Índice General

1. Introducción e instalación.
2. Comunicación punto a punto bloqueante.
3. Comunicación punto a punto no bloqueante.
4. Llamadas colectivas.
   - Barrier, Broadcast, gather, scatter.
5. Llamadas colectivas.
   - Todos con todos, reduction.
6. Tipos de datos / Comunicadores.

3. Comunicación No Bloqueante

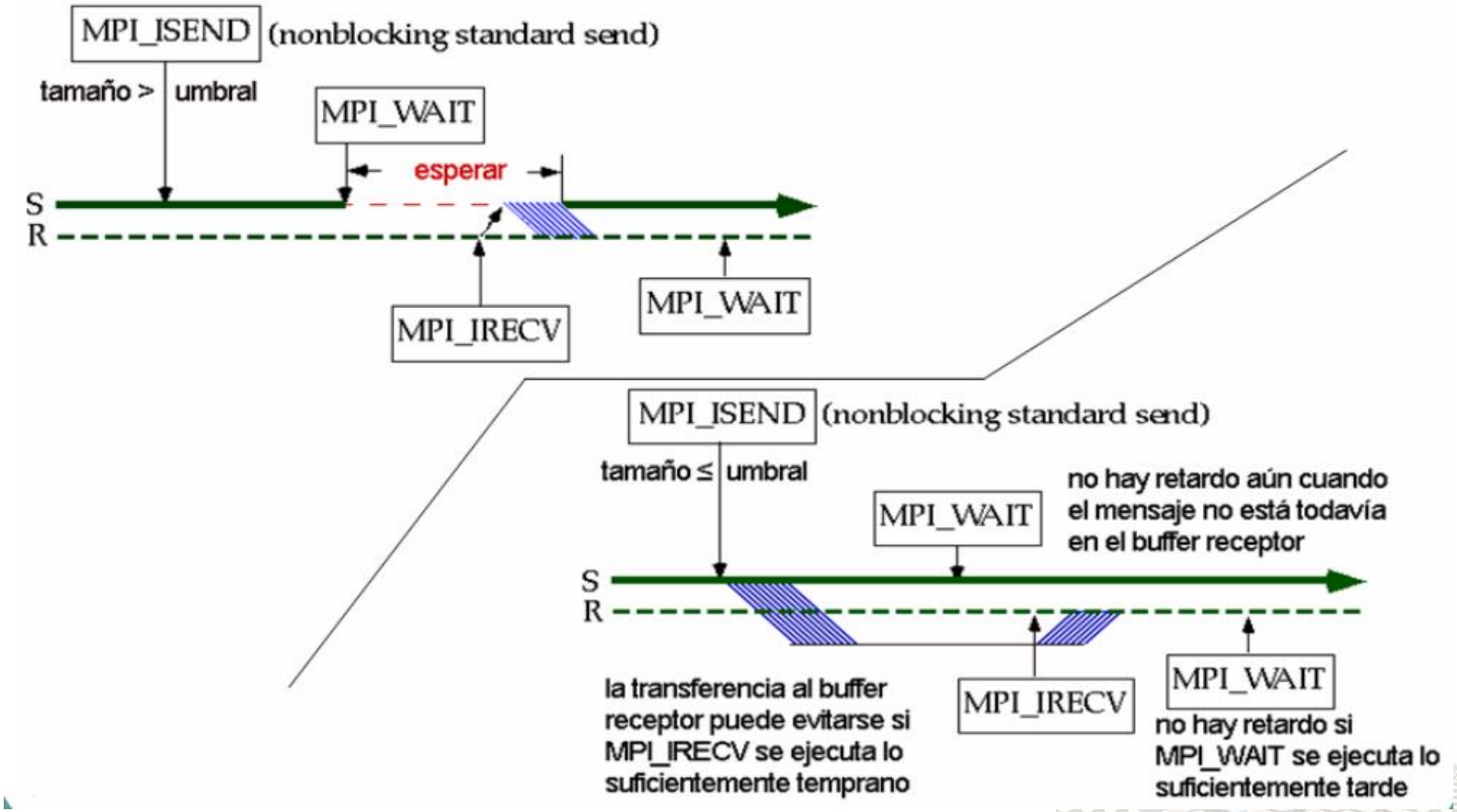
- Cuando no interesa bloquearse esperando una comunicación, sino seguir realizando computación mientras el buffer se libera o llega el mensaje.
- Las operaciones vienen acompañadas de un recibo (request), guardado en memoria del sistema que es opaco al usuario.
  - La constante MPI_REQUEST_NULL indica uno no válido.
- Se puede utilizar tanto en envíos (cuando el buffer está libre) como en recepciones (cuando el mensaje está en el buffer).
- El **status** se obtiene cuando se completa la operación**.
- **Recepción** (sólo hay modo básico):
  - ```
    int MPI_Irecv(void* buf, int count,
    MPI_Datatype datatype, int source,
    int tag, MPI_Comm comm,
    MPI_Request *request);
    ```


    ```

3. Comunicación No Bloqueante

- **Envío** (Básico, con Buffer, Síncrono, Ready):
  - `int` **`MPI_Isend`**`(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request);`
  - `int` **`MPI_Ibsend`**`(void* buf, int count, MPI_Datatype, datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request);`
  - `int` **`MPI_Issend`**`(void* buf, int count, MPI_Datatype, datatype, int dest, int tag, MPI_Comm comm , MPI_Request *request);`
  - `int` **`MPI_Irsend`**`(void* buf, int count, MPI_Datatype, datatype, int dest, int tag, MPI_Comm comm , MPI_Request *request);`

3. Modo Básico



3. Comunicación No Bloqueante

- Todas las funciones no bloqueantes toman un dato de tipo **MPI_Request**, que sirve de recibo de la operación y podrá ser utilizado (sólo o en un array) para comprobar si la operación ha terminado (bien bloqueándose, bien sólo preguntando o bien para cancelarla):
  - `int MPI_Wait(MPI_Request *request, MPI_Status *status);`
  - `int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);`
  - `int MPI_Cancel(MPI_Request *request);`
- Si se ha terminado (flag a 1) se obtiene **status** **.
- Un objeto de tipo ***request*** desaparece con una llamada a ***wait*** o ***test*** correcta, también puede hacerse desaparecer explícitamente con:
  - `int MPI_Request_free(MPI_Request *request);`

3. Comunicación No Bloqueante

- En algunos casos un proceso necesita hacer varias comunicaciones no bloqueantes, hacer un trabajo y después comprobar si todas o algunas de ellas han tenido fin.
- Para ello se puede definir un array de recibos, utilizar cada elemento en una comunicación y después comprobarlos:
  - `int` **`MPI_Testany`**`(int count, MPI_Request *array de request, int *index, int *flag, MPI_Status *status);`
  - `int` **`MPI_Testall`**`(int count, MPI_Request *array de request,int *flag, MPI_Status *array de status);`
  - `int` **`MPI_Testsome`**`(int count, MPI_Request *array de request, int *cuantos, int *array de index, int *flag, MPI_Status *array de status);`
- Existen las operaciones equivalentes para wait: **`MPI_Waitany`**, **`MPI_Waitall`**, **`MPI_Waitsome`**

3. Comunicación por Encuesta

- En algunos casos conviene saber en la recepción, antes de copiarlo al buffer de entrada, cuál es el mensaje que nos ha llegado en cuanto a su tipo de dato, etiqueta o longitud. Para ello existen llamadas bloqueantes y no bloqueantes:
  - `int` **`MPI_Iprobe`**`(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status);`
  - `int` **`MPI_Probe`**`(int source, int tag, MPI_Comm comm, MPI_Status *status);`
- Después deberemos llamar a la correspondiente función de recepción para copiar el mensaje y tratarlo, pero ya tendremos información de si ha llegado algo y qué ha llegado.

3. Comunicación persistente

- En algunos casos, se realiza dentro de un bucle una serie de comunicaciones con los mismos argumentos. Se puede optimizar esta serie a través de una comunicación persistente.
- Para ello se obtiene un recibo que es usado después (no se comunica nada):
  - `int MPI_Send_init(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request);`
  - `int MPI_Recv_init(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request);`
- La comunicación comienza con:
  - `int MPI_Start(MPI_Request *request);`
- Todas las operaciones persistentes siempre son **no** bloqueantes (hay cuatro envíos, sólo aparece el basic) y deben ser recibidas de la misma forma.
- Existe también para arrancar varias pendientes:
  - `int MPI_Startall(int count, MPI_Request *array de request)`

3. Entorno

- Hay partes donde la implementación de MPI puede interactuar con el sistema, como en E/S o señales, pero MPI no obliga a hacerlo por mor de la compatibilidad.
- Si en nuestros programas decidimos usar señales debemos asegurarnos que no interfieren con el MPI, cuestión que probablemente ocurrirá con SIGALRM, SIGFPE y SIGIO.
- También podemos conseguir el nombre del computador:
  - `int MPI_Get_processor_name(char *name, int *resultlen);`

3. Otras consideraciones

- La entrada/salida no suele ser paralela (en MPI 2 si), la salida puede ser posible en cada procesador, pero no tiene sentido la entrada en varios procesadores, por lo que el lanzador desvía la entrada de los procesos distintos del raíz a /dev/null.
  - Ver práctica de trapezoides al recoger los valores de a, b y n.
- Otra cosa es que la entrada se realice desde un fichero.

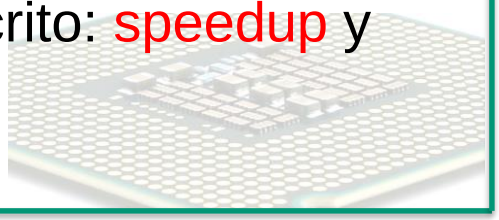
3. Instrumentalización

- Cuando deseamos medir el tiempo de ejecución de un trabajo MPI, tenemos dos maneras:
  - Lo habitual, lanzándolo con time.
  - Usando llamadas MPI para medir el tiempo de una parte del código a otra.
- Esta llamada es MPI_WTIME , que mide el tiempo en segundos de reloj en un procesador desde un tiempo en el pasado. Normalmente se usa dos veces para calcular la diferencia. La precisión del tiempo viene dada por MPI_Wtick().
  - `double MPI_Wtime( );`
  - `double MPI_Wtick( );`
- Ejemplo:

```
double inicio, fin;
inicio= MPI_Wtime();
.... Tiempo a medir  ...
fin = MPI_Wtime();
printf("El tiempo ha sido de %lf segundos\n",fin-inicio);
```

3. Paralelismo

- Cuando estamos trabajando con programas secuenciales se habla de análisis asintótico (notación O). Una función que sirve de cota superior y que fundamentalmente depende del número de datos. Ts(N).
- En programación paralela, ese tiempo de computación no sólo dependerá de los datos N, sino del número de procesadores P. Tp(N,P).
- En el caso ideal, el tiempo de ejecución del programa paralelo será P veces menor que el programa secuencial. Esto raras veces ocurre, ya que hay que tener en cuenta el overhead de las comunicaciones al resolver el problema.
- Se usan dos parámetros para medir lo descrito: speedup y eficiencia.



3. Paralelismo

- Se usan dos medidas para juzgar un programa paralelo:
  - El speedup es la ratio entre el tiempo de ejecución del programa secuencial más rápido en uno de los procesadores y el tiempo de ejecución en la máquina paralela (ley de Amdahl). Si el valor es igual a P se dice que tenemos speedup lineal. Normalmente es menor por la comunicaciones.

$$S = Ts(N) / Tp(N,P)$$

  - La eficiencia es la comparación entre como se usa un procesador con el programa secuencial y como se hace con el paralelo, típicamente es el cociente entre el speedup y el número de procesadores. Aquí la eficiencia suele ser menor que uno por el motivo anterior.

$$E = S / P$$

3. Paralelismo

- El *speedup* que podemos obtener para una fracción R paralelizable es:

$$S = Ts/Tp = Ts / ((1-R)xTs+RxTs/P) = 1 / ((1-R) + R/P)$$

- Aplicado a una mejora infinita:

$$S \rightarrow 1 / (1-R)$$

- No está incluido el factor N explícitamente.
- Pero deberemos vigilar siempre las tres fuentes de overhead (no están en el programa secuencial que hacen que S no sea lineal):
  - Comunicación. Se mejora haciendo coincidir el tiempo de cálculo con el de comunicación.
  - Tiempo de espera. Como antes y con balanceo de carga.
  - Extra computación. Cálculo sólo necesario para la versión paralela.

3. Paralelismo

- Pero, ¿cómo evaluamos el tiempo?
- El tiempo de ejecución del caso paralelo será la suma del tiempo de cálculo más la E/S (caso secuencial) más el tiempo de comunicación:

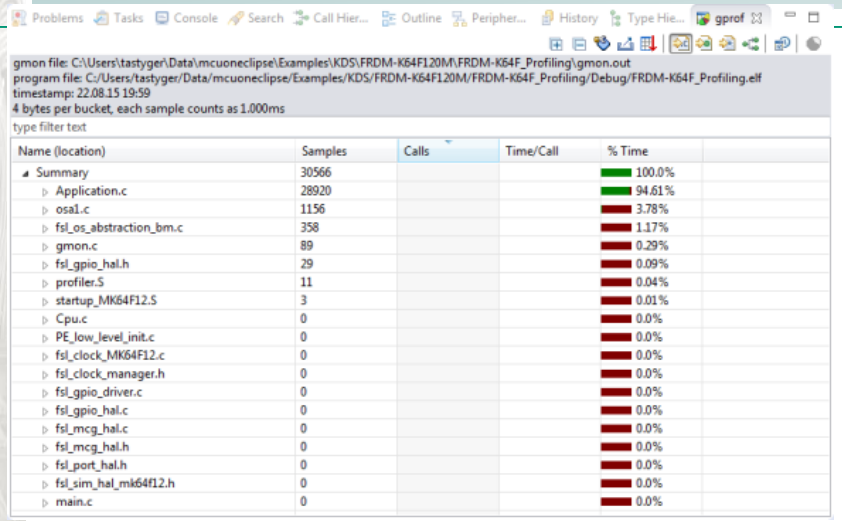
$$Tp(N,P) = Tc + Tio + Tcom = Ts + Tcom$$

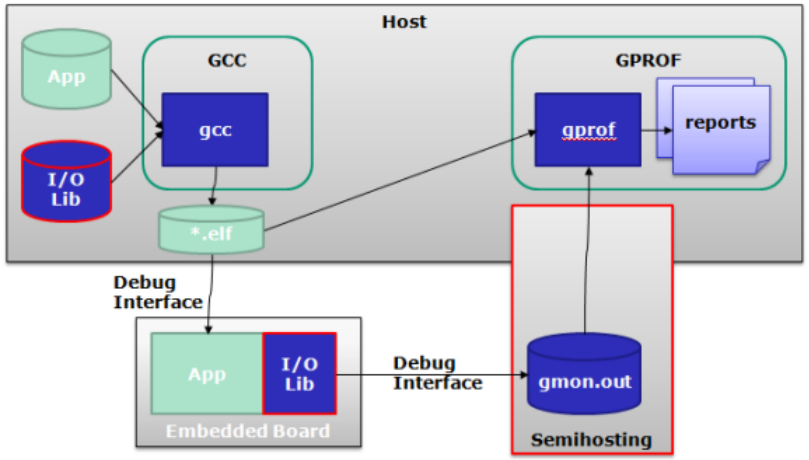
- El tiempo de comunicación es dependiente de la implementación pero en general depende de:
  - El start-up, típicamente copia del mensaje junto con su "envoltorio" (rango, origen, destino, tag...). También se le llama latencia.
  - La comunicación. Transmisión de los datos entre los procesadores. Usualmente este tiempo depende linealmente del tiempo en mandar una unidad de datos. Al inverso se le llama ancho de banda (*bandwidth*).

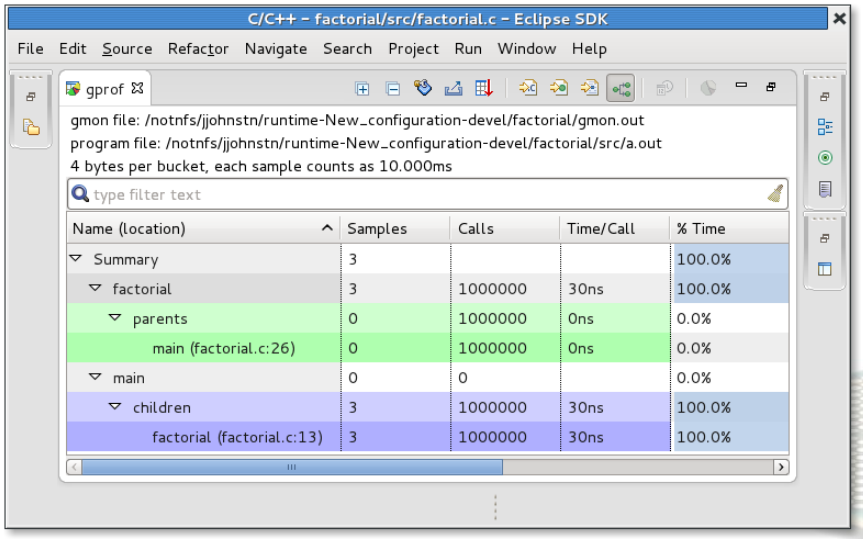
3. Diseño

- Brevemente hay dos maneras de abordar el diseño de programa paralelos:
  - Datos. Se reparten los datos a procesar entre los distintos procesadores (procesos) que tengamos. Suele ser más fácil de hacer y escala mejor.
  - Control. Repartimos las tareas a realizar entre los distintos procesadores. Cada proceso hace algo diferente.
  - Método mixto.
- Pero antes de proceder hay que saber donde intervenir en un programa secuencial por la ley de Amdahl.
  - Es inútil paralelizar partes del código que usan muy poca CPU o no se ejecutan frecuentemente.
  - Para ello existen herramientas básicas GNU de profiling como GNU gprof y Callgrind/Valgrind (cache).

3. Profiling


Problems
Tasks
Console
Search
Call Hier...
Outline
Peripher...
History
Type Hie...
gprof
gmon file: C:\Users\tastyger\Data\mcuoneclipse\Examples\KDS\FRDM-K64F120M\FRDM-K64F_Profiling\gmon.out
program file: C:/Users/tastyger/Data/mcuoneclipse/Examples/KDS/FRDM-K64F120M/FRDM-K64F_Profiling/Debug/FRDM-K64F_Profiling.elf
timestamp: 22.08.15 19:59
4 bytes per bucket, each sample counts as 1.000ms
type filter text
Name (location) | Samples | Calls | Time/Call | % Time
Summary 30566 100.0%
Application.c 28920 94.61%
osa1.c 1156 3.78%
fsl_os_abstraction_bm.c 358 1.17%
gmon.c 89 0.29%
fsl_gpio_hal.h 29 0.09%
profiler.S 11 0.04%
startup_MK64F12.S 3 0.01%
Cpu.c 0 0.0%
PE_low_level_init.c 0 0.0%
fsl_clock_MK64F12.c 0 0.0%
fsl_clock_manager.h 0 0.0%
fsl_gpio_driver.c 0 0.0%
fsl_gpio_hal.c 0 0.0%
fsl_mcg_hal.c 0 0.0%
fsl_mcg_hal.h 0 0.0%
fsl_port_hal.h 0 0.0%
fsl_sim_hal_mk64f12.h 0 0.0%
main.c 0 0.0%


Host
App
GCC
gcc
I/O Lib
*.elf
GPROF
gprof
reports
Debug Interface
App
I/O Lib
Embedded Board
Debug Interface
gmon.out
Semihosting


C/C++ – factorial/src/factorial.c – Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
gprof
gmon file: /notnfs/jjohnstn/runtime-New_configuration-devel/factorial/gmon.out
program file: /notnfs/jjohnstn/runtime-New_configuration-devel/factorial/src/a.out
4 bytes per bucket, each sample counts as 10.000ms
type filter text
Name (location) | Samples | Calls | Time/Call | % Time
Summary 3 100.0%
factorial 3 1000000 30ns 100.0%
parents 0 1000000 0ns 0.0%
main (factorial.c:26) 0 1000000 0ns 0.0%
main 0 0 0.0%
children 3 1000000 30ns 100.0%
factorial (factorial.c:13) 3 1000000 30ns 100.0%

3. Debugging

- Hay herramientas del BSC muy utilizadas en el mundo HPC:
- Dimemas: Para desarrollar y tunear aplicaciones paralelas mediante simulación de arquitecturas. Predice el rendimiento de estas aplicaciones.
- Paraver: Para ver gráficamente el comportamiento de aplicaciones paralelas de las que se dispone de una traza.
- Extrae: Genera trazas para Paraver.