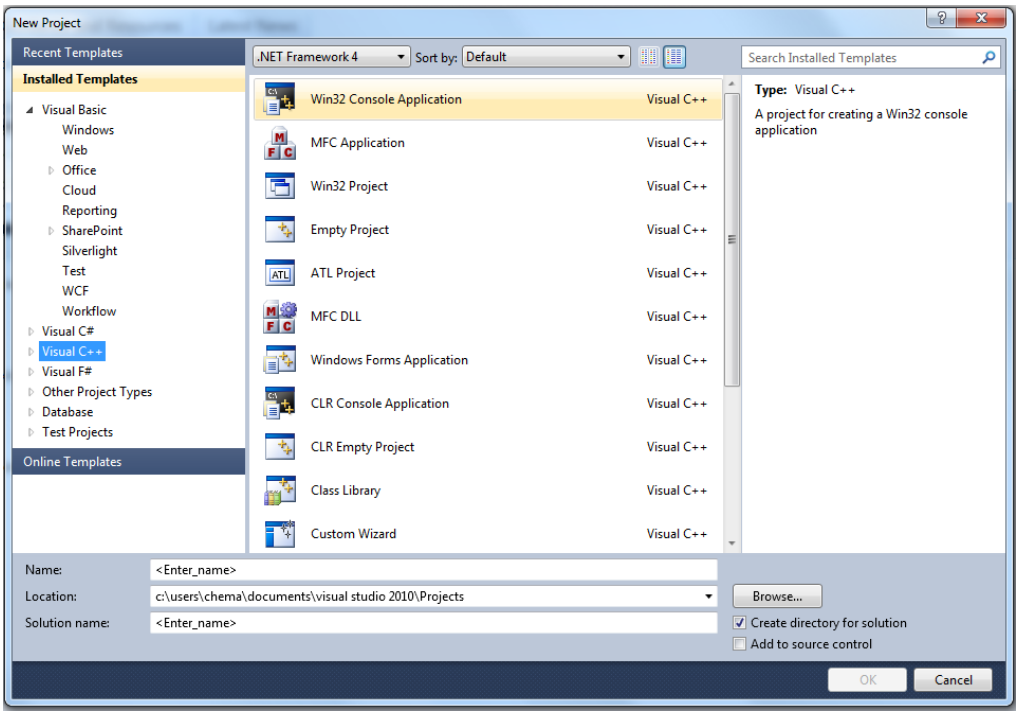
Podemos añadir nuevas máquinas a mano o hacer que las busque dentro del dominio especificado.

Se puede obtener más información en la web de Deino: http://mpi.deino.net/.

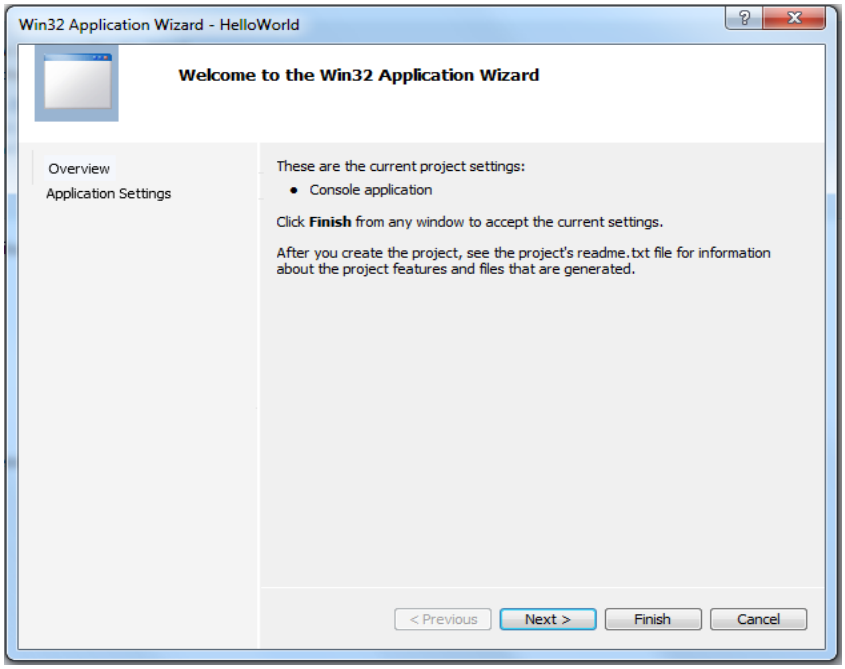
# Apéndice B: Configuración de un Proyecto en Visual Studio 2010 y versiones sucesivas.

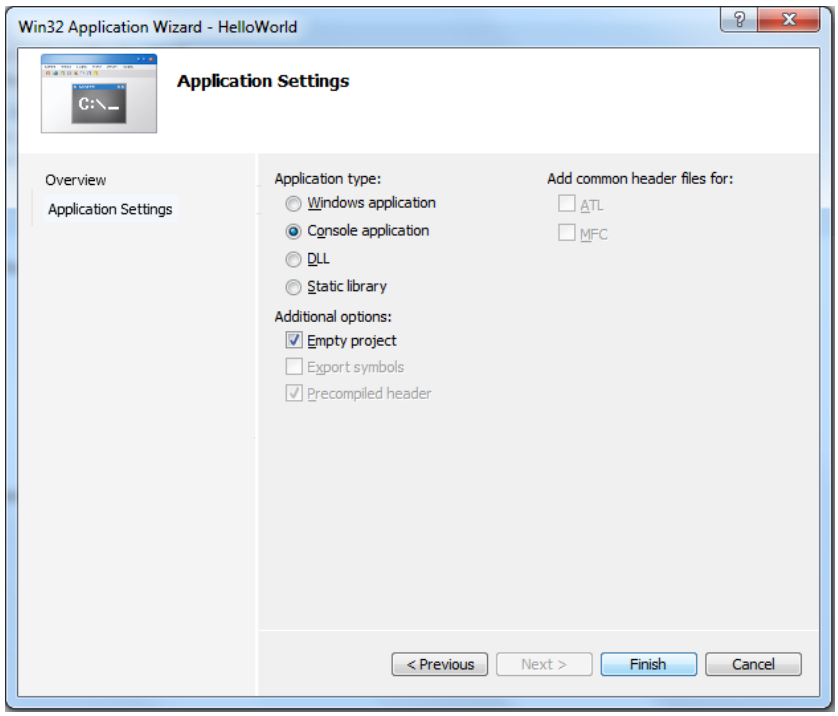
En esta sección vamos preparar un proyecto que permita construir una aplicación MPI dentro del nuevo entorno de trabajo proporcionado por Microsoft Visual Studio 2010.

- Crear un **nuevo proyecto y solución**. Se les puede dar a ambos el mismo nombre:

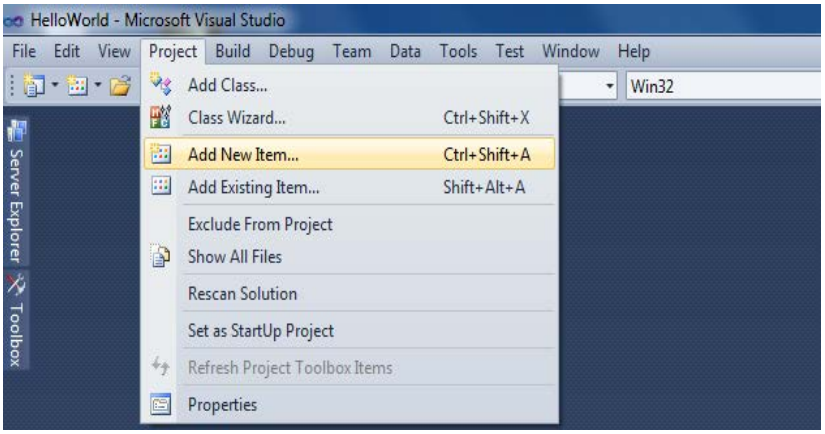


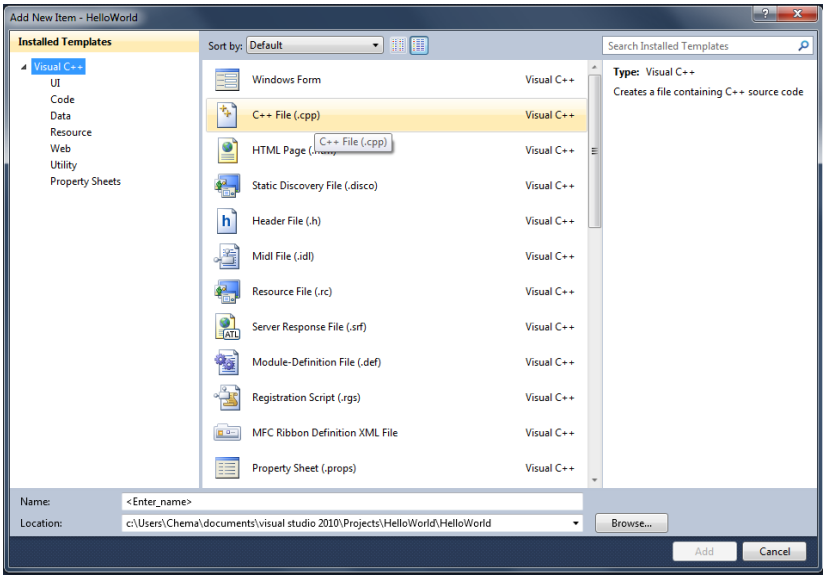

- Configurarlo como **proyecto vacío** ("*Empty project*"):



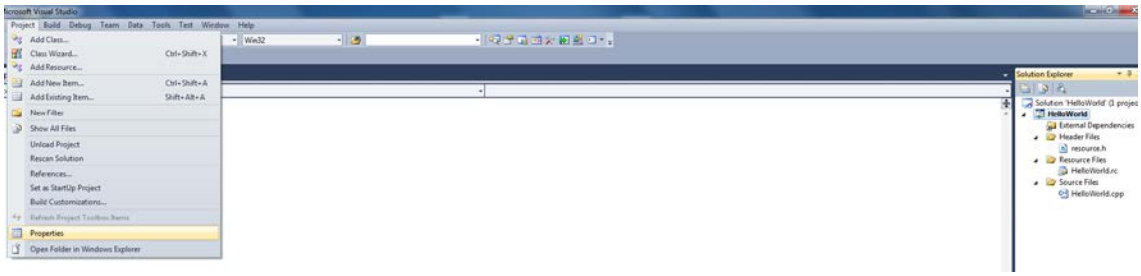




- Una vez creado el proyecto y la solución, añadir un fichero de código como **nuevo elemento** (“*Add New Item*”):

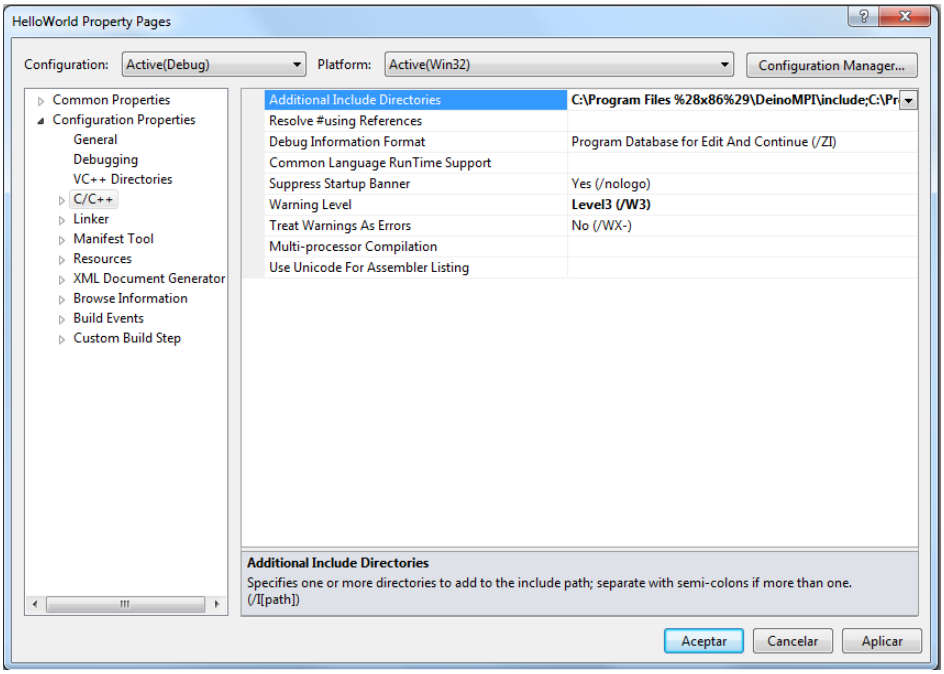


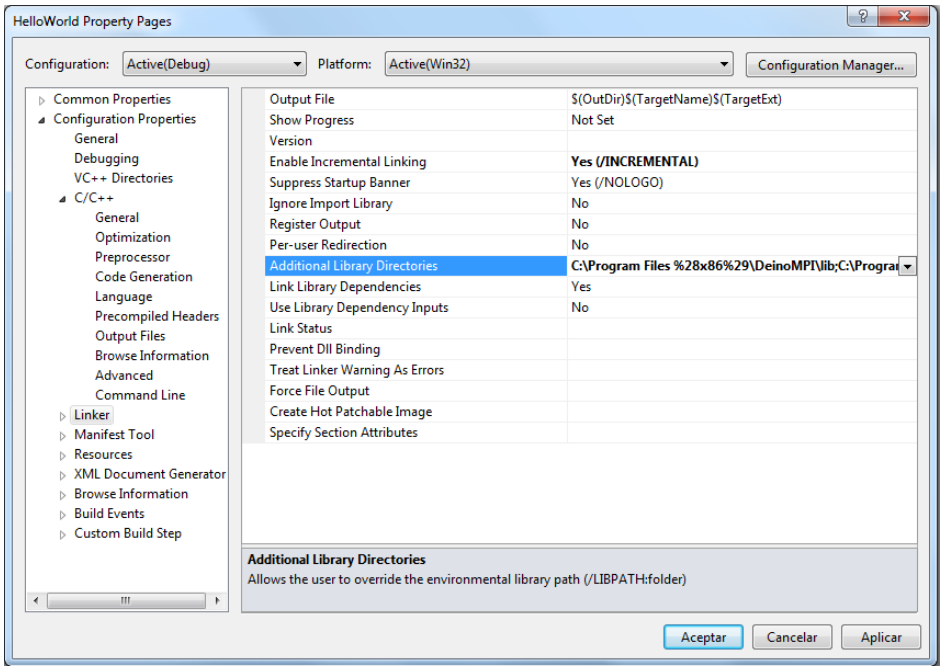

- A continuación, y no antes, se van a ajustar las **propiedades del proyecto** (“*Properties*”):



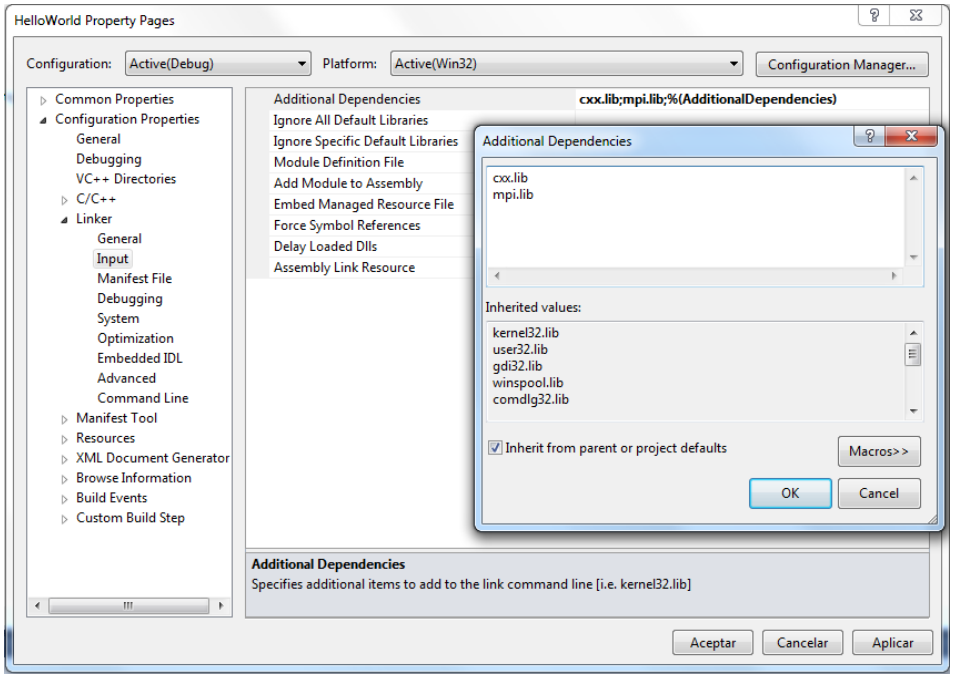
1. En la **sección C/C++** debemos introducir la ruta de la carpeta donde se encuentran los ficheros de cabecera de MPI (“*Additional Include Directories*”). Por defecto colgará de *\Archivos de Programa (x86)\DeinoMPI\include*:



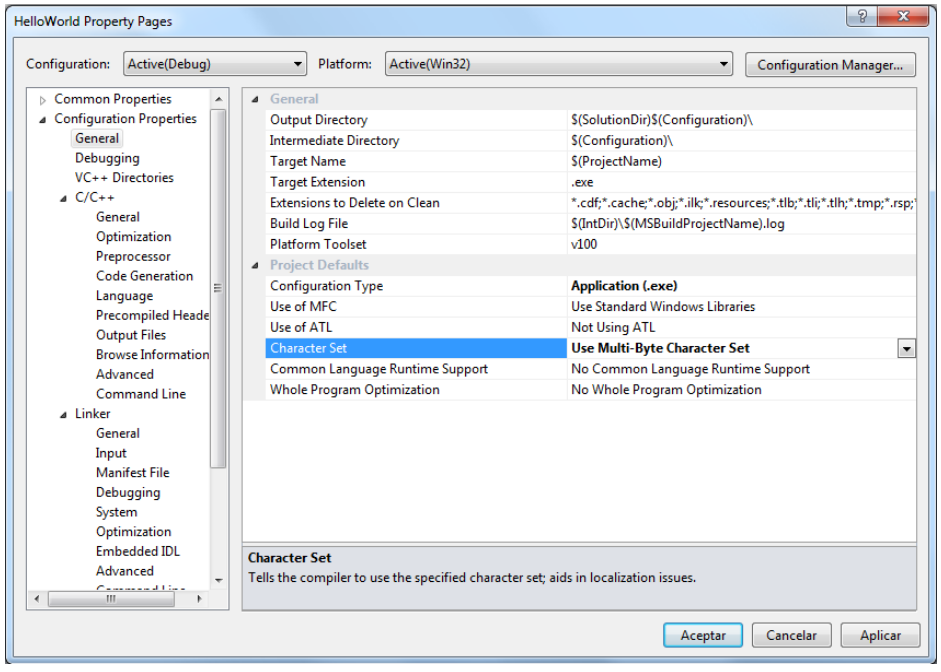

2. En la **sección vinculador (Linker)** debemos añadir la ruta de la carpeta donde se encuentran las librerías de MPI (“*Additional Library Directories*”). Por defecto colgará de *\Archivos de Programa (x86)\DeinoMPI\lib*:



3. Dentro de la **sección vinculador (Linker)**, en el apartado de **entrada** (“*Input*”) hay que añadir como **dependencias adicionales** (“*Additional Dependencies*”) los ficheros “**cxx.lib**” y “**mpi.lib**”:




4. En la sección **General** es necesario usar el **juego de caracteres** (“*Character Set*”) **Multi-Byte**:

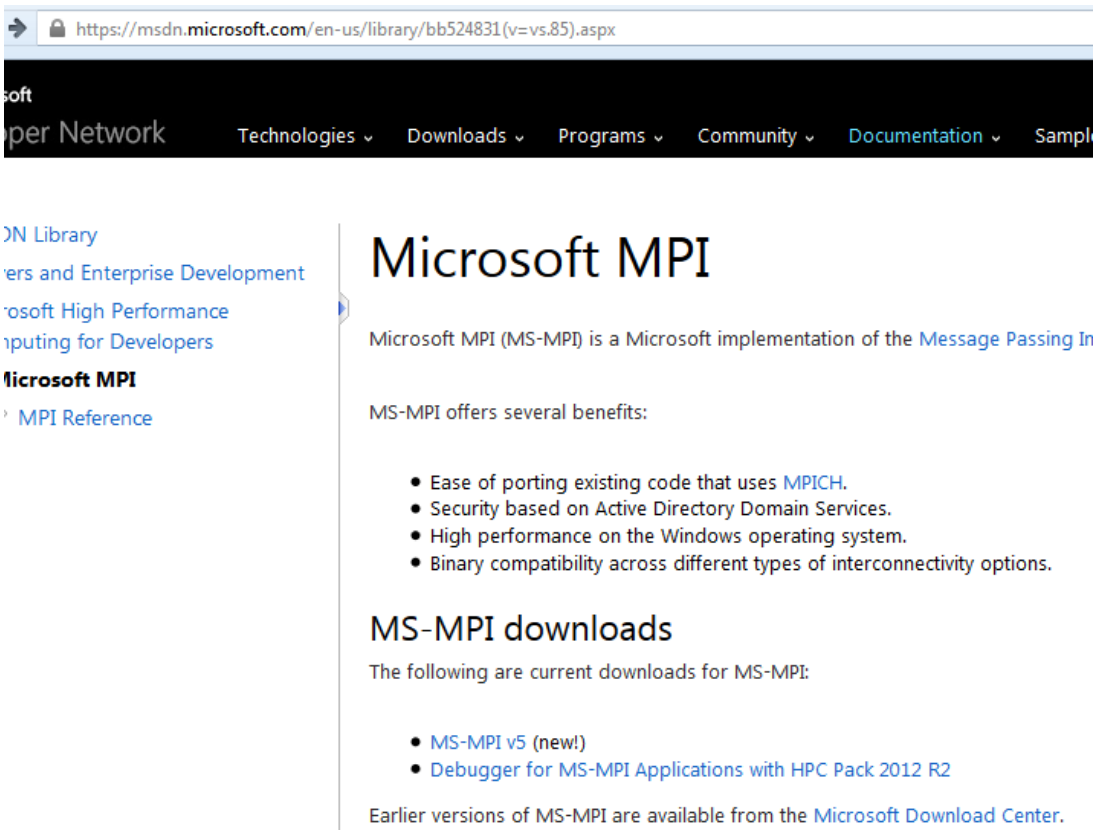



- Finalmente, se debe insertar el código en el fichero fuente creado y **construir**.

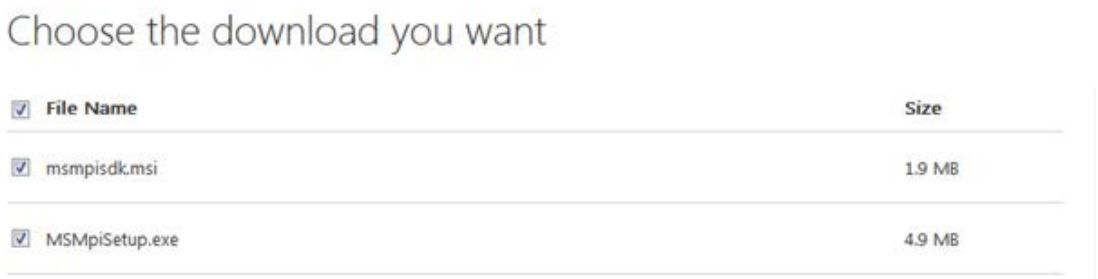
# Apéndice C: Configuración de MS-MPI.

DeinoMPI es difícil de configurar en algunos sistemas y puede en ocasiones no funcionar. Como alternativa podemos instalar y configurar la distribución de MPI de Microsoft. No nos va a proporcionar una interfaz gráfica pero podemos hacer lo mismo desde la línea de comandos. Seguiremos los siguientes pasos para ponerlo en funcionamiento:

- Descargar MS-MPI v5 de su localización web:



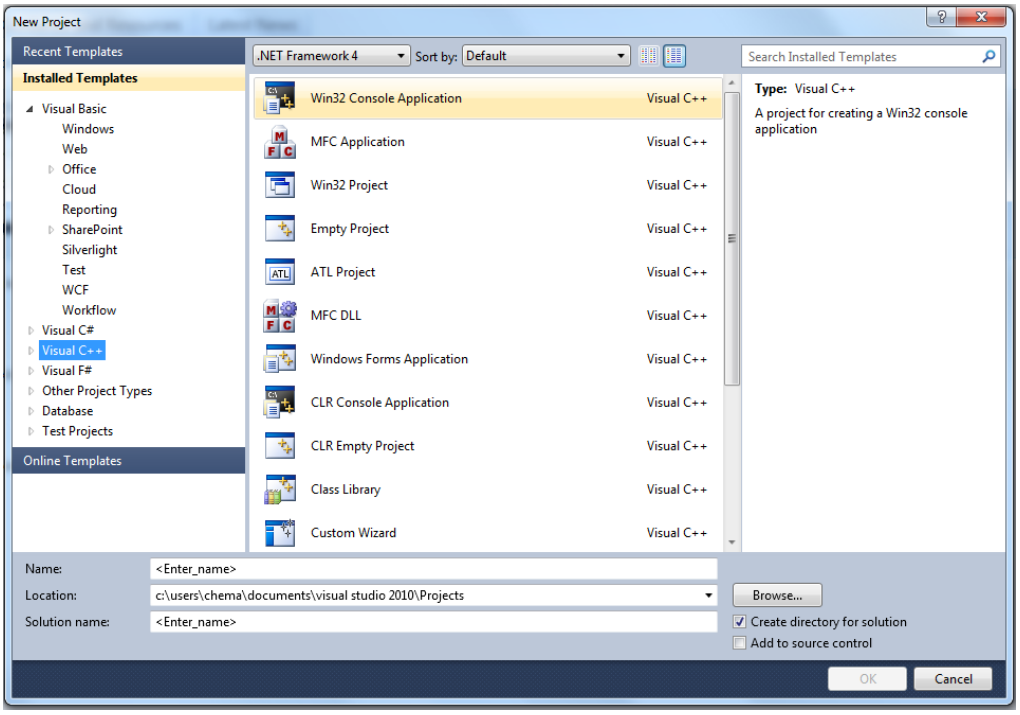

- Hay dos ficheros que descargar e instalar:



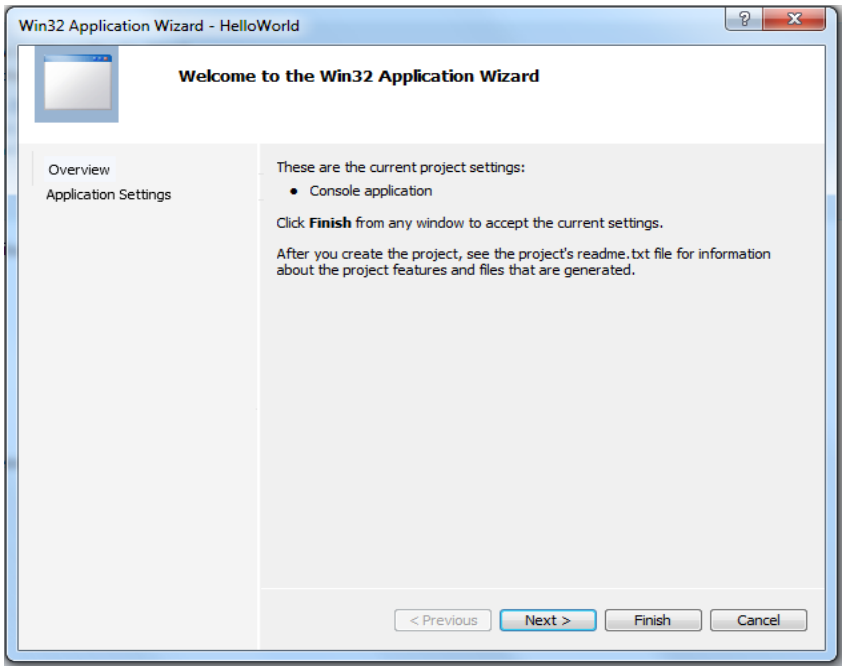

- Cada paquete crea una nueva carpeta: Archivos de Programa > Microsoft MPI y Archivos de Programa > Microsoft SDKs > MPI.

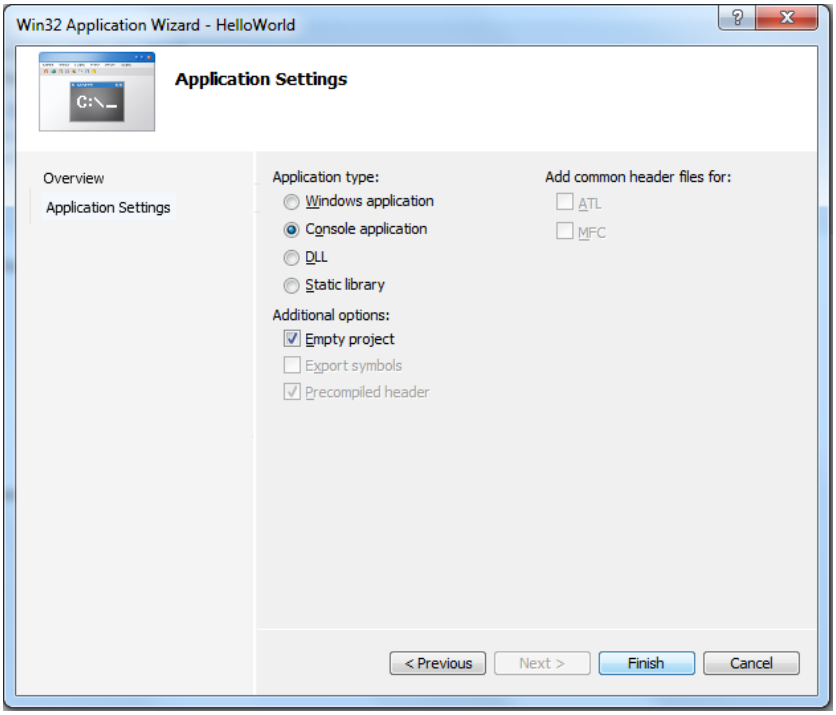
Ahora ya podemos crear una aplicación. Lo haremos a través de MS Visual Studio siguiendo los pasos que se detallan a continuación:

- Crear un **nuevo proyecto y solución**. Se les puede dar a ambos el mismo nombre:

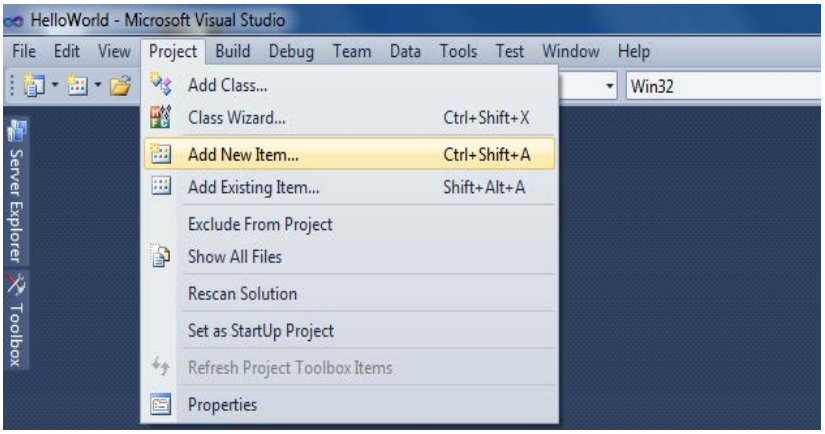


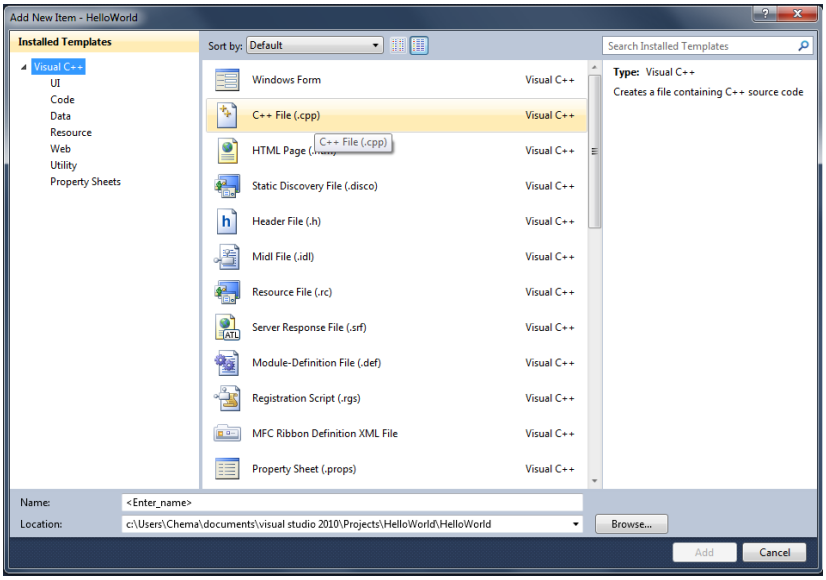

- Configurarlo como **proyecto vacío** ("*Empty project*"):



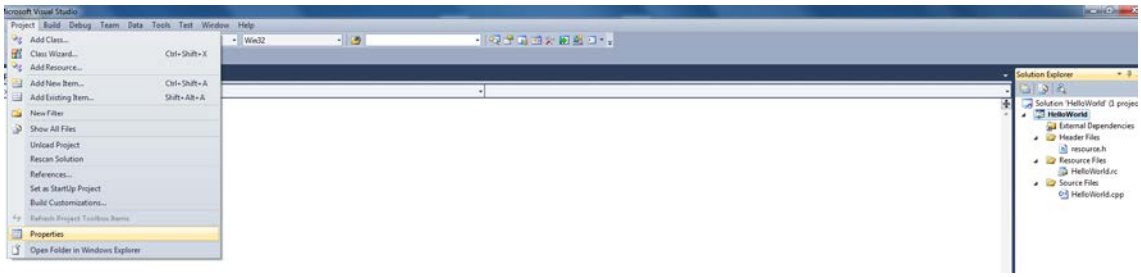



- Una vez creado el proyecto y la solución, añadir un fichero de código como **nuevo elemento** (“*Add New Item*”):

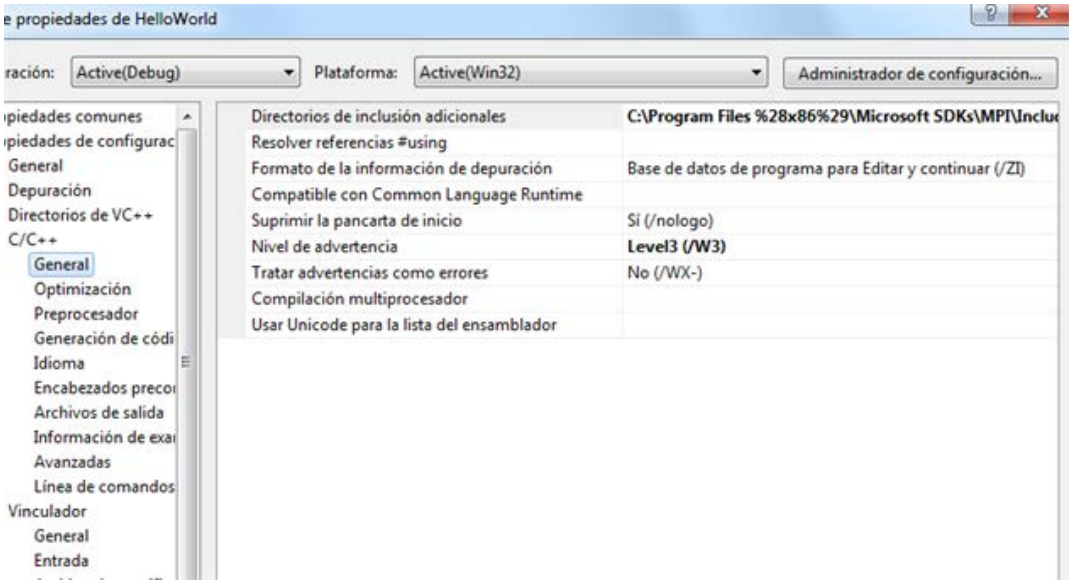




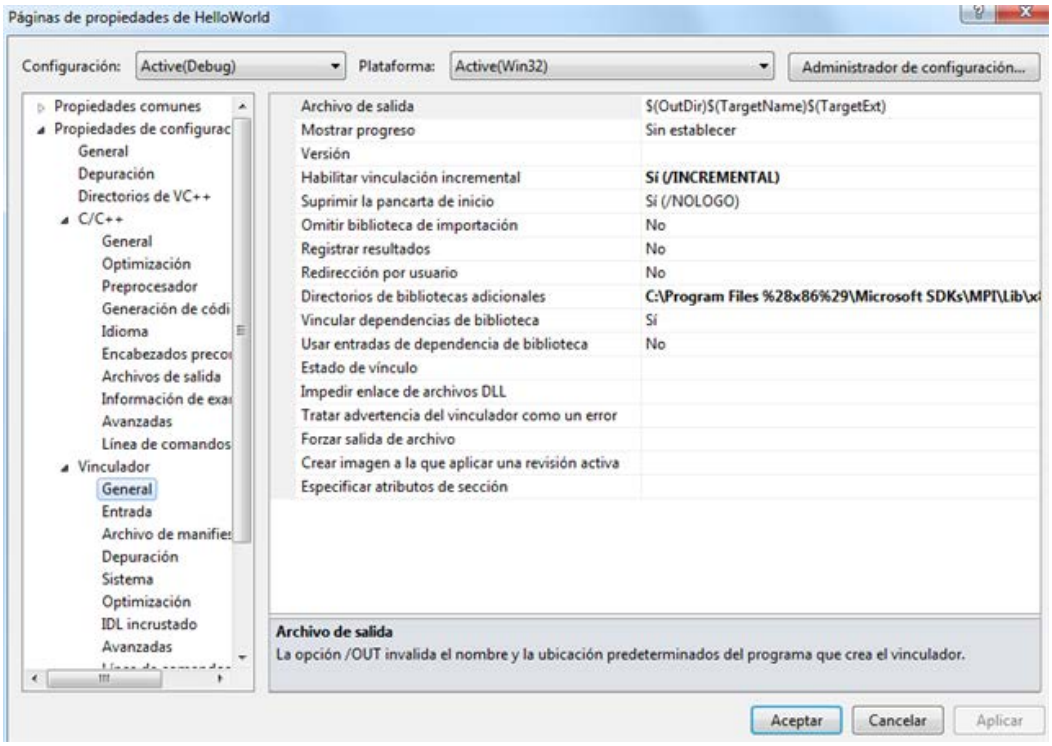

- A continuación, y no antes, se van a ajustar las **propiedades del proyecto** (“*Properties*”):



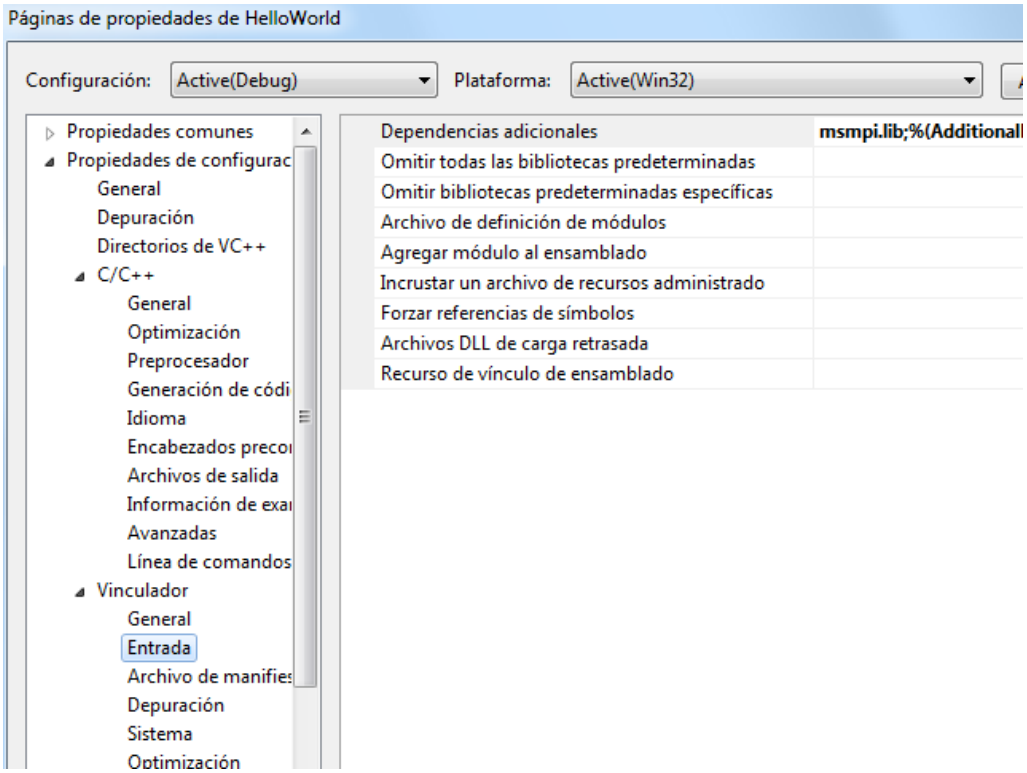
1. Configurar el Nuevo directorio include adicional.



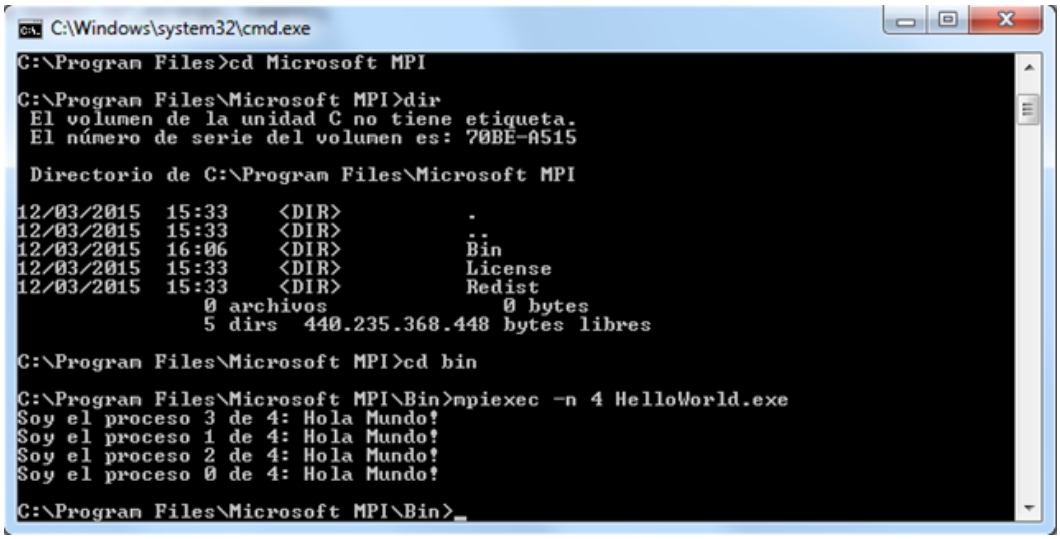

2. De forma análoga configurar la nueva carpeta lib. Es necesario seleccionar la subcarpeta x86 para aplicaciones de 32 bit o x64 para las de 64 bits.




3. Ajustar también la nueva librería MPI.




4. Ahora ya se puede construir la nueva solución como siempre. Para ejecutar el programa, el fichero .exe y el lanzador de MPI deben estar en la misma carpeta o si no se debe ajustar el path para que apunte a la ruta del lanzador. El lanzador es mpiexec.exe y se encuentra en Archivos de Programa > Microsoft MPI > bin. Escribe mpiexec –n np program.exe, donde np es el número de procesos que se pretende lanzar.


C:\Windows\system32\cmd.exe
C:\Program Files>cd Microsoft MPI
C:\Program Files\Microsoft MPI>dir
El volumen de la unidad C no tiene etiqueta.
El número de serie del volumen es: 70BE-A515
Directorio de C:\Program Files\Microsoft MPI
12/03/2015 15:33 <DIR> .
12/03/2015 15:33 <DIR> ..
12/03/2015 16:06 <DIR> Bin
12/03/2015 15:33 <DIR> License
12/03/2015 15:33 <DIR> Redist
0 archivos 0 bytes
5 dirs 440.235.368.448 bytes libres
C:\Program Files\Microsoft MPI>cd bin
C:\Program Files\Microsoft MPI\Bin>mpiexec -n 4 HelloWorld.exe
Soy el proceso 3 de 4: Hola Mundo!
Soy el proceso 1 de 4: Hola Mundo!
Soy el proceso 2 de 4: Hola Mundo!
Soy el proceso 0 de 4: Hola Mundo!
C:\Program Files\Microsoft MPI\Bin>