Introducción a BSP

La motivación para el modelo de computación paralela BSP (*The Bulk-Synchronous Parallel Model*) surge de una comparación con lo que se observa en el mundo de la computación secuencial donde el modelo de Von Neumman ha sido el estándar de facto desde 1944. En particular, en el mundo secuencial observamos una gran industria de software donde se produce software portable para computadores secuenciales Von Neumman. En este caso, Von Neumman actúa como el punto de unión entre software y hardware, es decir el software se produce para máquinas de Von Neumman y los fabricantes de computadores producen máquinas de Von Neumman cada vez más eficientes para ejecutar dicho software.

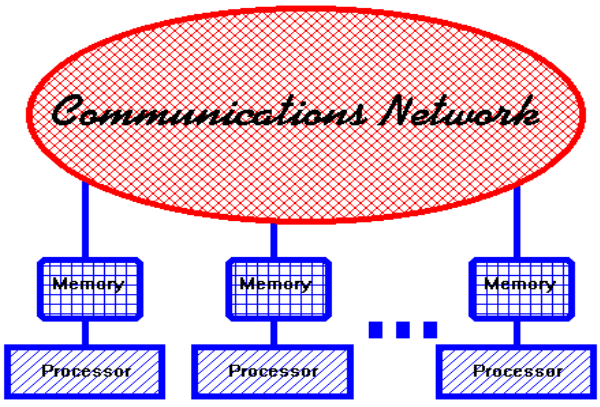
Como contraste, en el mundo de la computación paralela no se ha logrado tal desarrollo en productos de software y hardware. Algunos investigadores conjeturan de que la razón principal se debe a que no se ha adoptado un modelo de computación paralela universalmente aceptado en el sentido del modelo de Von Neumman. Para lograr tal rol, el modelo debería cumplir los siguientes criterios

- **Escalabilidad:** El desempeño del software y hardware debe ser eficiente desde un grupo pequeño de procesadores a un grupo muy grande de procesadores.
- **Portabilidad:** El software debe correr sin cambios, con desempeño eficiente, en cualquier arquitectura computacional de propósito general.
- **Predecibilidad:** El desempeño del software sobre diversas arquitecturas debe ser predecible con facilidad.
- **Simplicidad:** Debería ser sencillo escribir programas paralelos, e idealmente la verificación de estos programas no debería ser más complicada que la de los programas secuenciales.

Durante los últimos treinta años, han sido propuestos un gran número de modelos de computación paralela. Sin embargo, ninguno de estos modelos han alcanzado los primeros tres requerimientos. El modelo BSP (*The Bulk-Synchronous Parallel Model*) es uno de los primeros modelos en cumplir con tales requerimientos, y en los últimos años ha alcanzado creciente aceptación de parte de los usuarios de computación paralela.

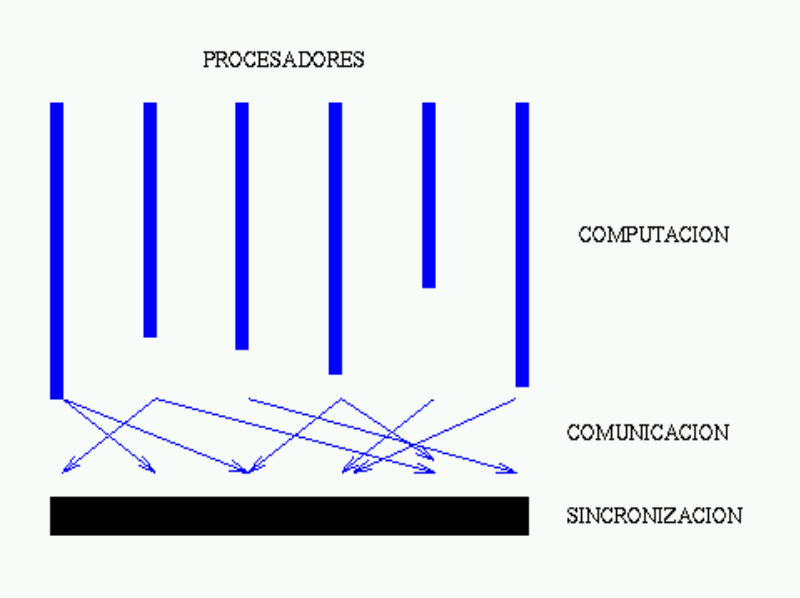
El modelo

En BSP, un computador paralelo es visto como un conjunto de procesadores con memoria local e interconectados a través de una red de comunicación de topología transparente al usuario. Esta abstracción se aplica tanto a sistemas de memoria distribuida como de memoria compartida.

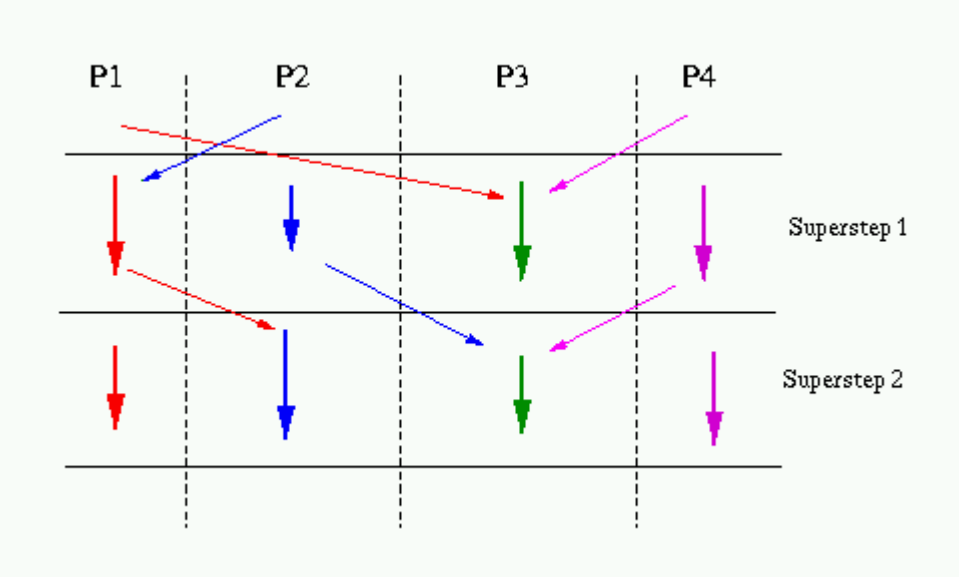


La computación es organizada como una secuencia de *supersteps*. Durante un superstep, cada procesador puede realizar operaciones sobre datos locales y depositar mensajes a ser enviados a otros procesadores. Al final del superstep, todos los mensajes son enviados a sus destinaciones y los procesadores son sincronizados en forma de barrera para iniciar el siguiente superstep. Es decir, los mensajes están disponibles en sus destinaciones al instante en que se inicia el siguiente superstep.

Tal como lo indica la figura siguiente, un superstep está formado por una fase en que los procesadores realizan computaciones sobre datos almacenados en sus memorias locales, tiempo en el cual pueden depositar mensajes a ser enviados a otros procesadores, y al final del superstep se envian todos los mensajes y los procesadores se sincronizan con el objetivo de asegurar que todos los han recibido los mensajes, y así comenzar el siguiente superstep.



En general, un programa en BSP estará compuesto de una secuencia de supersteps, cada uno iniciado luego de finalizar el anterior como se muestra en la siguiente figura para cuatro procesadores *P1*, *P2*, *P3*, y *P4*.



Note que la estructura del modelo facilita la predicción del desempeño de programas BSP. El costo de un programa esta dado por la suma del del costo de todos sus supersteps, donde el costo de cada superstep esta dado por

- la suma del costo originado por las computaciones sobre datos locales (el máximo sobre los

procesadores),

- el costo de las comunicaciones de mensajes entre procesadores (una función del máximo enviado/recibido sobre los procesadores), y
- el costo asociado a la sincronización de los procesadores.

Mediante benchmarks aplicados al momento de instalar BSP en una plataforma computacional dada, se puede determinar

- el costo *g* de transmitir un word desde un procesador a otro en una situación de tráfico continuo en la red de comunicaciones, y
- el costo *l* de sincronizar los procesadores.

El costo de un algoritmo se representa como una función de los parámetros *g* y *l*. El valor de estos parámetros depende del número de procesadores *p*. Un ejemplo de parámetros para diversas máquinas se muestra en la siguiente tabla (*s* es la velocidad del procesador en *flops*).

MACHINE	*s* *(Mflop/s)*	*p* *(no. procs)*	*l* *(flops)*	*g* *(flops/word)*
Cray T3D	12	1	68	0.3
		2	164	1.0
		4	168	0.8
		8	175	0.8
		9	383	1.2
		16	181	1.0
		25	486	1.5
		32	201	1.4
		64	148	1.7
		128	301	1.8
		256	387	2.4
IBM SP2 (Switch)	26	1	244	1.3
		2	1903	7.8
		4	3583	8.0
		8	5412	11.4
Pentium Pro NOW	61	1	85	1.0
		2	52745	484.5
		4	139981	1128.5
		8	539159	1994.1
		10	826054	2436.3
		16	2884273	3614.6

El costo de un superstep está dado por la suma

$$w + \max\{h_s, h_r\} \cdot g + l$$

donde w representa el máximo en cantidad de computación registrada en cualquier procesador, h_s/h_r es el máximo enviado/recibido en comunicación por cualquier procesador durante el superstep. El costo del programa es la suma de los costos de sus supersteps.

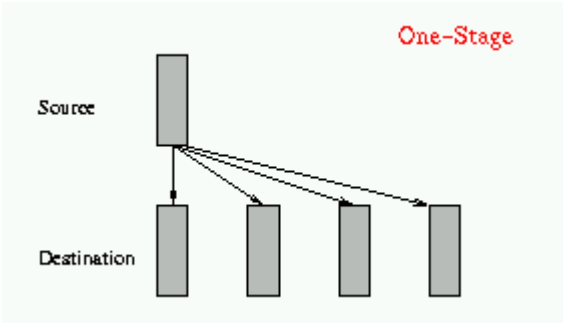
Por lo tanto, el diseño de algoritmos BSP eficientes debería estar guiado por los siguientes criterios

- balance en computación en cada proceso (procesador), puesto que w es un máximo en computación.
- balance en comunicación puesto que h_s/h_r es un máximo sobre los datos enviado o recibidos, y
- minimización del número total de supersteps requeridos para completar el programa.

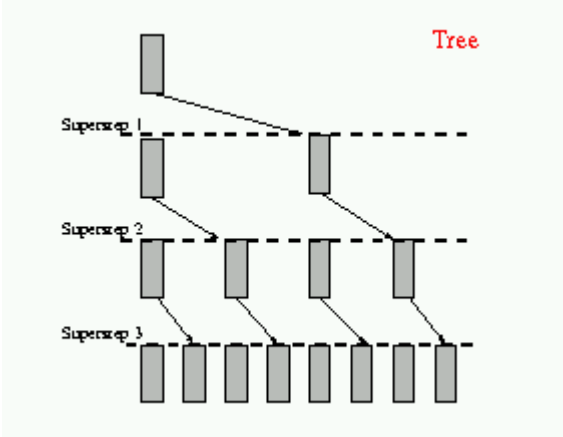
Ejemplos de algoritmos BSP

Como ejemplo de diseño de algoritmos BSP vamos a describir algunas soluciones para el caso en que un procesador desea comunicar información a todos los demás procesadores.

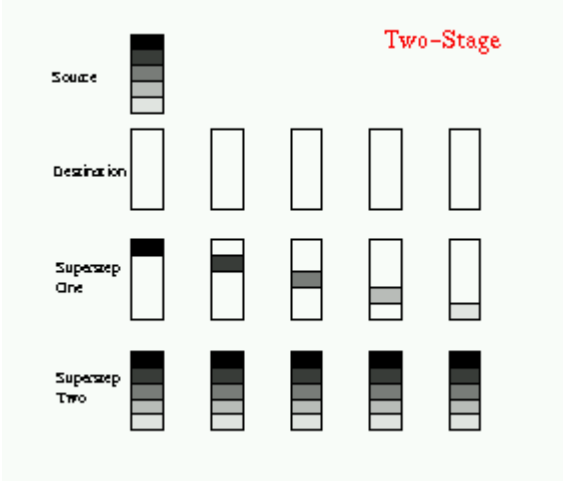
Un primer método es realizar un superstep donde el procesador fuente envía un dato de tamaño N a todos los otros, lo cual tiene un costo igual a [$l + p\,N\,g$], donde p es el número de procesadores.



Otra solución es realizar un número logarítmico de supersteps y comunicar el dato en forma de árbol, a un costo total igual a [$(l + N\,g)\,log\,p$].



Finalmente, una tercera alternativa es proceder en dos pasos (supersteps). En el primero, el procesador fuente un segmento distinto del dato, cada uno de tamaño *N/p*, a cada procesador respectivamente. En el siguiente paso todos los procesadores envian a todos los otros el segmento de datos que les corresponde (note que en este caso cada procesador incurre en un costo igual a [*p N/p g*] en comunicación). Luego el costo total de la operación es igual a [*2 l + 2 N g*].



BSP*lib*

Es una biblioteca de comunicaciones que permite codificar programas paralelos de manera simple, liberando al programador de los detalles de la comunicación y sincronización entre procesos (procesadores). Como muchas otras bibliotecas de comunicación, BSPLib adopta un modelo de programación de un solo programa con múltiples datos (SPMD). Es decir, se escribe un sólo programa, el cual se copia en todos los procesadores y se ejecutan en paralelo. Al interior de cada programa se pueden seguir distintos caminos de ejecución y/o se pueden realizar operaciones sobre distintos datos. La única restricción es respetar la organización en supersteps del programa, es decir, los programas deben ejecutar la misma secuencia de supersteps. El conjunto de operaciones de >BSP*lib* es pequeño. La tabla siguiente muestra un resumen.

Tipo	*Operación*	*Significado*
Inicialización	bsp_begin	Inicia la ejecución en varios procesadores
	bsp_end	Fin de la ejecución paralela
	bsp_abort	Terminación anormal (error)
Consulta	bsp_nprocs	Número de procesadores
	bsp_pid	Identificador de proceso
	bsp_time	tiempo de ejecución
Fin de superstep	bsp_sync	Syncronización de barrera
Memoria remota	bsp_push_reg	Declara visibilidad de mem.
	bsp_pop_reg	Remueve visibilidad de mem.
	bsp_put	Escribe en mem. remota
	bsp_get	Lee desde mem. remota
Cola de mensajes	bsp_set_tagsize	Tamaño de encabezado de mensajes
	bsp_send	Envía mensaje

bsp_qsize	Número de mensajes recibidos
bsp_get_tag	Obtiene encabezado del sig. mensaje
bsp_move	Obtiene mensaje recibido desde la cola

Una descripción detallada de las primitivas de BSP*lib* esta disponible aquí.

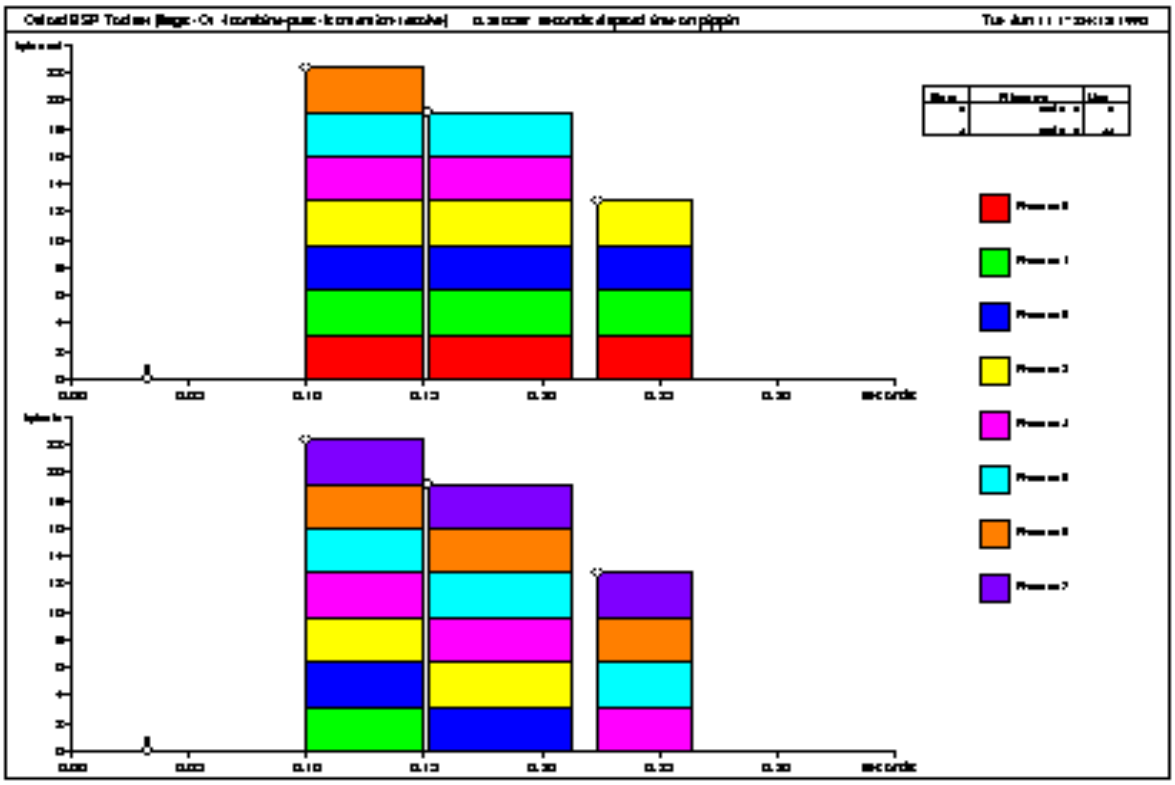
Ejemplo de programa en BSP*lib*

```
#include <stdio.h>
#include <bsp.h>

#define NB 1000

main()
{
  int i,a,b;
  char *p;

  bsp_begin(8); // Inicia 8 procesos.

  p=(char *)malloc(NB); // Crea un buffer de comunicaciones.
  bsp_pushregister(p,NB); // hace el buffer publico,
   // la registracion tiene efecto en el siguiente superstep.

  bsp_sync(); // termina el primer superstep

  // Ahora se ejecutan 10 supersteps, donde en cada
  // superstep cada proceso envia un string al proceso
  // ubicado a su derecha en sentido circular.

  for(i=0;i<10;i++)
  {
    // se crea el string que lleva el mensaje.
    // El string (mensaje) lleva dos enteros.
     sprintf(p,"%d %d",bsp_pid(),i);

    // El string se comunica al vecino derecho. Solo
    // se envian los bytes necesarios.
     bsp_put((bsp_pid()+1)%bsp_nprocs(),p,p,0,strlen(p));

    // Fin del superstep i-esimo.
     bsp_sync();

    // El proceso pid=0 muestra los enteros recibidos.
     if (bsp_pid()==0)
     {
       // El toman los dos enteros desde el string p.
        sscanf(p,"%d %d",&a,&b);

       // Imprime en la salida estandar los dos enteros
       // de la iteracion actual. La salida debe ser la secuencia
       // de pares (7,0) (7,1) (7,2) ..... (7,9)
        printf("(%d,%d) ",a,b);
     }
  }

  bsp_end();
}
```

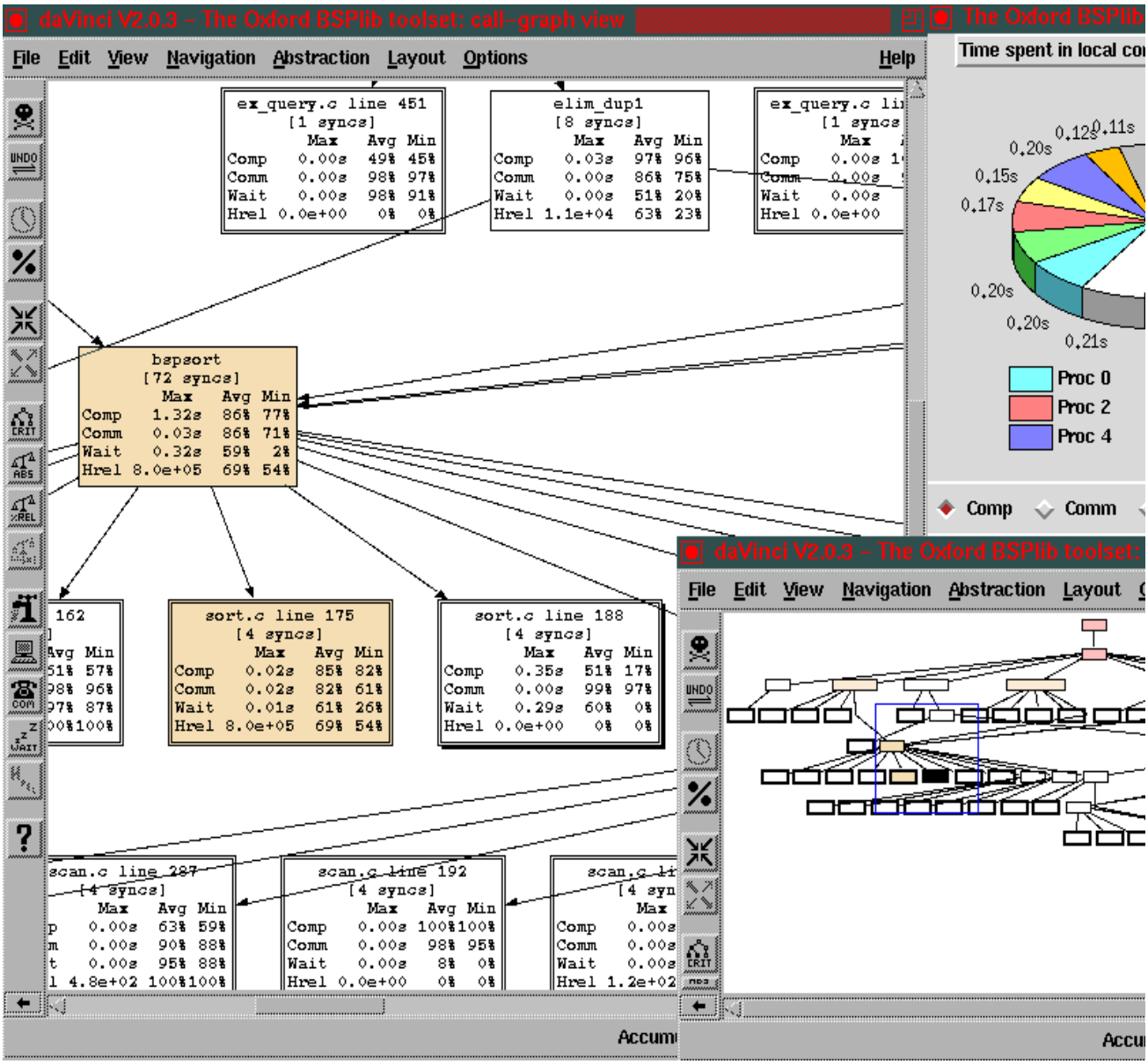
Más ejemplos de programas BSP*lib* están disponibles aquí.

Herramientas gráficas

BSP*lib* incluye herramientas gráficas de visualización del costo BSP de programas. Un ejemplo de gráfico que se obtiene con el comando **bspprof** se muestra en la figura siguiente. La figura muestra dos gráficos, el primero muestra la cantidad de bytes enviados por cada proceso en el tiempo y el segundo los bytes recibidos, en tanto que cada columna señala un superstep ejecutado en el programa y cada color un procesador distinto. El espacio en blanco entre las barras representa el tiempo transcurrido para el procesador que toma mayor tiempo en realizar su fase de cómputo local. El tiempo tomado para realizar la comunicación se muestra en el ancho de las barras, y la etiqueta encontrada en la esquina superior izquierda de cada barra es el número del superstep. La tabla al costado superior derecho indica qué programa y en que línea se ejecuta cada superstep.



Además, es posible visualizar el perfil de ejecución de programas BSP a nivel de código fuente. El objetivo es detectar punto de desbalance en computación, comunicación y sincronización. Un ejemplo es la siguiente figura. Cada nodo del gráfo indica estadíticas recolectadas al término de cada superstep.


daVinci V2.0.3 - The Oxford BSPlib toolset: call-graph view
File Edit View Navigation Abstraction Layout Options Help
ex_query.c line 451
[1 syncs]
Max Avg Min
Comp 0.00s 49% 45%
Comm 0.00s 98% 97%
Wait 0.00s 98% 91%
Hrel 0.0e+00 0% 0%
elim_dup1
[8 syncs]
Max Avg Min
Comp 0.03s 97% 96%
Comm 0.00s 86% 75%
Wait 0.00s 51% 20%
Hrel 1.1e+04 63% 23%
bspsort
[72 syncs]
Max Avg Min
Comp 1.32s 86% 77%
Comm 0.03s 86% 71%
Wait 0.32s 59% 2%
Hrel 8.0e+05 69% 54%
sort.c line 175
[4 syncs]
Max Avg Min
Comp 0.02s 85% 82%
Comm 0.02s 82% 61%
Wait 0.01s 61% 26%
Hrel 8.0e+05 69% 54%
sort.c line 188
[4 syncs]
Max Avg Min
Comp 0.35s 51% 17%
Comm 0.00s 99% 97%
Wait 0.29s 60% 0%
Hrel 0.0e+00 0% 0%
scan.c line 192
[4 syncs]
Max Avg Min
Comp 0.00s 100%100%
Comm 0.00s 98% 95%
Wait 0.00s 8% 0%
Hrel 0.0e+00 0% 0%
Time spent in local co
0.11s
0.12s
0.20s
0.15s
0.17s
0.20s
0.20s
0.21s
Proc 0
Proc 2
Proc 4
Comp
Comm
daVinci V2.0.3 - The Oxford BSPlib toolset:
File Edit View Navigation Abstraction Layout