TFG – Arquitectura de Computadores y Sistemas Operativos

Memoria

Estudio de políticas de job scheduling mediante el simulador SST-Simulator

Autor:

José Antonio Domínguez Medina
jdominguezmed@uoc.edu

UOC-GEI
(Segundo semestre del 2015)

Consultor:

Francesc Guim Bernat

Dedicatoria

A mis padres, por todas las facilidades que me han brindado durante toda mi formación.

A mi princesa Ana, que siempre ha estado a mi lado, alentándome y velando para que cumpla mis objetivos.

Omnia mea mecum porto.

Bías de Priene (S. VI a. C)

1 Índice

1 Índice 4

2 Introducción 6

2.1 Motivación 6

2.2 Objetivos 7

2.3 Resultados 8

2.4 Requerimientos de sistema 10

2.5 Planificación 11

3 El simulador SST-Simulator 12

3.1 Introducción 12

3.2 Configuración 14

3.3 El componente Scheduler 21

3.4 Workloads 26

3.4.1 The San Diego Supercomputer Center (SDSC) SP2 log 31

4 Políticas de Job Scheduling 34

4.1 El Scheduler 34

4.2 Algoritmos de planificación 35

4.2.1 La planificación a corto plazo 35

4.2.1.1 Algoritmo de orden de llegada (FCFS - first come, first served) 36

4.2.1.2 Algoritmo de menor tiempo primero (SJF - shortest job first) 37

4.2.1.3 Algoritmo de menor tiempo pendiente (SRTF - shortest remaining time first) 37

4.2.1.4 Algoritmo Prioritario 38

4.2.1.5 Algoritmo de repartimiento de tiempo (RR - round robin) 39

4.3 Consideraciones sobre las políticas de job scheduling en sistemas HPC 39

4.3.1 Backfilling policies 40

5 Construcción del proyecto 43

5.1 Análisis de requerimientos/definición de funcionalidades 43

5.2 Diseño técnico 44

5.2.1 Función de adquisición de información 46

5.2.2 Función de procesado de la información 48

5.2.3 Función de presentación de la información 53

5.2.4 Función principal (Main) 54

5.3 Ejecución de pruebas 56
5.3.1 Entornos 56
5.3.2 Ejecución 56
5.3.3 Resultados 57

6 Análisis de resultados ***58***

6.1 Uso del procesador (CPU utilization) 60
6.1.1 Consideraciones sobre la métrica 60
6.1.2 Análisis PQScheduler 61
6.1.3 Análisis EASYScheduler 63

6.2 Productividad (throughput) 64
6.2.1 Consideraciones sobre la métrica 64
6.2.2 Análisis PQScheduler 64
6.2.3 Análisis EASYScheduler 66

6.3 Tiempo de retorno (Turnaround Time) 68
6.3.1 Consideraciones sobre la métrica 68
6.3.2 Análisis PQScheduler 68
6.3.3 Análisis EASYScheduler 70

6.4 Tiempo de espera (Waiting Time) 71
6.4.1 Consideraciones sobre la métrica 71
6.4.2 Análisis PQScheduler 71
6.4.3 Análisis EASYScheduler 72

7 Conclusiones ***74***

8 Referencias ***75***

2 Introducción

2.1 Motivación

La simulación es un concepto que ya de pequeño me intrigaba. Esa capacidad que dispone el ser humano para visualizar situaciones, secuencias de acciones que podrían suceder si, en un determinado momento y, proporcionando algún tipo de operación, diésemos paso a la realidad (ejem. Siguiente movimiento del contrario al actuar sobre una determinada pieza del ajedrez).

La capacidad de realizar simulaciones, de evaluar, es vital para nuestra supervivencia, para poder determinar, arriesgar o no, continuar o desistir; en definitiva, para progresar. Y es el afán de progreso uno de los principales motores que dirige la evolución humana, el que hace que busquemos formulas empíricas y propias de nuestra propia naturaleza para aplicarlas a las ciencias y la tecnología. Una de estas fórmulas es, sin duda, la simulación.

Las Ciencias de la Computación es una de las ciencias que se ha visto beneficiada por la capacidad de simulación de los sistemas informáticos, en los cuales, el ingeniero/investigador, configura en el entorno de simulación los recursos disponibles (provee una situación o contexto inicial), inicia una serie de operaciones (en forma de procesos e hilos de ejecución), para comprobar la secuencia de acciones que se suceden y los resultados que son obtenidos.

Actualmente, en el ámbito del desarrollo de sistemas computadores de alto rendimiento (HPC), realizar modificaciones en software y/o hardware (co-design) para crear prototipos y, comprobar el correcto funcionamiento y mejoras en el rendimiento de cualquiera de sus componentes, así como de todo el sistema computador, requeriría un gasto económico y un tiempo de desarrollo que determinarían estos proyectos como inviables. Debido a que los sistemas HPC son de gran importancia para el estudio y el progreso en áreas muy diversas de las ciencias y la tecnología (mecánica cuántica, predicción del tiempo, modelado molecular y comportamiento químico, simulaciones físicas, etc.) resolviendo problemas muy complejos, existe un número de instituciones[1] que han desarrollado un framework para simular sistemas HPC, el SST-Simulator [SST 1], que aspira a convertirse en el estándar de simuladores de sistemas HPC.

SST-Simulator provee un entorno que permite modelar y evaluar las diversas características inherentes a los actuales sistemas HPC, como el hecho de estar compuestos de múltiples procesadores (sistemas UMA y NUMA) y/o multinúcleos, en los cuales el número de componentes hardware se vuelve extremo y la energía necesaria para poner en marcha esto sistemas adquiere un importancia relevante. Un de las características que determina el

rendimiento de un sistema HPC son las políticas de job scheduling empleadas para asignar la CPU a los diferentes procesos que compiten por ella y, en general, por los recursos del sistema. Son estas políticas de asignación de job scheduling las que serán estudiadas en este TFG, analizando sus rendimientos.

[1] Sandia National Labs, Oak Ridge National Laboratory, New Mexico State University, Georgia Tech, and Auburn, Micron Technologies, The University of Maryland.

2.2 Objetivos

El objetivo principal de este proyecto es el de realizar una aplicación en Perl [Perl 1] que analice el comportamiento de las políticas de job scheduling [JSCH 1] aplicadas por un simulador de sistemas HPC [HPC 1]. Para ello la aplicación deberá obtener y procesar los resultados generados por el simulador SST-Simulator al someterle trabajos (jobs) para su planificación y ejecución. Para cumplir con este objetivo se diseña y dimensiona un plan de proyecto director que determine las tareas y duraciones de éstas para finalizarlo en los plazos que establece este TFG.

El plan de proyecto divide las tareas en tres fases:

Fase 1. Estudio del simulador SST.

Conjunto de tareas encaminadas a obtener unos conocimientos del simulador SST que nos permita avanzar en el desarrollo de las siguientes fases. Es necesario entender el entorno de ejecución del simulador, qué parámetros de entrada acepta, qué procesos se suceden y qué información es capaz de generar.

Fase 2. Preparación del entorno.

Son las tareas que deberán realizarse para disponer de todo el entorno necesario que permita la ejecución del simulador y que disponga de las herramientas y complementos necesarios para desarrollar la aplicación en Perl de análisis de las políticas de job scheduling.

Fase 3. Desarrollo de aplicación.

Conformada por las tareas más propias de un proyecto TIC basado en la metodología SDLC [Wiki 1], en donde se detallan las diferentes tareas que se deben abordar de forma secuencial para elaborar y mantener un Sistema de Información durante su ciclo de vida útil.

2.3 Resultados

Como resultado de la consecución del proyecto se obtendrá:

a) Documentación.

i. Memoria del TFG.

Se elaborará una memoria del TFG que incluirá todo el estudio previo comentado referente al simulador SST, así como las tareas de adecuación del entorno necesarias para disponer de las herramientas imprescindibles para el desarrollo de la aplicación Perl de análisis de Políticas de Job Scheduling.

Esta documentación también incluirá el análisis de todas aquellas tareas relacionadas con el desarrollo del ciclo de vida del proyecto, propias de la Fase 3; éstas son:

- Análisis de requerimientos.
- Diseño funcional.
- Diseño técnico.
- Resultado de pruebas.

Finalmente se presentará el análisis final comparativo de las diferentes métricas obtenidas por Schedlyzer sobre los datos proporcionados por SST-Simulator al aplicar las diferentes políticas de job scheduling.

ii. Presentación del TFG.

Se elaborará una presentación en Power Point que sintetizará el trabajo realizado y que permitirá obtener una visión general y completa de las tareas realizadas y del producto final obtenido.

b) Aplicación

Se desarrollará una aplicación en lenguaje de computación Perl que analizará los resultados de los procesos de las políticas de job scheduling del simulador SST-Simulator, proporcionando resultados que permitirán evaluar la idoneidad de éstas en un contexto de ejecución determinado.

La aplicación, que se denominará Schedlyzer (Scheduler Analyzer), se desarrollará estructurando su código en tres funciones básicas:

- Función de adquisición de información: Encargada de leer/incoporar los resultados generados por el simulador SST y crear las estructuras de datos precisas para su procesado.
- Función de procesado de información: Encargada de procesar la información generada por el simulador SST y de crear las estructuras de datos precisas para realizar una representación analítica de las mismas.
- Función de representación de la información: Cuya responsabilidad será la de generar una salida correctamente formateada de los datos obtenidos por la función de procesado que permita realizar una evaluación de las bondades de las políticas de job scheduling aplicadas.

2.4 Requerimientos de sistema

Para el desarrollo de este TFG se deberá disponer de los siguientes requerimientos básicos:

- Sistema computador o máquina virtual de 64 bits, con un sistema operativo UNIX-Like, como puede ser Linux [Wiki 2].

  Para este TFG se utilizará un sistema PC con Windows 7 de 64 bits (máquina host) con el sistema de virtualización VirtualBox [VBOX 1] corriendo un OS Linux Ubuntu de 64 bits.

- Simulador SST-Simulator instalado (versión 4.0 o superior).
- Una implementación de MPI [MPI 1] soportada por las librerías Boost [BOO 1].
- Las librerías Boost C++. Para las versiones de SST 4.0 o superior, la mínima versión deberá ser la 1.53.

En la siguiente URL se puede obtener un mayor detalle de los requerimientos así como de requisitos opcionales: https://www.sst-simulator.org/wiki/UserSSTBuildPrerequisites

- Intérprete de Perl.
- Para la salida gráfica de la información generada se deberá disponer instalado el módulo de Perl Chart::Clicker: http://search.cpan.org/~gphat/Chart-Clicker-2.88/lib/Chart/Clicker.pm

2.5 Planificación

En el siguiente diagrama Gannt se detalla, agrupadas por fases, la cronología de las tareas que deben ser abordadas para realizar este proyecto. Se trata de una primera aproximación que, durante el transcurso del TFG, podrá variar para adaptar/incorporar posibles imprevistos y/o contratiempos que se puedan dar.

En el diagrama también se puede observar las fechas de las diferentes entregas o hitos de los que consta este TFG así como los tiempos que deben emplearse para la elaboración de la presente Memoria y la Presentación.

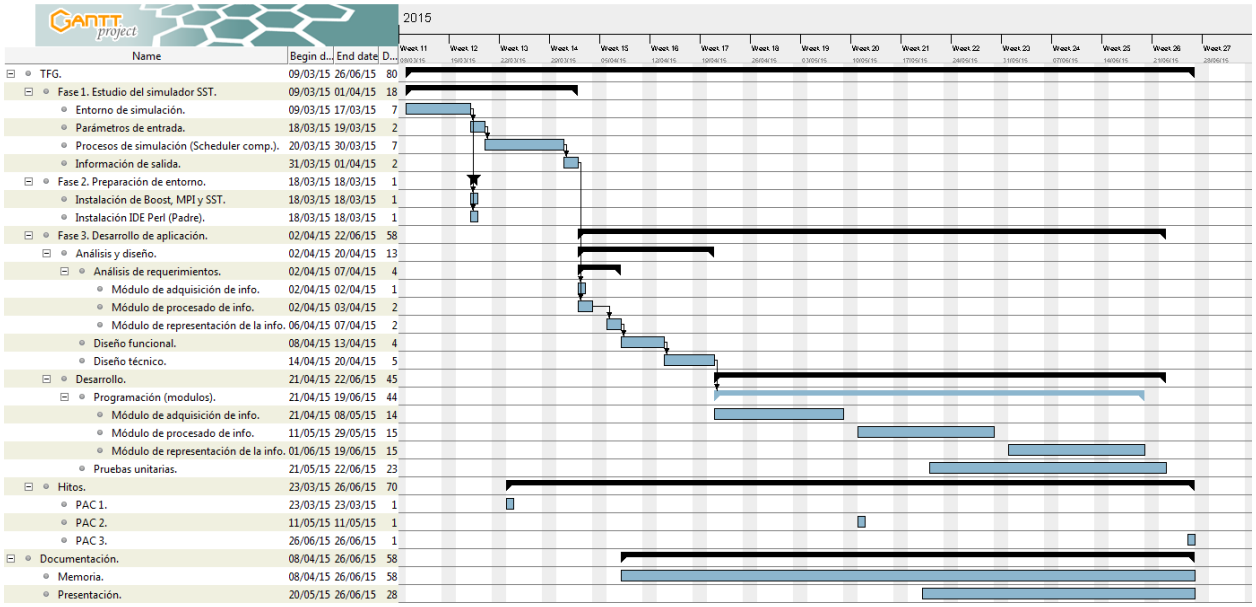


Fig 1. Cronograma Gannt de las tareas a realiza en el proyecto.

3 El simulador SST-Simulator

3.1 Introducción

El Structural Simulation Toolkit (SST) es un entorno de trabajo (framework) open source [OPEN 1] para la simulación de sistemas HPC. Dicho entorno permite definir los elementos necesarios, su interconexión y comportamiento, para emular la implementación de un sistema computador físico con el objetivo de estudiar/abordar nuevos retos en la construcción de sistemas HPC, como pueden ser: la escalabilidad, el rendimiento, la fiabilidad y el consumo energético.

SST proporciona un núcleo implementado mediante servicios[1] que permite integrar diferentes componentes e interconectarlos recreando la configuración real de un sistema HPC en el cual un componente puede interactuar con otro enviando eventos a través de links. Cada configuración de componentes integrados en el núcleo de SST constituye un modelo a simular.

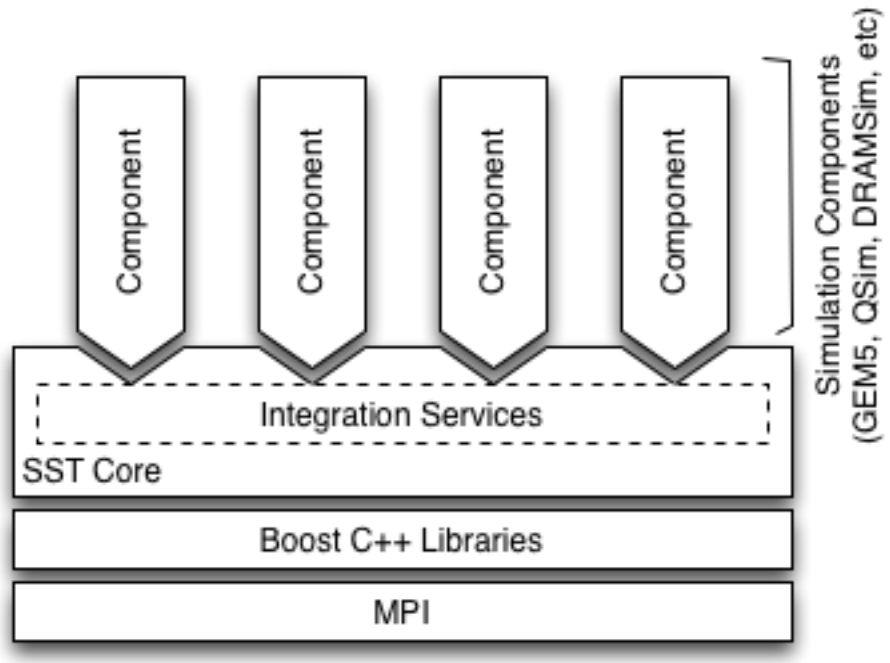


Fig 2. Arquitectura SST framework.

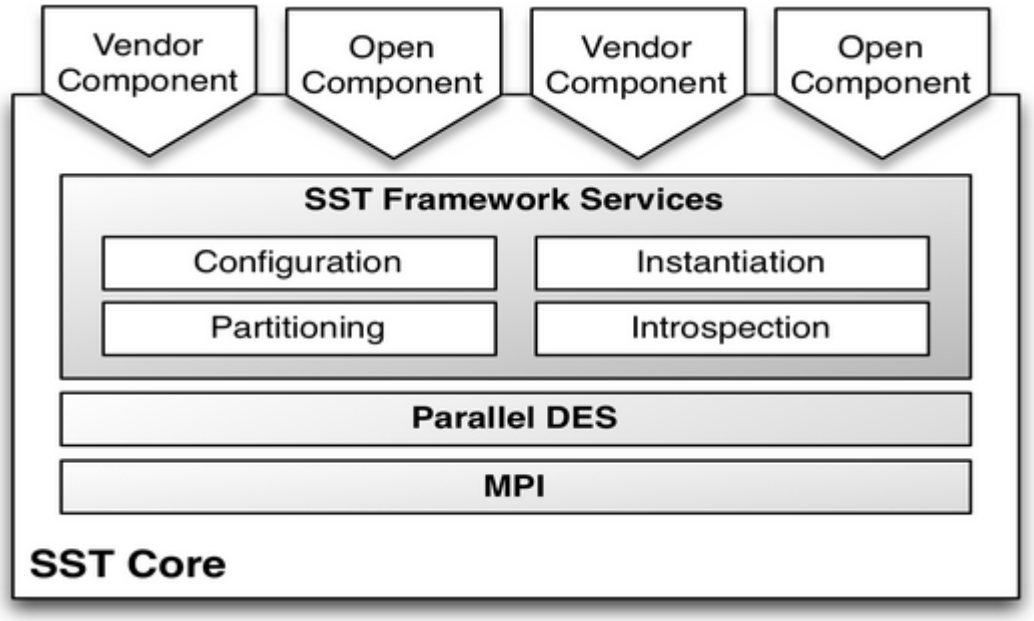


Fig 3. Núcleo de SST (detalle de servicios).

Para obtener un buen rendimiento, SST proporciona un entorno de ejecución paralelo basado en el modelo dinámico de sistemas Discrete Event Simulation (DES) [DES 1] implementado sobre MPI. Este entorno de ejecución paralelo le permite a SST abordar simulaciones de sistemas complejos. La paralelización es gestionada por el núcleo y es totalmente transparente para el desarrollador. Los eventos deben permitir ser serializados para ser enviado de un componente a otro y es responsabilidad del núcleo de SST gestionar si el destinatario de un mensaje se encuentra en su ámbito local o remoto.

[1] SST proporciona servicios para la comunicación, ejecución, sincronización y monitorización de componentes.

3.2 Configuración

SST permite ser configurado mediante ficheros XML en donde se describe el modelo a simular y su tipología. La especificación realizada en estos ficheros es denominada System Description Language (SDL). Básicamente SDL permite definir componentes y cómo estos se interconectan para formar un sistema computador en SST. Los tipos de componentes que pueden definirse son los siguientes:

Componentes de procesador: Dispone de varios componentes de procesadores que permiten simular varios modelos/arquitecturas. Por ejemplo el componente **Gem5**, permite ser configurado para simular arquitecturas Alpha, ARM, SPARC y x86.

Compoenentes de memoria: Dispone de varios componentes que permiten definir una arquitectura de memoria a implementar. Por ejemplo, **DRAMSIM2** permite definir modelos de controladores DDRx, canales y bancos de de memoria.

Componentes de red (network): Permite definir elementos de red y protocolos de comunicación entre elementos para simular el encaminamiento de información entre componentes. Por ejemplo, **Merlin** implemneta un router genérico y **Firefly** implementa un protocolo de bajo nivel para interconectar componentes de red como los definidos por Merlin.

Componentes de potencia: Al parecer son componentes en los que no se ha trabajado mucho pero a los que se le pretende dar más importancia en el futuro. Actualmente soporta la implementación de componentes Hotspot, IntSim, McPat (HP) y Orion.

Componentes System-Level: Se dispone del simulador **SST/Macro** para aplicaciones de gran tamaño y necesidades de alto rendimiento en computación paralela que permite la simulación de las actuales y futuras arquitecturas en Supercomputación [HPC 1].

La estructura interna de SDL se compone de las siguientes secciones otorgándole una estructura bien definida:

- **Sección de configuración:**

```
<config>
    run-mode=both
    partitioner=self
```

```
</config>
```

Cod 1. Sección de configuración.

Se informan los parámetros que serán utilizados por el entorno de simulación y que caracterizarán su ejecución. Estos parámetros son los mismos que los utilizados en la línea de comandos al invocar al simulador. A continuación se exponen dichos parámetros:

```
  -h [ --help ]         print help message
  -v [ --verbose ]      print information about core runtimes
  --no-env-config       disable SST automatic dynamic library environment
                        configuration
  -V [ --version ]      print SST Release Version
  --debug arg            { all | cache | queue  | clock | sync | link |
                                 linkmap | linkc2m | linkc2c | linkc2s | comp
                        | factory |                 stop | compevent | sim |
                        clockevent | sdl | graph }
  --debug-file arg      file where debug output will go
  --lib-path arg        component library path (overwrites default)
  --add-lib-path arg    add additional component library paths
  --run-mode arg        run mode [ init | run | both ]
  --stop-at arg         set time at which simulation will end execution
  --heartbeat-period arg Set time for heart beats to be published (these are
                        approximate timings published by the core to update
on
                        progress), default is every 10000 simulated seconds
  --timebase arg        sets the base time step of the simulation (default:
                        1ps)
  --partitioner arg     partitioner to be used < linear | roundrobin | self |
                        simple | single | lib.partitioner_name >
                        Descriptions:
```

```
                        -  linear: Partitions components by dividing
Component
                        ID space into roughly equal portions.  Components
with
                        sequential IDs will be placed close together.
                        -  roundrobin: Partitions components using a simple
                        round robin scheme based on ComponentID.  Sequential
                        IDs will be placed on different ranks.
                        -  self: Used when partitioning is already specified
                        in the configuration file.
                        -  simple: Simple partitioning scheme which attempts
                        to partition on high latency links while balancing
                        number of components per rank.
                        -  single: Allocates all components to rank 0.
                        Automatically selected for serial jobs.
                        -  lib.partitioner_name: Partitioner found in element
                        library 'lib' with name 'partitioner_name'
  --generator arg       generator to be used to build simulation
                        <lib.generator_name>
  --gen-options arg     options to be passed to generator function (must use
                        quotes if whitespace is present)
  --output-partition arg Dump the component partition to this file (default is
                        not to dump information)
  --output-config arg   Dump the SST component and link configuration graph
to
                        this file (as a Python file), empty string (default)
                        is not to dump anything.
  --output-dot arg      Dump the SST component and link graph to this file in
                        DOT-format, empty string (default) is not to dump
                        anything.
  --output-directory arg Controls where SST will place output files including
```

```
                          debug output and simulation statistics, default is
for
                          SST to create a unique directory.
  --model-options arg     Provide options to the SST Python scripting engine
                          (default is to provide no script options)
```

Cod 2. Parámetros permitidos en la sección de configuración.

- **Sección de definición de variables:**

  Permite informar variables que luego podrán ser utilizadas a lo largo del fichero y en aquellas sentencias que permitan trabajar con variables. A continuación se expone un ejemplo de asignación de variable:

```
<variables>
    <nic_link_lat> 200ns </nic_link_lat>
    <rtr_link_lat> 10ns </rtr_link_lat>
</variables>
```

Cod 3. Sección de variables.

Para referenciar el contenido de una variable ésta debe referenciarse precedida del carácter $, ejemplo: $<nic_link_lat>

- **Sección de parámetros:**

  Permite definir grupos de parámetros que posteriormente podrán ser asignados a componentes en su correspondiente sección de parámetros. Si un componente define explícitamente uno de estos parámetros, el parámetro homologo definido en el grupo será sobrescrito. A continuación se expone un ejemplo de la creación de grupos de parámetros:

```
<param_include>


    <rtr_params>
        <clock> 500Mhz </clock>
```

```
        <network.xDimSize> 2 </network.xDimSize>
        <network.yDimSize> 2 </network.yDimSize>
        <network.zDimSize> 2 </network.zDimSize>

        <routing.xDateline> 0 </routing.xDateline>
        <routing.yDateline> 0 </routing.yDateline>
        <routing.zDateline> 0 </routing.zDateline>
    </rtr_params>

    <cpu_params>
        <radix> 2 </radix>
        <nodes> 8 </nodes>
        <msgrate> 5MHz </msgrate>
        <xDimSize> 2 </xDimSize>
        <yDimSize> 2 </yDimSize>
        <zDimSize> 2 </zDimSize>
        <application> allreduce.tree_triggered </application>
        <latency> 500 </latency>
    </cpu_params>

    <nic_params>
        <clock> 500Mhz </clock>
        <timing_set> 2 </timing_set>
    </nic_params>

</param_include>
```

Cod 4. Sección de parámetros.

- **Sección de modelado del simulador:**

  En esta sección se definen todos los componentes que conforman el modelo a simular y como están interconectados a través de links. Para ello, se define la sección que deberá presentar la siguiente estructura:

  <sst>

  <component> … </component>

  <component> … </component>

  <component> … </component>

  …

  </sst>

  Un componente es definido en SDL estructurado en tres partes:

  - Componente.

```
<component name="name" type="type" rank="rank">
```

Cod 5. Definición de un componente.

Define el nombre, el tipo y el rango MPI a asignar al componente. Esta asignación de rango[1] tiene el objetivo de realizar un balanceo de carga dependiendo de la latencia de sus links, considerando que los rangos con alta latencia tendrán una menor frecuencia de envío de mensajes.

Existen muchos tipos de componentes que el simulador SST proporciona y es capaz de integrar (*internal components*) a la vez que es posible desarrollar nuevos componentes customizados (*external components*) e integrarlos, algunos bajo licencia SST.

Algunos componentes fueron desarrollados para ser ejecutados como simuladores independientes e interactúan con el núcleo de SST para comunicarse con otros componentes. Estos modelos de componentes se clasifican de la siguiente forma: de procesador, de memoria, de network, de energía y componentes de nivel de sistema (ver inicio de esta sección).

[1] Esta partición es realizada mediante la librería Zoltan [ZOLT 1].

- Parámetros.

```
<params include=include1,include2,...>
    <param1> value1 </param1>
    <param2> value2 </param2>
    ...
</params>
```

Cod 6. Sección de parámetros de un componente.

Permite definir parejas de parámetros nombre-valor que serán utilizadas por el componente y que determinarán su comportamiento. También es posible incluir una lista de parámetros si previamente ha sido definida en la sección de parámetros anteriormente comentada.

- Links.

```
<link name="name" port="port" latency="latency"/>
```

Cod 7. Definición de un link de un componente.

Define cómo los puertos de cada componente están intercomunicados con el entorno de simulación. Un enlace es compartido por dos componentes one-to-one, siendo la comunicación biderccional y quedando la información referente a esta

conexión repartida entre los dos componentes. En la declaración del *link* se define una latencia (latency) y se refierea a la latencia que presenta el componente a la hora de enviar un mensaje por dicho puerto.

Ejemplo de la declaración de un componente con todas sus secciones comentadas:

```
<component name=name type=type rank=rank>
    <params include=include_list>
        <param1> value1 </param1>
        <param2> value2 </param2>
    ...
    </params>
    <link name=link1 port=port1 latency=10ns/>
    <link name=link2 port=port2 latency=$port2_lat/>
    ...
</component>
```

Cod 8. Ejemplo de integración de un componente.

3.3 El componente Scheduler

Dentro de los componentes internos que fueron desarrollados para comportarse como un simulador por sí solos se encuentra el componente Scheduler. Este componente está basado en un simulador HPC que permite monitorizar y obtener información a bajo nivel de cuándo un job pasa a ejecutarse y a qué procesador ha sido asignado, hasta que es finalizado.

Este componente es capaz de simular 5 modelos de job scheduling mediante la especialización de 2 componentes:

- schedComponent: Para la gestión de scheduling, allocation y machine models.
- nodeComponent: Para la gestión de los nodos y fallos.

El simulador obtiene, mediante la carga paralela de un fichero (ver: Workloads), las características de los trabajos que deberá analizar, gestionando un flujo de llegadas y aplicando la política de job scheduling configurada en un parámetro de componente, hasta su terminación. Para ello utiliza varias clases de su modelo interno (Scheduler, Allocator y Machine) para registrar estados y tomar decisiones.

Varias son las políticas de job scheduling y allocators que han sido implementadas mediante este componente para formar la base de análisis de futuros estudios; éstas son:

- EASY (*EASYScheduler*) [EASY 1], basado en EASY-Backfilling (ver: Backfilling policies).
- Una generalización de Conservative Backfilling (*StatefulScheduler*).
- Priority-based scheduler (*PQScheduler*).

Las siguientes son implementaciones que están pensadas para su ejecución en mesh systems [MESYS 1] y son derivadas de la clase Allocator:

- Basadas en linear orderings (*LinearAllocator*).
- Basadas en center-based allocators (*NearestAllocator*).
- basadas en buddy systems (*MBSAllocator*).

Para las simulaciones que pretenden obtener un buen rendimiento y no es necesario especificar el modo en que el job es asignado a un procesador (allocation), siendo el mecanismo de planificación el que adquiere el foco de análisis, es posible especificar un sistema sin noción de localidad, *SimpleMachine*, con su correspondiente allocator, *SimpleAllocator*.

El siguiente diagrama resume las clases comentadas:

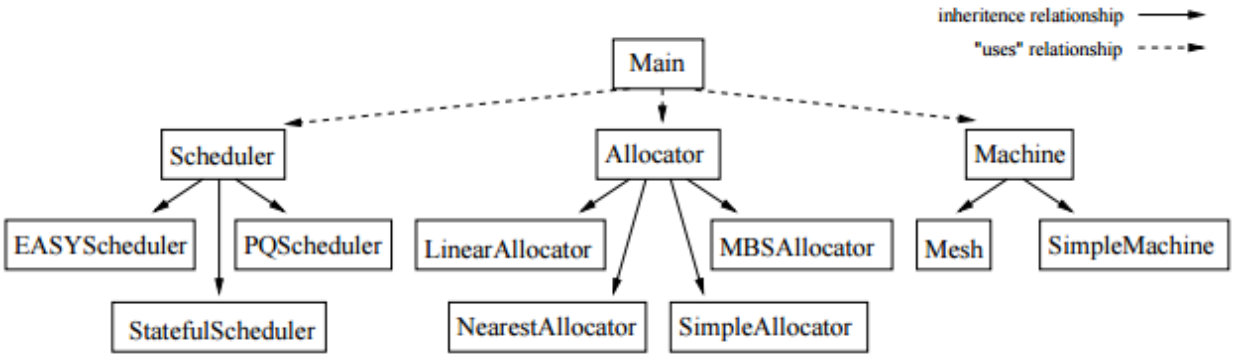


Fig 4. Relación de clases de la implementación de job scheduler.

A continuación se expone un ejemplo de la especificación de un modelo de job scheduler mediante SDL:

```
<?xml version="1.0"?>

<sdl version="2.0"/>


<config>

    run-mode=both

    partitioner=self

</config>


<sst>

  <component name="s" type="scheduler.schedComponent" rank=0>

    <params>

           <traceName>test_scheduler_0001.sim</traceName>

           <scheduler>easy</scheduler>

           <machine>mesh[16,8]</machine>

           <allocator>sortedfreelist</allocator>

           <FST>none</FST>

           <timeperdistance>0</timeperdistance>

    </params>

   <link name="l0" port="nodeLink0" latency="0 ns"/>

   <link name="l1" port="nodeLink1" latency="0 ns"/>

   <link name="l2" port="nodeLink2" latency="0 ns"/>

           .

           .
```

```
        .
 <link name="l125" port="nodeLink125" latency="0 ns"/>
 <link name="l126" port="nodeLink126" latency="0 ns"/>
 <link name="l127" port="nodeLink127" latency="0 ns"/>
</component>

<component name="n0" type="scheduler.nodeComponent" rank=0>
  <params>
     <nodeNum>0</nodeNum>
  </params>
  <link name="l0" port="Scheduler" latency="0 ns"/>
</component>
<component name="n1" type="scheduler.nodeComponent" rank=0>
  <params>
     <nodeNum>1</nodeNum>
  </params>
  <link name="l1" port="Scheduler" latency="0 ns"/>
</component>
        .
        .
        .
<component name="n126" type="scheduler.nodeComponent" rank=0>
  <params>
     <nodeNum>126</nodeNum>
  </params>
  <link name="l126" port="Scheduler" latency="0 ns"/>
```

```
  </component>
  <component name="n127" type="scheduler.nodeComponent" rank=0>
    <params>
       <nodeNum>127</nodeNum>
    </params>
    <link name="l127" port="Scheduler" latency="0 ns"/>
  </component>
</sst>
```

Cod 9. Ejemplo de especificación de un job scheduler.

Los anteriores schedulers adminten una serie de comparadores para seleccionar entra varias políticas de planificación, como pueden ser: FCFS (First-come, first-served), Shortest Job First, Widest Job First y algoritmos similares basados priority-based schemes; son los siguinentes:

- fifo - first in, first out.
- largefirst - job with most required nodes first.
- smallfirst - job with fewest required nodes first.
- longfirst - job with longest estimated running time first.
- shortfirst - job with shortest estimated running time first.
- betterfit - job with most processors first, if tied longest estimated running time, if tied first arrival time.

Ejemplo de especificación de algoritmo FCFS (sin aplicar backfilling):

```
<component name="s" type="scheduler.schedComponent" rank=0>
    <params>
          <traceName>test_scheduler_0001.sim</traceName>
          <scheduler>pqueue[fifo]</scheduler>
          <machine>simple</machine>
          <allocator>simple</allocator>
```

```
            <FST>none</FST>
            <timeperdistance>0</timeperdistance>    </params>
   <link name="l0" port="nodeLink0" latency="0 ns"/>
   <link name="l1" port="nodeLink1" latency="0 ns"/>
   <link name="l2" port="nodeLink2" latency="0 ns"/>
            .
            .
            .
   <link name="l125" port="nodeLink125" latency="0 ns"/>
   <link name="l126" port="nodeLink126" latency="0 ns"/>
   <link name="l127" port="nodeLink127" latency="0 ns"/>
</component
```

Cod 10. Ejemplo de especificación de algoritmo FCFS..

3.4 Workloads

Dentro del universo del análisis de políticas de job scheduling en sistemas HPC, existe la necesidad de disponer de datos que puedan ser utilizados como entrada de los procesos de simulación. Para ello, existen organizaciones, principalmente Universidades, que en su labor de investigación en áreas tales como la Supercomputación o, High Performance Computing, recopilan y mantienen archivos de logs generados por sistema reales en entornos productivos. Estos archivos contienen información referente a los trabajos que son gestionados por un determinado sistema indicando, por cada unos de ellos, datos referentes a su ejecución.

Uno de los beneficios directos, que se obtiene del uso de estos archivos por la comunidad científica y universitaria, es la posibilidad de contrastar los resultados obtenidos con el objetivo de ratificar y fortalecer los diferentes estudios en los que se utiliza.

Actualmente, uno de los repositorios más importantes de este tipo de archivos es el *Parallel Worklod Archive* [PWA 1]. En dicho repositorio es posible encontrar, los ficheros de

logs originales, generados por los sistemas Supercomputador en entono productivo y, la adaptación aplicada a los mimos para dejarlos en formato .swf (Standard Workload Format). El SWF es un formato estándar para facilitar el uso de este tipo de archivos de logs, eliminando y/o formateando los datos de los jobs de los archivos originales. La información de los archivos SWF está agrupada en dos grandes grupos:

Header Comments: Define los aspectos globales del workload así como el contexto en el que se ejecutaron los jobs.

***1. Version**: Version number of the standard format the file uses. The format described here is version 2.*
***2. Computer**: Brand and model of computer*
***3. Installation**: Location of installation and machine name*
***4. Acknowledge**: Name of person(s) to acknowledge for creating/collecting the workload.*
***5. Information**: Web site or email that contain more information about the workload or installation.*
***6. Conversion**: Name and email of whoever converted the log to the standard format.*
***7. MaxJobs**: Integer, total number of jobs in this workload file.*
***8. MaxRecords**: Integer, total number of records in this workload file. If no checkpointing/swapping information is included, there is one record per job, and this is equal to MaxJobs. But with chekpointing/swapping there may be multiple records per job.*
***9. Preemption**: Enumerated, with four possible values. 'No' means that jobs run to completion, and are represented by a single line in the file. 'Yes' means that the execution of a job may be split into several parts, and each is represented by a separate line. 'Double' means that jobs may be split, and their information appears twice in the file: once as a one-line summary, and again as a sequence of lines representing the parts, as suggested above. 'TS' means time slicing is used, but no details are available.*
***10. UnixStartTime**: When the log starts, in Unix time (seconds since the epoch)*
***11. TimeZone**: **DEPRECATED and replaced by TimeZoneString**.*
A value to add to times given as seconds since the epoch. The sum can then be fed into gmtime (Greenwich time function) to get the correct date and hour of the day. The default is 0, and then gmtime can be used directly. Note: do not use localtime, as then the results will depend on the difference between your time zone and the installation time zone.
***12. TimeZoneString**: Replaces the buggy and now deprecated TimeZone.*
TimeZoneString is a standard UNIX string indicating the time zone in which the log was generated; this is actually the name of a zoneinfo file, e.g. ``Europe/Paris''. All times within the SWF file are in this time zone. For more details see the usage note below.

***13. StartTime**: When the log starts, in human readable form, in this standard format: Tue Feb 21 18:44:15 IST 2006 (as printed by the UNIX 'date' utility).*
***14. EndTime**: When the log ends (the last termination), formatted like StartTime.*
***15. MaxNodes**: Integer, number of nodes in the computer. List the number of nodes in different partitions in parentheses if applicable.*
***16. MaxProcs**: Integer, number of processors in the computer. This is different from MaxNodes if each node is an SMP. List the number of processors in different partitions in parentheses if applicable.*
***17. MaxRuntime**: Integer, in seconds. This is the maximum that the system allowed, and may be larger than any specific job's runtime in the workload.*
***18. MaxMemory**: Integer, in kilobytes. Again, this is the maximum the system allowed.*
***19. AllowOveruse**: Boolean. 'Yes' if a job may use more than it requested for any resource, 'No' if it can't.*
*20. **MaxQueues**: Integer, number of queues used.*
***21. Queues**: A verbal description of the system's queues. Should explain the queue number field (if it has known values). As a minimum it should be explained how to tell between a batch and interactive job.*
***22. Queue**: A description of a single queue in the following format: queue-number queue-name (optional-details). This should be repeated for all the queues.*
***23. MaxPartitions**: Integer, number of partitions used.*
***24. Partitions**: A verbal description of the system's partitions, to explain the partition number field. For example, partitions can be distinct parallel machines in a cluster, or sets of nodes with different attributes (memory configuration, number of CPUs, special attached devices), especially if this is known to the scheduler.*
***25. Partition**: Description of a single partition.*
***26. Note**: There may be several notes, describing special features of the log. For example, ``The runtime is until the last node was freed; jobs may have freed some of their nodes earlier''.*

Tabla 1. Sección *header comments* de un archivo SWF.

The Data Fields: Cada línea se corresponde con un job aportando toda la información referente a su ejecución por el sistema.

***1. Job Number** -- a counter field, starting from 1.*
***2. Submit Time** -- in seconds. The earliest time the log refers to is zero, and is usually the submittal time of the first job. The lines in the log are sorted by ascending submittal times. It makes sense for jobs to also be numbered in this order.*

3. Wait Time *-- in seconds. The difference between the job's submit time and the time at which it actually began to run. Naturally, this is only relevant to real logs, not to models.*
4. Run Time *-- in seconds. The wall clock time the job was running (end time minus start time).* *We decided to use ``wait time'' and ``run time'' instead of the equivalent ``start time'' and ``end time'' because they are directly attributable to the scheduler and application, and are more suitable for models where only the run time is relevant.* *Note that when values are rounded to an integral number of seconds (as often happens in logs) a run time of 0 is possible and means the job ran for less than 0.5 seconds. On the other hand it is permissable to use floating point values for time fields.*
5. Number of Allocated Processors *-- an integer. In most cases this is also the number of processors the job uses; if the job does not use all of them, we typically don't know about it.*
6. Average CPU Time Used *-- both user and system, in seconds. This is the average over all processors of the CPU time used, and may therefore be smaller than the wall clock runtime. If a log contains the total CPU time used by all the processors, it is divided by the number of allocated processors to derive the average.*
7. Used Memory *-- in kilobytes. This is again the average per processor.*
8. Requested Number of Processors*.*
9. Requested Time*. This can be either runtime (measured in wallclock seconds), or average CPU time per processor (also in seconds) -- the exact meaning is determined by a header comment. In many logs this field is used for the user runtime estimate (or upper bound) used in backfilling. If a log contains a request for total CPU time, it is divided by the number of requested processors.*
10. Requested Memory *(again kilobytes per processor).*
11. Status *1 if the job was completed, 0 if it failed, and 5 if cancelled. If information about chekcpointing or swapping is included, other values are also possible.* ***See usage note below.*** *This field is meaningless for models, so would be -1.*
12. User ID *-- a natural number, between one and the number of different users.*
13. Group ID *-- a natural number, between one and the number of different groups. Some systems control resource usage by groups rather than by individual users.*
14. Executable (Application) Number *-- a natural number, between one and the number of different applications appearing in the workload. in some logs, this might represent a script file used to run jobs rather than the executable directly; this should be noted in a header comment.*
15. Queue Number *-- a natural number, between one and the number of different queues in the system. The nature of the system's queues should be explained in a header comment. This field is where batch and interactive jobs should be differentiated: we suggest the convention of denoting interactive jobs by 0.*

16. Partition Number *-- a natural number, between one and the number of different partitions in the systems. The nature of the system's partitions should be explained in a header comment. For example, it is possible to use partition numbers to identify which machine in a cluster was used.*
17. Preceding Job Number *-- this is the number of a previous job in the workload, such that the current job can only start after the termination of this preceding job. Together with the next field, this allows the workload to include feedback as described below.*
18. Think Time from Preceding Job *-- this is the number of seconds that should elapse between the termination of the preceding job and the submittal of this one.*

Tabla 2. Sección *data fields* de un archivo SWF.

Los ficheros de log, antes de ser usados en el simulador SST-Simulator, deben ser adaptados al formato de entrada que éste requiere para la simulación de políticas de job scheduling. El formato final del fichero de log a utilizar es el siguiente:

2. Submit Time *-- in seconds. The earliest time the log refers to is zero, and is usually the submittal time of the first job. The lines in the log are sorted by ascending submittal times. It makes sense for jobs to also be numbered in this order.*
5. Number of Allocated Processors *-- an integer. The number of processors the job uses.*
4. Run Time *-- in seconds. The wall clock time the job was running (end time minus start time).*
9. Requested Time*. Rruntime (measured in wallclock seconds). In many logs this field is used for the user runtime estimate (or upper bound) used in backfilling.*

Tabla 3. Formato de un archivo SIM.

Este formato de fichero final es el que será especificado en los ficheros SDL utilizados por el SST-Simulator (ver: Configuración y El componente Scheduler). En este caso trabajaremos con el fichero denominado SDSC-SP2-1998-4.2-cln.sim.

En la sección Diseño técnico se presenta un esquema del sistema implicado en este TFG que facilita la compresión de lo comentado en este punto.

3.4.1 The San Diego Supercomputer Center (SDSC) SP2 log

Para la realización de este TFG se ha seleccionado, como archivo SWF para usar en el simulador SST-Simulator, la versión *cleaned* del San Diego Supercomputer Center (SDSC) SP2 Log: SDSC-SP2-1998-4.2-cln.swf. Este log ha sido generado por un sistema IBM SP2 con 128 nodos. Contiene información de los trabajos sometidos durante dos años (May 1998 thru April 2000) registrando un total 73.496 jobs, para la versión original, quedando un total de 59.715 jobs para la versión *cleaned* utilizada; aunque finalmente, para generar el fichero .time requerido por SST-Simulator y especificado en el SDL, se ha obtenido un total final de 54.034 jobs.

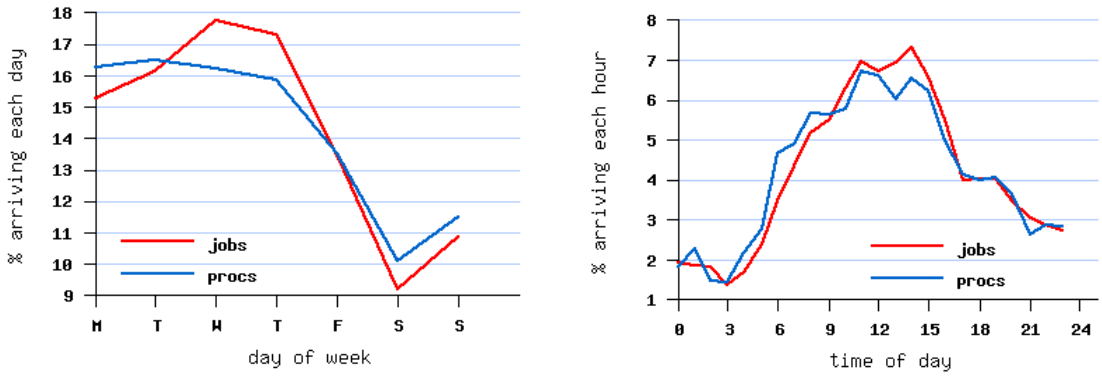
Este log proporciona información detallada referente a los tiempos de ejecución de un job, como pueden ser el *submit time*, *wait time* o el *run time*, así como de información referente a los recursos de sistema utilizados y auditoría, como pueden ser el *requested number of processors*, *user ID* o el *queue number* (ver: Workloads).

Se ha seleccionado este log como base para nuestras pruebas porque ha sido generado por un sistema con una arquitectura de 128 nodos, sin utilizar preemption (un job inicia su ejecución y finaliza hasta completarla sin ser interrumpido) [PREE 1], igual a la que hemos modelado mediante SST-Simulator y para la que se pretende analizar el comportamiento de las políticas de planificación de procesos para jobs de cálculo masivo (no interactivo).

Algunas de las características que pueden extraerse de este log y que definen cómo se comportó el sistema IBM SP2 durante el periodo de registro de este log, son las que se muestran en las siguientes gráficas (versión *cleaned*):

Nota: Todas las gráficas mostradas en esta sección han sido obtenidas de la Web Parallel Workloads Archive [PWA 1].

- A continuación se muestran gráficas de los porcentajes de llegada de trabajos en el sistema por día y por hora.



(a) Llegada de jobs por día. (b) Llegada de jobs por hora.

Fig 5. Ratio de llegada en SDSC workload.

En ellas se puede observar cómo la mayor parte de los trabajos, entran en el sistema entre el lunes y el jueves, alrededor de unos 20 por día, siendo los miércoles cuando se recibe el máximo de ellos. También es posible observar como es en la franja horaria de 09h a 15h cuando más actividad de trabajo se produce en el sistema.

- En las siguientes gráficas se puede identificar los requerimientos de procesador y el nivel de paralelismo entre procesos.

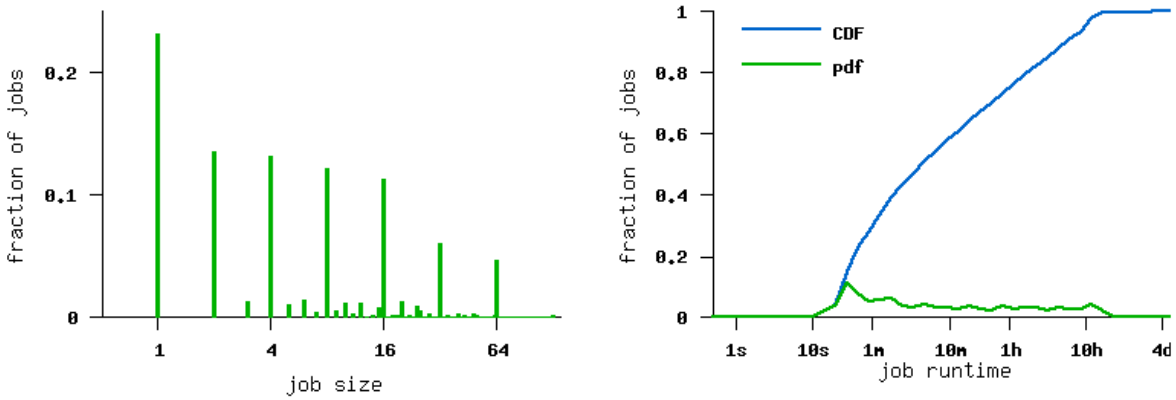


Fig 6. *Job size* y *job runtime* en SDSC workload.

Como se puede observar la mayoría de jobs requieren de 1 sólo procesador, repartiéndose el resto de porcentajes entre los que necesitan 2, 4, 8, 16 procesadores y, en una menor proporción, los que necesitan 32 y 64; siendo casi despreciable el requerimiento de 100 y 128 procesadores.

También es posible apreciar que el 40% de los jobs necesitan más de 10 minutos para completar su ejecución.

- La siguiente gráfica muestra la relación de recursos de CPU requeridos y tiempo de runtime empleado.

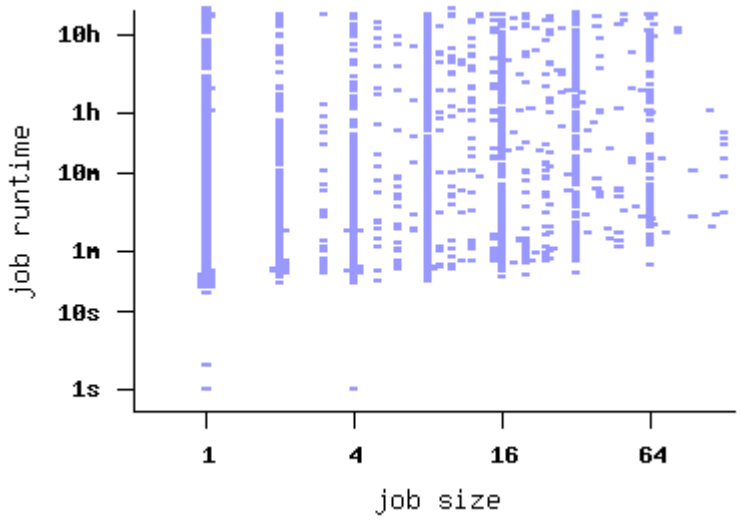


Fig 7. Scattered plot of runtime and job size en SDSC workload.

En ella puede apreciarse como los jobs que requieren de un única CPU, que como acabamos de ver son los que tienen más presencia en el sistema, presentan unos tiempos de ejecución muy variados, llegando a superar las 10h de ejecución. Vemos como conforme aumenta el requerimiento de CPU's, los tiempos de runtime más pesados disminuye, obteniéndose una dispersión más o menos uniforme.

4 Políticas de Job Scheduling

4.1 El Scheduler

El scheduler (planificador) es el mecanismo mediante el cual el sistema operativo es capaz de determinar cuál es el próximo job (proceso) a asignar el procesador. Pare ello, el scheduler se vale de una serie de algoritmos que le permite seleccionar qué proceso es el más adecuado en cada momento para maximizar el uso de la(s) CPU(s). Estos algoritmos de selección conforman las denominadas políticas de job scheduling.

El scheduler tiene como objetivo optimizar las siguientes características inherentes al uso de la CPU:

- **Disponibilidad**: Todo proceso debe poder acceder al procesador en algún momento, no puede quedar esperando indefinidamente.
- **Repartimiento**: Se debe intentar maximizar el número de procesos a los que le son asignados el procesador por unidad de tiempo.
- **Equilibrio**: Deberá intentar priorizar procesos que utilizan recursos poco utilizados.
- **Aprovechamiento**: El scheduler deberá intentar disponer siempre de procesos para ser ejecutados, aprovechando al máximo el procesador.
- **Eficiencia**: Se debe minimizar el tiempo en que el procesador está ejecutando código del scheduler, dejando de ejecutar código de usuario.
- **Interactividad**: Se debe priorizar los procesos interactivos para dar respuestas rápidas a los usuarios y porque el tiempo de usuario es más caro que el tiempo de máquina.

Para conseguir estos objetivos, el scheduler se divide en tres niveles de planificación, estos son: planificación a largo, medio y corto plazo.

Para la realización del presente TFG nos centraremos en el nivel de planificación a corto plazo.

4.2 Algoritmos de planificación

4.2.1 La planificación a corto plazo

El planificador a corto plazo es el responsable de seleccionar el próximo proceso a ejecutar y, por tanto, pasarlo a estado de 'ejecución' [PSTATE 1], de entre los que se encuentran en estado 'preparado'. Esta selección es realizada por el planificador cada vez que se produce un cambio de contexto, indicando que un proceso abandona la CPU y que, por tanto, se requiere una nueva asignación. De esta forma el planificador a corto plazo puede ser invocado múltiples veces por segundo. Es por esto que planificador debe ser muy eficiente al realizar los cambios de contexto para no penalizar en rendimiento. Por ello, estos planificadores –también llamados dispatchers- suelen estar desarrollados en lenguaje máquina y en algunas arquitecturas de microprocesadores suelen implementarse instrucciones que simplifican su programación.

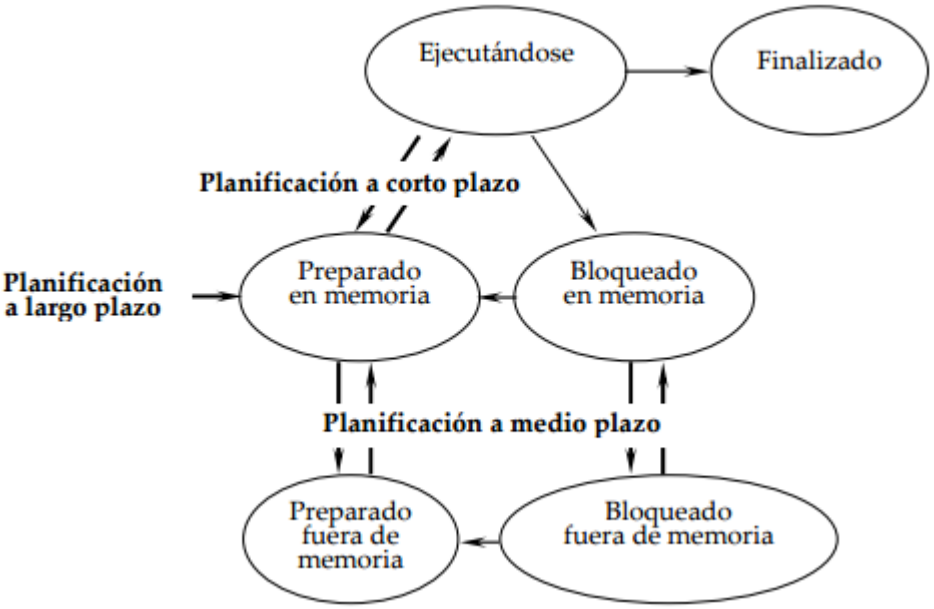


Fig 8. Diagrama de estados de procesos y lugar en dónde actúa la planificación a corto plazo.

Según el modo en el que se produce el cambio de contexto estos algoritmos se clasifican de la siguiente forma:

- **No apropiativos**: Cuando un proceso finaliza su racha de CPU [RACH 1] independientemente de su duración y del resto de proceso que compiten por la CPU.
- **Apropiativos**: El proceso puede ser interrumpido para realizar un cambio de contexto antes de finalizar su racha de CPU.

  Dentro de los apropiativos se encuentra la siguiente subclasificación:

  - **Apropiación inmediata**: El algoritmo de selección es invocado cada vez que un proceso se desbloquea y, se produce un cambio de contexto para que éste pase a ejecutarse si dispone de mayor prioridad.
  - **Apropiación diferida**: El algoritmo es invocado cuando finaliza las rachas de CPU y a periodos regulares de tiempo. Estas invocaciones periódicas permiten evaluar si en ese momento se encuentra entre los procesos candidatos –estado 'preparado'- alguno, por ejemplo, con mayor prioridad y, por tanto, candidato a ser asignado a la CPU, antes de que el actual finalice su racha de CPU.

A continuación se describen los algoritmos de planificación a corto plazo que suelen implementar los sistemas operativos.

4.2.1.1 Algoritmo de orden de llegada (FCFS - first come, first served)

El scheduler ordena los procesos por orden de llegada en una cola de 'preparados'. Estos son seleccionados para su ejecución cuando el proceso que actualmente dispone de la CPU finaliza su racha de CPU. Por tanto, este algoritmo es no apropiativo.

El FCFS ofrece bajos rendimientos y, al depender de la finalización de las rechas de CPU de los procesos en ejecución, ofrece un comportamiento con tiempos de retorno muy variables, tendiendo a penalizar a los procesos pequeños, que permanecen por más tiempo en estado 'preparado' esperando que procesos con mayor racha de CPU finalicen su ejecución. Como ventaja, comentar que su implementación es muy sencilla.

4.2.1.2 Algoritmo de menor tiempo primero (SJF - shortest job first)

El Schedule intenta ordenar los procesos según la duración de la siguiente racha de CPU que se disponen a ejecutar. De esta forma se intenta no penalizar a los procesos pequeños como pasa con el algoritmo FCFS, presentando un mejor rendimiento. La dificultad de este algoritmo estriba en poder determinar la duración de la durada de la siguiente racha de CPU de los procesos candidatos a asignar la CPU (preparados). Esto generalmente lo realiza mediante el cálculo de una estimación dependiendo de la durada de sus rachas de CPU de sus ejecuciones anteriores. En el caso de que varios procesos presentasen un empate, entonces el Schedule aplicaría el algoritmo FCFS.

La siguiente podría ser una fórmula para la estimación de cada proceso:

$$e_{n+1} = \alpha \cdot r_n + (1 - \alpha) \cdot e_n$$

Donde,

r_n: Duración real de la última racha de CPU.

e_n: Valor de la última estimación realizada de duración de racha de CPU.

α: Factor que toma valores entre 0 y 1.

Este algoritmo es invocado cada vez que un proceso finaliza su racha de CPU, siendo por tanto no apropiativo.

4.2.1.3 Algoritmo de menor tiempo pendiente (SRTF - shortest remaining time first)

Este algoritmo es invocado cada vez que un proceso debe ser añadido a la cola de procesos preparados, sin tener en cuenta si la racha de CPU del proceso que actualmente está ejecutándose ha finalizado. Por tanto, este algoritmo es apropiativo. Cada vez que este algoritmo es invocado, intenta determinar si la duración de la próxima racha de CPU del proceso que se acaba de añadir a la cola de preparados es menor que el resto de racha de CPU del proceso en ejecución. Si esta condición resultase cierta, se produciría el cambio de contexto.

Como los anteriores algoritmos vistos (FCFS, SJF), éste también debe de implementar una fórmula que determine la duración de las rachas de CPU. Debido a la condición impuesta por este algoritmo, se podría tener jobs que nunca pasasen a estado de ejecución por disponer de una racha de CPU relativamente mayor al resto. Por tanto, este algoritmo presenta el problema de la inanición (starvation) [STARV 1].

4.2.1.4 Algoritmo Prioritario

Este algoritmo asigna una prioridad a cada uno de los procesos que gestiona. Estas prioridades pueden ser de dos naturalezas:

- **Prioridades internas**: Son las que hacen referencia a propiedades inherentes a los procesos y que son medibles por el OS, como pueden ser: las rachas de CPU, tiempo de procesador consumido, tiempo transcurrido en estado preparado, etc.
- **Propiedades externas**: Son las que hace referencia a propiedades no atribuibles a propiedades inherentes de los procesos y que afectan a otras capacidades del sistema, como pueden ser: el usuario responsable del proceso, prioridades impuestas por dispositivos externos (es más importante comprobar si se ha activado una determinada alarma que comprobar si la máquina de café está en funcionamiento).

El algoritmo asignará el procesador a aquel proceso que tenga la prioridad más alta y, en caso de empate entra varios procesos, aplicará el algoritmo FCFS. La forma en la que se realiza el cambio de contexto puede ser tanto apropiativa como no apropiativa. En la apropiativa, cuando un proceso entra en la cola de 'preparados' se invoca este algoritmo y, si el proceso que está ejecutando tiene menos prioridad que el recién encolado, se produce el cambio de contexto pasando éste último a estado de ejecución.

Este algoritmo, al igual que el SRTF, puede provocar el problema de la inanición si los procesos que van entrando en la cola de preparados disponen de más prioridad que algún otro con prioridad más baja. Para estos casos existe una solución posible, la del envejecimiento (aging) [AGING 1], que permite ir incrementando la prioridad de un proceso paulatinamente hasta provocar su ejecución.

4.2.1.5 Algoritmo de repartimiento de tiempo (RR - round robin)

Pensado para sistemas de tiempo compartido, consiste en un algoritmo que reparte el tiempo de ejecución de la CPU en tiempos iguales entre todos los procesos de la cola de preparados. Este tiempo de ejecución, que es asignado a todos los procesos, se conoce con el nombre de *quantum*. Por tanto, los tiempos de ejecución son asignados de forma regular independientemente de la duración de las rachas de CPU.

Este algoritmo se basa en el FCFS aplicando apropiación diferida, es decir, cada *quantum* de tiempo, mediante una rutina de atención al temporizador o, al finalizar la racha de CPU, el algoritmo es invocado seleccionando el siguiente proceso de la cola de preparados que más tiempo lleve esperando (por orden de llegada).

4.3 Consideraciones sobre las políticas de job scheduling en sistemas HPC

En la realización del presente estudio se han considerado unas premisas que son inherentes a la forma de gestionar los trabajos en sistemas HPC. Las siguientes premisas han determinado la selección de las políticas de job scheduling analizadas en este estudio:

- Los jobs contemplados por el sistema, una vez iniciado el estado de ejecución, nunca son interrumpidos y sólo dejan este estado al finalizar completamente su ejecución (run-to-completion), es decir, no se produce el *context switching*, eliminando el *overead* derivado de éste.
- En SST-Simulator es posible especificar, como dato asociado a los jobs que figuran en cada una de las líneas de un workload, el tiempo estimado de su ejecución; siendo este dato proporcionado por el usuario.
- El número de procesadores necesarios para ejecutar un determinado job es un dato fijo asociado en el momento de ser sometido en el sistema.

Como consecuencia de las anteriores premisas, algunas políticas de job scheduling descritas en Algoritmos de Planificación no pueden ser simuladas; como es el caso de los algoritmos apropiativos (SRTF), el algoritmo Prioritario mediante ponderación o el Round Robin mediante

la aplicación de un *quantum* de tiempo (estos datos no son gestionados por SST-Simulator). Por otra parte, será posible la simulación de un algoritmo basado en EASY [EASY 1] el cual aplica la técnica de backfilling basándose en un contexto determinado:

- Número de nodos disponibles.
- El tiempo de finalización de los trabajos actualmente en ejecución (basado en el tiempo estimado de ejecución asociado a cada job).
- Características de los jobs en espera de ejecución.

4.3.1 Backfilling policies

Backfilling policies es una de las técnicas de planificación de procesos más usada en los centros de supercomputación. Básicamente consiste en una optimización del algoritmo de orden de llegada FCFS, en el cual, es posible que un job que llega más tarde a la cola de preparados inicie antes su ejecución que otro que está en la cabeza de dicha cola, si éste último está esperando por recursos (procesadores) y, la ejecución del que ha llegado más tarde no provocará el retraso del que espera por recursos.

Nota: Las siguientes figuras han sido obtenidas del documento de Tesis Doctoral de Francesc Guim Bernat, *Job-Guided Scheduling Strategies for Multi-Site HPC Infrastructures* (2008, pag. 15-16), con su consentimiento.

En la siguiente figura se muestra un diagrama que recrea la asignación de procesos en un sistema multiprocesador aplicando el algoritmo FCFS:

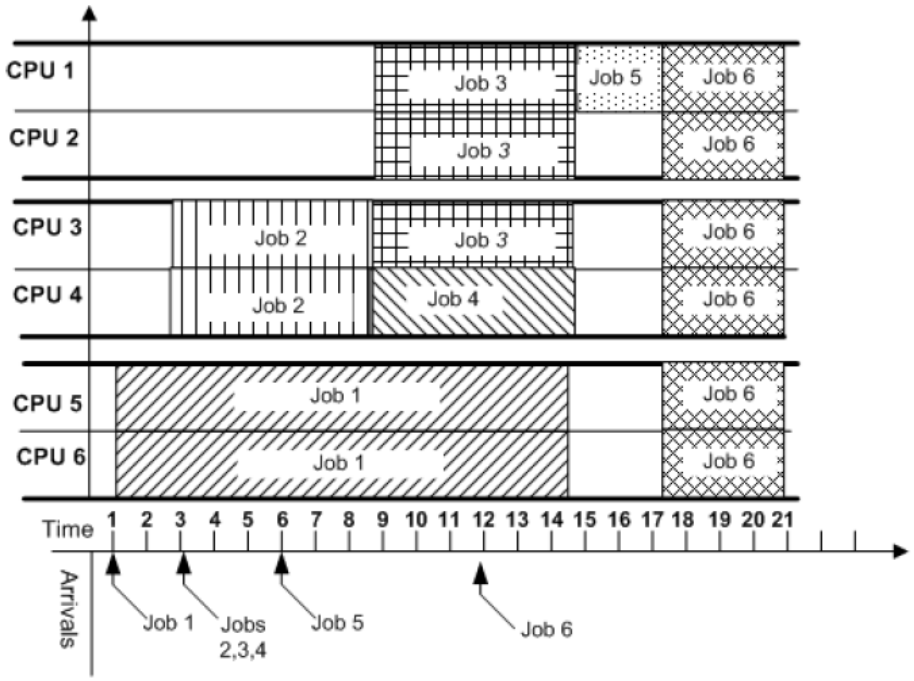


Fig 9. Ejecución de jobs mediante algoritmo FCFS.

En la anterior figura se puede observar como el job 3 no puede iniciar su ejecución porque el sistema sólo dispone de 2 procesadores libres. Por tanto, CPU 1 y CPU 2 quedan desaprovechadas, pues el job 4 y job 5, que sólo necesitan de 1 CPU, deben esperar su turno en un algoritmo FCFS, es decir, deben esperar a la ejecución del job 3 y hasta su finalización.

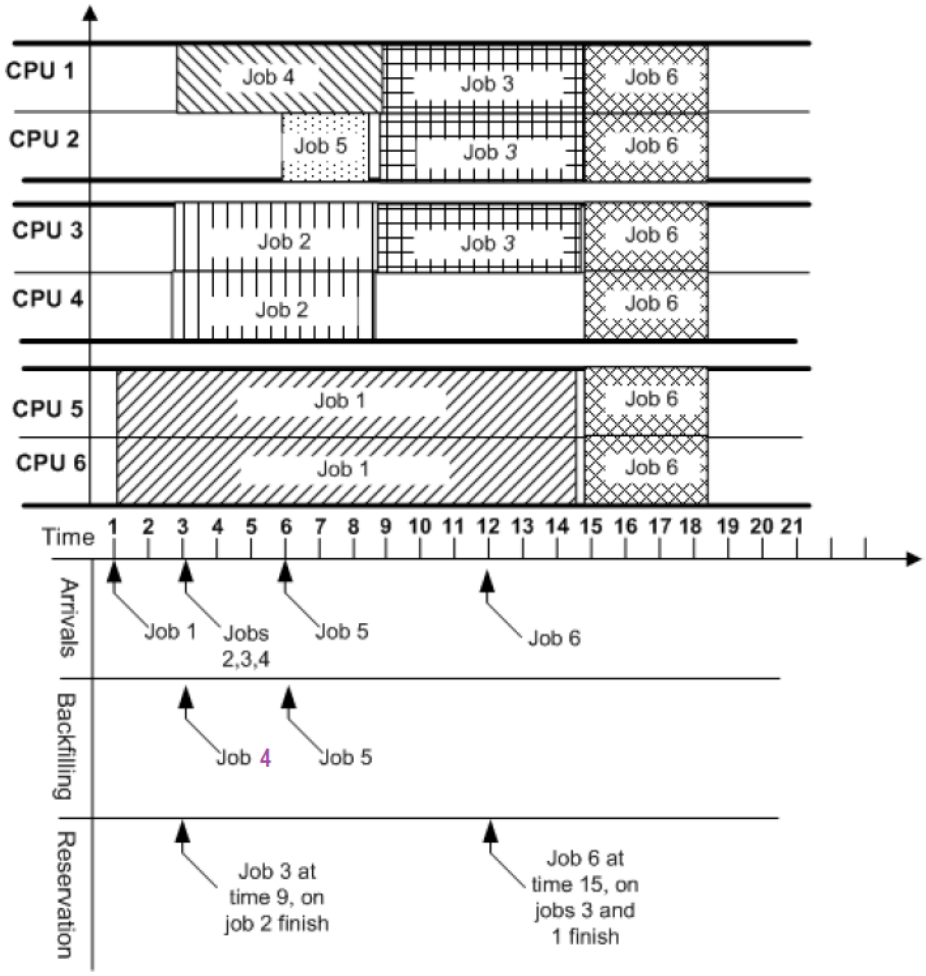


Fig 10. Ejecución de jobs mediante algoritmo FCFS aplicando EASY backfilling.

En la anterior figura se puede observar como el job 3 no puede iniciar su ejecución porque el sistema sólo dispone de 2 procesadores libres. En este caso, se produce backfilling del job 4 y job 5, aprovechando los procesadores que quedan disponibles sin retrasar por ello la ejecución del job 3 y, por tanto, tampoco la del job 6.

5 Construcción del proyecto

5.1 Análisis de requerimientos/definición de funcionalidades

Se requiere desarrollar una aplicación/tool que, dependiendo de los datos generados por el entorno de simulación de SST-Simulator, proporcione información sobre la idoneidad de la política de job scheduling establecida en dicho entorno. Esta información deberá arrojar resultados que permitan evaluar algunos de los parámetros que determinan el rendimiento de una determinada política de job scheduling. Se pretende poder evaluar las siguientes propiedades:

- **Uso del procesador**: Porcentaje de tiempo que el procesador ejecuta código de usuario.
- **Productividad (*throughput*)**: Número de procesos que finalizan por unidad de tiempo. Este parámetro tiende a variar dependiendo de la naturaleza de los procesos a ejecutar. Por ejemplo, procesos de larga duración (cálculos científicos) y corta duración (interactivos).
- **Tiempo de retorno (*turnaround time*)**: Tiempo transcurrido desde que un usuario inicia la ejecución de un proceso hasta que éste finaliza. Desde el punto de vista del usuario, es el tiempo que el programa ha tardado en ejecutarse.
- **Tiempo de espera**: Es el tiempo que un proceso ha pasado en el estado de '*preparado*' antes de cambiar a '*ejecución*'.
- **Tiempo de respuesta**: Es el tiempo que transcurre desde que se solicita una operación I/O hasta que se obtiene la respuesta.

Por tanto, la aplicación deberá:

- Aceptar unos datos de entrada, que serán los diferentes ficheros generados por el simulador SST-Simulator.
- Procesar los ficheros y obtener las diferentes métricas antes comentadas.
- Generar información, que podrá presentarse en formato gráfico o, como texto esquematizado sintetizando los resultados, que permita realizar posteriormente un análisis comparativo.

5.2 Diseño técnico

La aplicación a desarrollar, denominada Schedlyzer (Scheduler Analyzer), deberá estar estructurada funcionalmente de tal forma que cada función tenga unas responsabilidades bien definidas. Para ello, la aplicación se dividirá en tres funciones de tal forma que la información de salida de una función sea la de entrada de la siguiente.

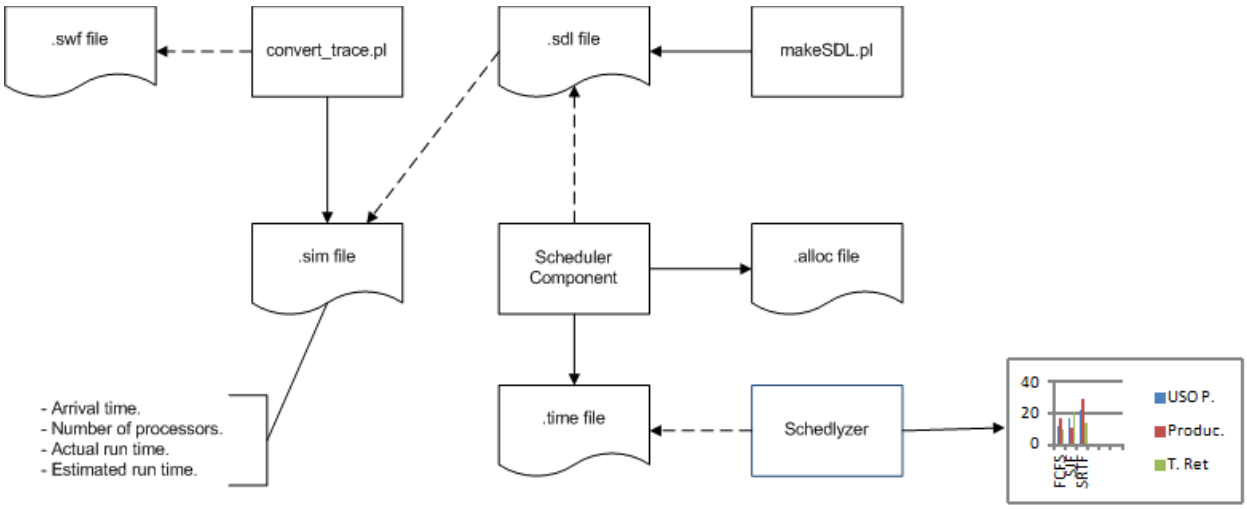


Fig 11. Esquema del sistema.

La siguiente figura muestra de forma simplificada la estructura funcional de la aplicación a desarrollar:

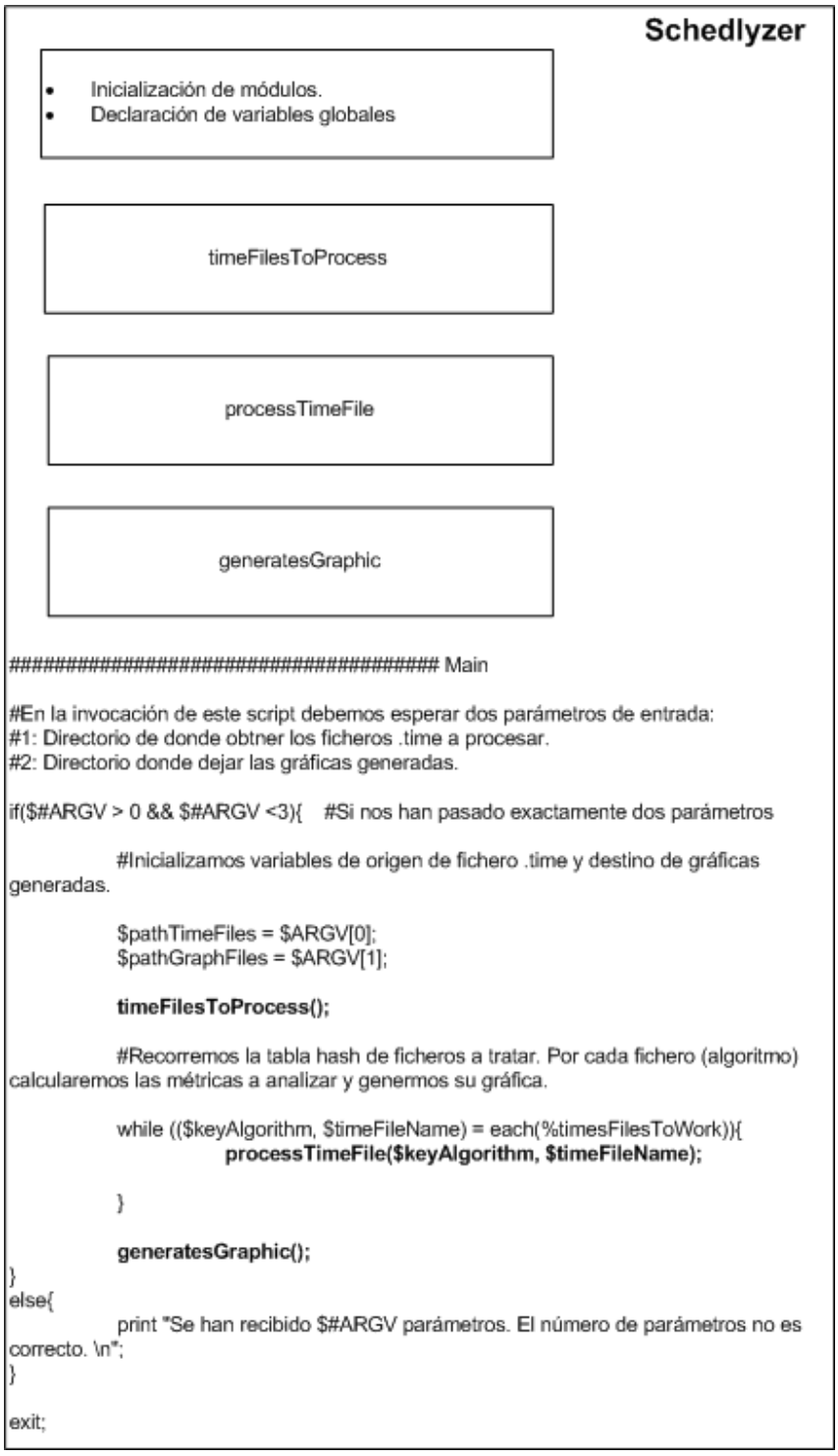


Fig 12. Estructura básica funcional de la aplicación Schedlyzer.

A continuación se presenta el diseño de cada función.

5.2.1 Función de adquisición de información

Nombre: timeFilesToProcess.

Parámetros:

- pPathFiles: Contendrá el path de los fichero .time a procesar.

Descripción: Encargada de obtener los diferentes ficheros que se encentren en la carpeta especificada en el parámetro de entrada 'pPathFiles', almacenándolos en la tabla hash 'timesFilesToWork'.

La función deberá obtener del path especificado todos los archivos con el siguiente formato:

<iniciales>_<resto de archivo.time>

Donde:

<iniciales>: Deberá ser la siglas de alguno de los algoritmos a analizar.

<resto de archivo.time>: Nombre que vendrá determinado por el fichero .sim utilizado en el SDL.

Los nombres de los archivos obtenidos deberán almacenarse en una tabla hash asociados al algoritmo de planificación con el que fueron generados:

```
$timesFilesToWork{$prefix} = $_;
```

Donde:

%timesFilesToWork: Tabla hash que asocia un fichero .time con la política de job scheduling con la que fue generado.

$prefix: Son las iniciales prefijadas obtenidas del nombre del fichero .time.

$_: Variable especial que contiene el último registro leído de un fichero. Útil en el procesado de un fichero mediante una estructura iterativa (bucle).

Algoritmo a desarrollar:

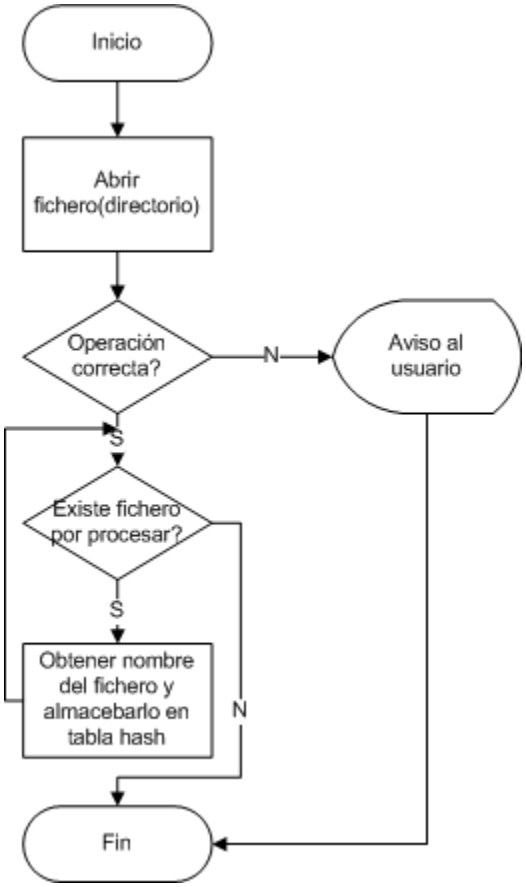


Fig 13. Diagrama de flujo de timeFilesToProcess.

5.2.2 Función de procesado de la información

Nombre: processTimeFile.

Parámetros:

- pPrefix: Contendrá las siglas del algoritmo considerado.
- pTimeFileName: Contendrá el nombre del fichero a procesar (obtención de métricas).

Descripción: Encargada de procesar el fichero de entrada informado en el parámetro pTimeFileName, calculando las siguientes métricas (ver: Análisis de requerimientos/definición de funcionalidades) para el algoritmo informado en el parámetro pPrefix; los cálculos obtenidos deberán guardarse en tablas hash para cada una de las métricas consideradas:

- **Uso del procesador.**

  Se calculará de la siguiente forma:

  ($totalRun / ($totalRun + $totalNOuserJob))*100

  Donde:

  $totalRun: La suma total de todos los tiempos de ejecución de los procesos (jobs) tratados en el sistema.

  $totalNOuserJob: La suma de los tiempos en los cuales no se estaba ejecutado ningún proceso de usuario; el sistema estaba ejecutando procesos de sistema. Para obtener este valor se debe ordenar las entradas del fichero .time por orden de inicio de ejecución y, posteriormente, para un determinado trabajo en inicio de ejecución, se deberá obtener el valor máximo de tiempo de finalización de sus predecesores. Si el tiempo de inicio del nuevo job en el sistema es mayor que el máximo obtenido significaría que ha existido un intervalo de tiempo en el cual el sistema no estaba ejecutando ningún job de usuario (ver: Uso del procesador (CPU utilization)).

- **Throughput (Productividad).**

  Se calculará de la siguiente forma:

  $totalJobs / ($theMaxEndTime - $startTime)

  Donde:

  $totalJobs: Número total de trabajos tratados por el sistema.

  $startTime: Marca el tiempo inicial en el cual comienza la simulación.

  $theMaxEndTime: Marca el tiempo final en el cual finaliza la simulación. Deberá obtenerse como el máximo tiempo de finalización alcanzado por un job ejecutado en el sistema.

- **Turnaround time (Tiempo de retorno).**

  Se calculará de la siguiente forma:

  $totalTurnaround / $totalJobs

  Donde:

  $totalTurnaround: Suma total de todos los tiempos de respuesta de los procesos (jobs) tratados por el sistema. El tiempo de respuesta de un proceso se calcula sumando su tiempo de ejecución + el tiempo consumido en estado de espera.

  $totalJobs: Número total de trabajos tratados por el sistema.

- **Waiting time (Tiempo de espera).**

  Se calculará de la siguiente forma:

  $totalWaitingTime / $totalJobs

  Donde:

  $totalWaitingTime: Suma total de todos los tiempos de espera de los procesos tratados por el sistema.

  $totalJobs: Número total de trabajos tratados por el sistema.

Estas tablas hash deberán disponerse de forma global para que puedan ser tratadas por la función de presentación de la información.

Los nombres de ficheros proporcionados a esta función son los obtenidos por la función de adquisición de información, es decir, son ficheros .time generados por SST mediante un SDL al procesar un archivo de worload (.sim).

A continuación se expone una muestra del contenido de un fichero .time:

```
# Simulation for trace SDSC-SP2-1998-4.2-cln.001.sim started Mon Jun  1 16:23:54
2015
# [Machine]
# SimpleMachine with 128 nodes, 1 cores per node
# [Scheduler]
# Priority Queue Scheduler
# Comparator: ShortestFirstComparator
# [Allocator]
# Simple Allocator
# [TaskMapper]
# Simple Task Mapper
```

# Job	Arrival	Start	End	Run	Wait	Resp.	Procs
2	567314	567314	575385	8071	0	8071	8
12	584165	584165	587799	3634	0	3634	24
15	584826	587799	587839	40	2973	3013	8
18	584875	587799	587846	47	2924	2971	16

19	584919	587846	588059	213	2927	3140	16
20	584928	588059	588099	40	3131	3171	16
16	584832	588099	588410	311	3267	3578	8
17	584838	588059	589287	1228	3221	4449	8
13	584785	588099	589328	1229	3314	4543	2
26	589449	589449	589724	275	0	275	1
10	582841	582841	591226	8385	0	8385	32

Fig 14. Ejemplo de contenido de fichero .time.

Cada línea del fichero .time, referente a la información de ejecución de un job, deberán almacenarse en una array bidimensional en el que, cada fila se corresponderá con una línea del fichero .time y, las columnas serán cada uno de los campos de los que se compone la línea.

Para obtener las métricas anteriormente descritas se deberá trabajar con las siguientes columnas:

```
$start = 2;
$end = 3;
$run = 4;
$wait = 5;
$resp = 6;
```

Algoritmo a desarrollar:

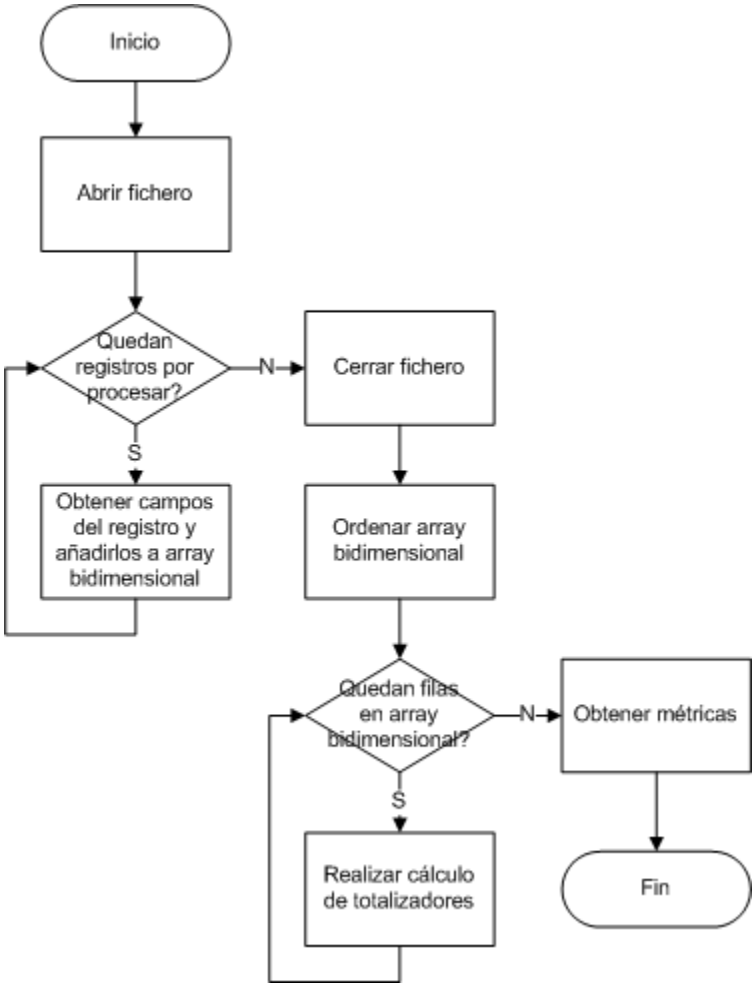


Fig 15. Diagrama de flujo de processTimeFile.

5.2.3 Función de presentación de la información

Nombre: generatesGraphic

Parámetros: N/A

Descripción: Encargada de generar la salida gráfica para representar, mediante diagrama de barras, la información obtenida en la función de procesado de la información.

Esta función deberá tratar la información almacenada en la tablas hash (accesibles globalmente) de cada métrica calculadas en la función de procesado de la información. Deberá por tanto generar una tabla por métrica calculada contemplando todos los algoritmos de planificación/comparadores tratados.

Algoritmo a desarrollar:

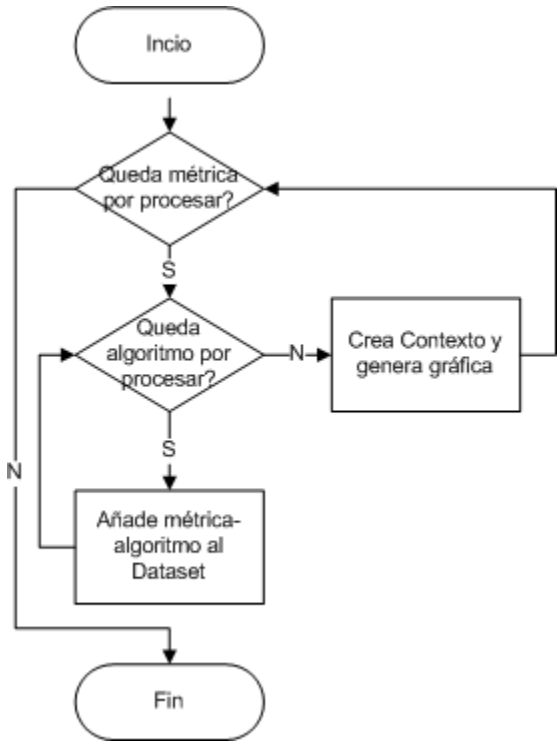


Fig 16. Diagrama de flujo de generatesGraphic.

5.2.4 Función principal (Main)

Nombre: N/A

Parámetros:

- @ARGV: Contendrá los argumentos de entrada que se deben especificar al realizar la invocación de la aplicación.

Descripción: No será propiamente una función. Este bloque de código será el encargado de aceptar los parámetros de entrada especificados en la invocación de la aplicación y realizar las llamadas a las funciones anteriormente descritas en el orden adecuado para que, a partir de unos ficheros de resultados de simulación de SST-Simulator (ficheros .time), se generen gráficas de las métricas obtenidas para su posterior análisis.

El bloque Main deberá comprobar que el número de parámetros de entrada es 2, correspondiente al path de la carpeta que contiene los ficheros .time de origen y al path de la carpeta de destino que contendrá las gráficas generadas por cada métrica obtenida. Si el número de parámetros no es correcto se deberá avisar al usuario finalizando la ejecución de la aplicación.

Algoritmo a desarrollar:

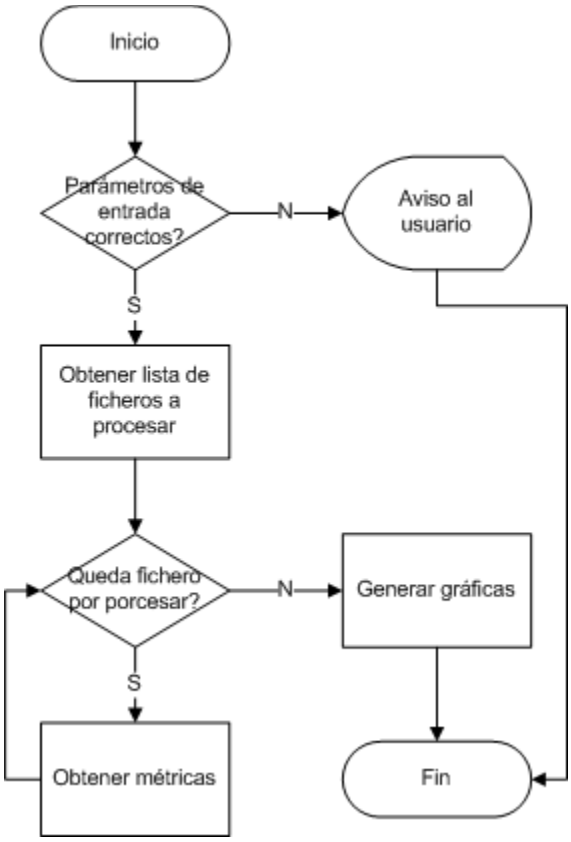


Fig 17. Diagrama de flujo del bloque principal (Main).

5.3 Ejecución de pruebas

5.3.1 Entornos

Para la ejecución de pruebas de simulación se dispondrá del servidor de la UOC *eimtarqso.uoc.edu* con una cuenta de usuario que permita la creación de nuevas carpetas en donde crear las diferentes pruebas que deberán ser sometidas al simulador SST-Simulator. En dicho Servidor se dispone de todos los requerimientos de sistema necesarios para la ejecución del simulador (ver: Requerimientos de sistema).

Para la realización de la aplicación, así como la obtención de los datos/gráficas a analizar, se dispondrá una máquina local virtualizada con distribución de Linux Ubuntu. Dicha distribución deberá incorporar un intérprete de Perl y el módulo Chart::Clicker. Por tanto, los ficheros .time generados en el servidor *eimtarqso.uoc.edu* deberán ser copiados a la máquina virtual para poder ser procesados por la aplicación.

5.3.2 Ejecución

Una vez se disponga de los ficheros .time en una carpeta del sistema se deberá invocar a la aplicación desarrollada de la siguiente forma:

schedlyzer_in <*path ficheros .time*> <*path gráficas a generar*>

Donde:

<*path ficheros .time*>: Se deberá especificar el path en donde se encuentran los fichero .time a procesar. Ejemplo: /home/jadm/jdominguezmed/work/schedlyzer/timefiles

<*path gráficas a generar*>: Se deberá especificar el path en donde queremos generar las gráficas de las métricas analizadas para cada uno de los algoritmos. Ejemplo: /home/jadm/jdominguezmed/work/schedlyzer/graphics

5.3.3 Resultados

Como resultado se generará una serie de gráficas, una por cada métrica obtenida, en la carpeta indicada en el segundo parámetro especificado en la invocación de la aplicación (ver: Ejecución). El formato de dichas gráficas será en PNG [PNG 1] y consistirá en un diagrama de barras en el que será posible realizar un análisis comparativo de los resultados por cada una de las políticas de job scheduling consideradas.

Para la realización de este TFG se obtendrán dos juegos de gráficas, uno generado mediante el procesamiento de ficheros .time obtenidos de le ejecución del algoritmo Prioritario utilizando varios comparadores y, el otro juego de gráficas, se obtendrá mediante el procesamiento de ficheros .time obtenidos de le ejecución del algoritmo EASY (implementando la técnica de backfilling) utilizando varios comparadores.

6 Análisis de resultados

En los siguientes puntos se analiza el resultado de la ejecución de SST-Simulator implementando varias políticas de job scheduling. El análisis se ha estructurado teniendo en cuenta las diferentes métricas obtenidas mediante la aplicación schedlyzer, realizando un análisis comparativo de los resultados obtenidos y como estos contribuyen en la mejora de las características que determinan el rendimiento de un scheduler (ver: El Scheduler).

Se realiza un análisis de las métricas obtenidas con el algoritmo prioritario *PQScheduler*, implementando diferentes comparadores para posteriormente ser contrastado con las métricas obtenidas mediante algoritmo *EASYScheduler*, que implementa la técnica de backfilling (ver: Backfilling policies). Para facilitar la referencia a los resultados de un determinado algoritmo, las gráficas presentadas disponen de la siguiente codificación de los nombres de dichos algoritmos:

Schedulers utilizados:

- PQS = PQScheduler
- EASY = EASYScheduler

Comparadores utilizados:

- **FIFO**: Implementa algoritmo FCFS.
- **LARGEFIRST**: Da prioridad a los jobs con mayor requerimiento de procesadores. Si se dispone de jobs con mismos números de procesadores se utiliza el algoritmo FIFO.
- **SMALLFIRST**: Da prioridad a los jobs con menor requerimiento de procesadores. Si se dispone de jobs con mismos números de procesadores se utiliza el algoritmo FIFO.
- **LONGFIRST**: Da prioridad a los jobs con mayor tiempo de runningtime (tiempo de ejecución) estimado. Si se dispone de jobs con mismos tiempos de ejecución estimado se utiliza el algoritmo FIFO.
- **SHORTFIRST**: Da prioridad a los jobs con menor tiempo de runningtime (tiempo de ejecución) estimado. Si se dispone de jobs con mismos tiempos de ejecución estimado se utiliza el algoritmo FIFO. Es una implementación del SJF.

Para todos ellos, en última instancia, se utilizaría el *job num* interno para determinar su prioridad.

Debido a las características de los sistemas HPC, ninguna de las implementaciones de las anteriores políticas de job scheduling es apropiativa, es decir, los jobs no son interrumpidos y ejecutan hasta finalizar.

Ejemplo de codificación de nombre de política de planificación al que hace referencia un dato:

PQS-SHORTFIRST: Algoritmo prioritario implementando el comparador shortfirst, es decir, algoritmo SJF.

Aclaraciones:

- **Para todas las métricas analizadas, sólo se comentan los resultados más relevantes.**
- **Para las implementaciones PQScheduler y EASYScheduler el comparador SMALLFIRST da el mismo resultado, por lo que consideramos que debe haber algún bug en SST y por tanto este comparador para EASYScheduler no será comentado.**

Antes de iniciar el estudio de las métricas calculadas por la aplicación Schedlyzer, pasamos a presentar algunos datos de ejemplo relevantes obtenidos de un fichero .time que vienen a completar los comentados en la descripción del fichero de workload The San Diego Supercomputer Center (SDSC) SP2 log, en cuanto a magnitudes se refiere.

- **Máximo Run Time obtenido**: 510209s -> 141,72h
- **Máximo Waiting Time obtenido**: 5509181s -> 1530,33h
- **Máximo Resp. Time obtenido**: 5541723s -> 1539,37h
- **Máximo Num. Procs requeridos**: 128

6.1 Uso del procesador (CPU utilization)

6.1.1 Consideraciones sobre la métrica

El *uso del procesador* es una métrica a maximizar para optimizar características del scheduler como son el aprovechamiento y la eficiencia. Al disponer de un sistema multiprocesador simulado de 128 nodos, consideramos el *uso del procesador* al porcentaje de tiempo sobre el total de cómputo en el cuál al menos una CPU ejecuta un proceso de usuario. Sólo cuando ninguna de las 128 CPU's están asignadas a jobs de *usuario* se realiza el cómputo de tiempo consumido por el *sistema*.

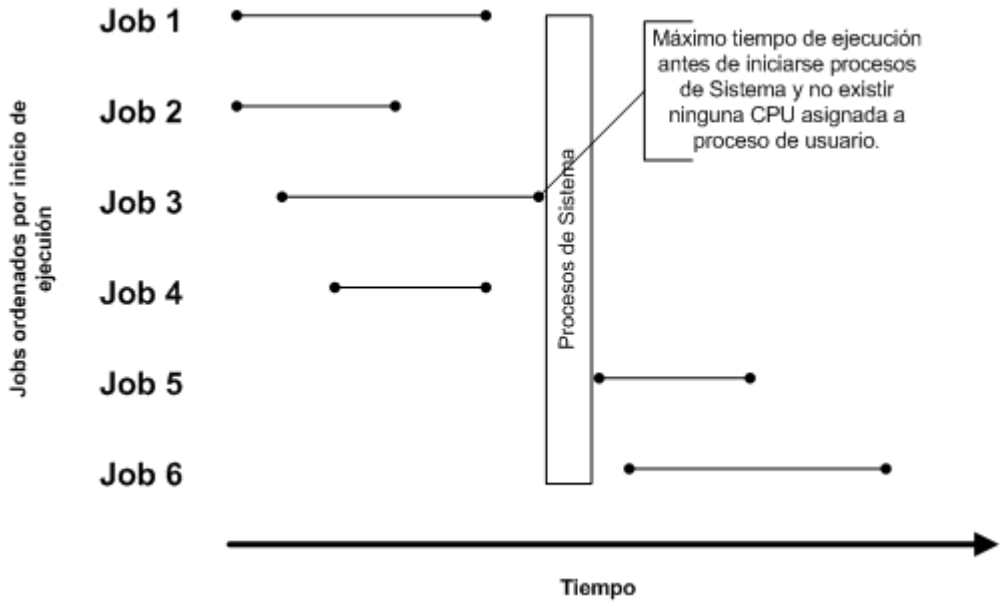


Fig 18. Intervalo de tiempo de cómputo de procesos de Sistema.

En la anterior figura podemos ver como diferentes procesos (job 1 a job 4) ejecutan en paralelo, cada uno de ellos con sus requerimientos de CPU particulares y ordenados por inicio de ejecución. Cada vez que entra un nuevo proceso en el sistema se mira si el tiempo de inicio de su ejecución es mayor que al máximo tiempo de ejecución de sus predecesores. Si es así (caso del job 5), se dispone de un intervalo de tiempo en el cual el OS ha estado ejecutando procesos del sistema y no de usuario.

6.1.2 Análisis PQScheduler

La siguiente gráfica muestra los resultados obtenidos mediante el algoritmo Prioritario implementando varios comparadores.

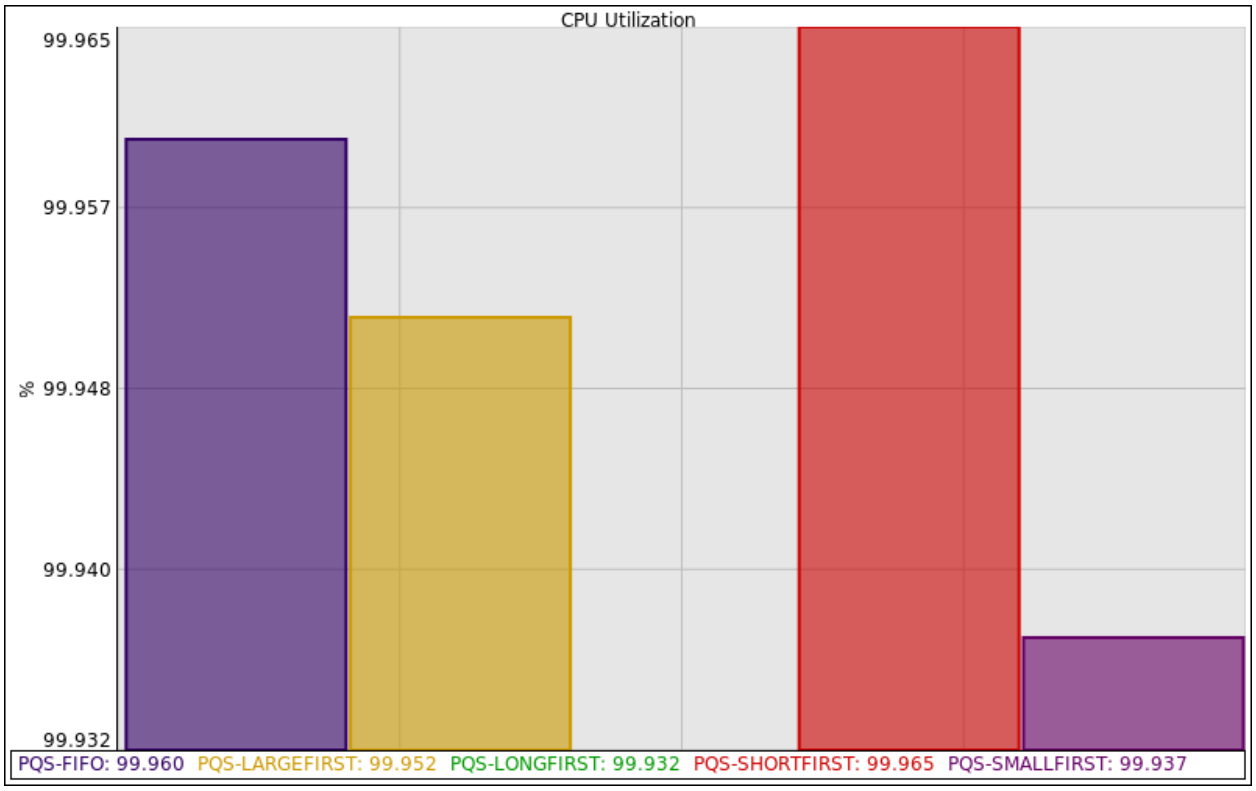


Fig 19. Resultado de la métrica *CPU utilization mediante* PQScheduler.

Inicialmente podemos comprobar que para todos los comparadores implementados, el uso del sistema por parte de procesos de usuario son valores muy cercanos al 100%, maximizando su eficiencia.

PQS-SHORTFIRST se presenta como mejor valor, obteniendo un resultado de 99,965% y PQS-LONGFIRST como peor valor, obteniendo un resultado de 99,932% (aunque tal y como he comentado anteriormente, ambos valores son muy altos). PQS-SHORTFIRST, al darle prioridad a los jobs con menor tiempo de *run time*, consigue tener un mayor tiempo de CPU ejecutando trabajos de usuario, asignando con mayor agilidad las CPU's de las que dispone el sistema a los diferentes jobs que se van presentando en la cola de preparados y requiriendo de menor tiempo de sistema para realizar estas tareas. Por lo general y, analizando los datos del workload, se

podría considerar tendencia de que los jobs con menor tiempo de *run time* son los que más recursos de CPU utilizan (ver fig 20). Por tanto, si en el sistema se presentan jobs con similares demandas de recursos de CPU, es posible obtener este buen rendimiento.

Al contrario, PQS-LONGFIRST, al darle prioridad a los jobs con mayor tiempo de *run time*, debe realizar una menor frecuencia de cambios y, tal como muestra la tendencia de la siguiente figura, con menor demanda de recursos de CPU. Esto puede provocar una mayor fragmentación de los recursos disponibles penalizando en los tiempos que son empleados por el planificador para gestionarlos, así como en los tiempos de espera (ver: Tiempo de espera (Waiting Time)) derivados de una posible mayor frecuencia/probabilidad de indisponibilidad de procesadores libres en el sistema.

	A	B	C	D	E	F	G	H
1	Job	Arrival	Start	End	Run	Wait	Resp.	Procs
68	85	620527	631100	667069	35969	10573	46542	11
69	86	621578	635996	671964	35968	14418	50386	11
70	87	621582	635996	671965	35969	14414	50383	11
71	88	638224	638224	646251	8027	0	8027	8
72	89	639157	639157	658012	18855	0	18855	4
73	90	639316	646251	646309	58	6935	6993	13
74	91	639459	646309	646369	60	6850	6910	16
75	92	639523	646369	646431	62	6846	6908	19
76	93	639759	646431	646492	61	6672	6733	3
77	94	639994	646431	646472	41	6437	6478	8
78	95	639995	646431	646476	45	6436	6481	8
79	99	645540	646472	682093	35621	932	36553	1
80	100	645808	646472	695275	48803	664	49467	1
81	101	645978	646472	669527	23055	494	23549	1
82	103	646393	646472	700620	54148	79	54227	1
83	116	652231	652231	667000	14769	0	14769	11
84	119	653562	653562	667952	14390	0	14390	1
85	124	662796	662796	727623	64827	0	64827	4
86	122	658981	671964	672251	287	12983	13270	32

a) PQS-LONGFIRST

	A	B	C	D	E	F	G	H
1	Job	Arrival	Start	End	Run	Wait	Resp.	Procs
68	68	617477	635996	636339	343	18519	18862	32
69	71	617729	635996	636178	182	18267	18449	16
70	72	617737	636339	636555	216	18602	18818	32
71	75	617914	636339	636504	165	18425	18590	16
72	76	617917	636555	636735	180	18638	18818	32
73	79	618202	636555	636908	353	18353	18706	16
74	80	618628	636735	636909	174	18107	18281	16
75	81	618949	636735	636896	161	17786	17947	16
76	84	619804	636896	636920	24	17092	17116	1
77	69	617482	637636	638023	387	20154	20541	64
78	70	617486	638023	638546	523	20537	21060	100
79	73	617748	638546	638809	263	20798	21061	64
80	74	617752	638809	639143	334	21057	21391	100
81	77	617923	639143	639342	199	21220	21419	64
82	78	617928	639342	639643	301	21414	21715	100
83	64	613917	639643	639738	95	25726	25821	64
84	85	620527	639643	675612	35969	19116	55085	11
85	86	621578	639643	675611	35968	18065	54033	11
86	87	621582	639643	675612	35969	18061	54030	11

b) PQS-SHORTFIRST

Fig 20. Muestra de datos analizados (se observa la relación: mayor tiempo de ejecución -> menor demanda de CPU's -> menor tiempo de espera).

PQS-LARGEFIRST vendría a corroborar lo anteriormente comentado, pues muestra como dar prioridad a jobs con mayor demanda de CPU's, es decir, los que hemos considerado que disponen de un menor tiempo de *run time*, obtienen un mayor uso de CPU que PQS-SMALLFIRST, que es el caso contrario.

Para esta métrica, el algoritmo más simple, el PQS-FIFO, obtiene el tercer mejor resultado para el workload considerado. Podría considerarse que existe un equilibrio entre los trabajos que se van sometiendo en el sistema de tal forma que los recursos son distribuidos/asignados sin introducir excesiva penalización por el efecto convoy y evitando el starvation [STARV 1].

6.1.3 Análisis EASYScheduler

La siguiente gráfica muestra los resultados obtenidos mediante el algoritmo EASY aplicando la técnica de backfilling e implementando varios comparadores.

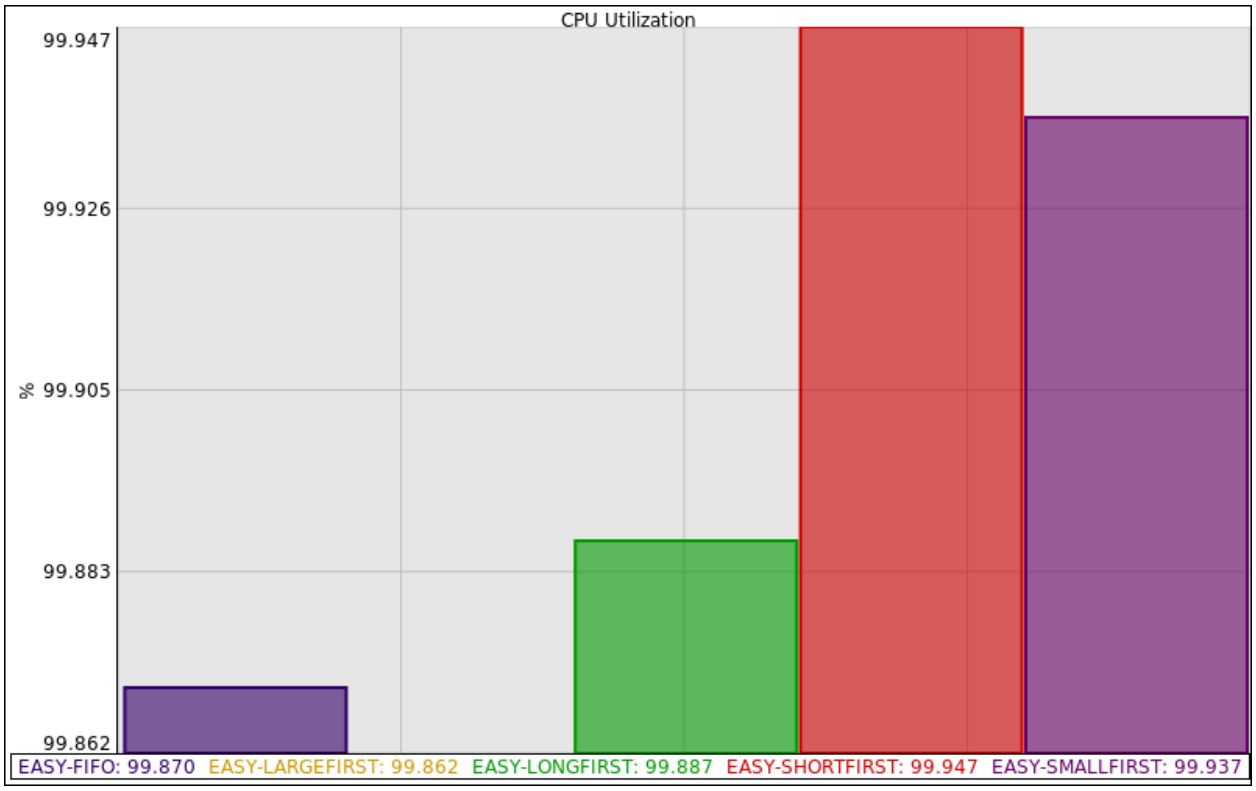


Fig 21. Resultado de la métrica *CPU utilization mediante* EASYScheduler.

Mediante la anterior gráfica podemos comprobar que, aplicando la técnica de backfilling, todas las políticas de job scheduling han empeorado sutilmente sus resultados en relación a los obtenidos por la implementación PQScheduler (EASY-SMALLFIRST ha obtenido el mismo). Aunque hay algoritmos que han experimentado un cambio en la relación que tenían con respecto a los demás. Por ejemplo, para los datos de workload que estamos analizando, vemos que EASY-LONGFIRST obtiene mejor resultado que EASY-FIFO.

Podríamos decir que, al introducir la técnica de bacfilling, al ser más complejo el algoritmo de planificación, se ha conseguido disminuir ligeramente el porcentaje de uso que el sistema ha dedicado a procesos de usuario.

6.2 Productividad (throughput)

6.2.1 Consideraciones sobre la métrica

La Productividad es una métrica a maximizar para optimizar características del scheduler como son el repartimiento, la interactividad y el aprovechamiento. Para obtener dicha métrica se debe disponer del cálculo del tiempo total que el sistema ha dedicado en el procesamiento de todos los jobs incluidos en el workload. Se comprobará que cada algoritmo ha empleado un tiempo diferente para completar la totalidad de los trabajos presentados en el sistema, dando lugar a los diferentes resultados obtenidos.

6.2.2 Análisis PQScheduler

La siguiente gráfica muestra los resultados obtenidos mediante el algoritmo Prioritario implementando varios comparadores.

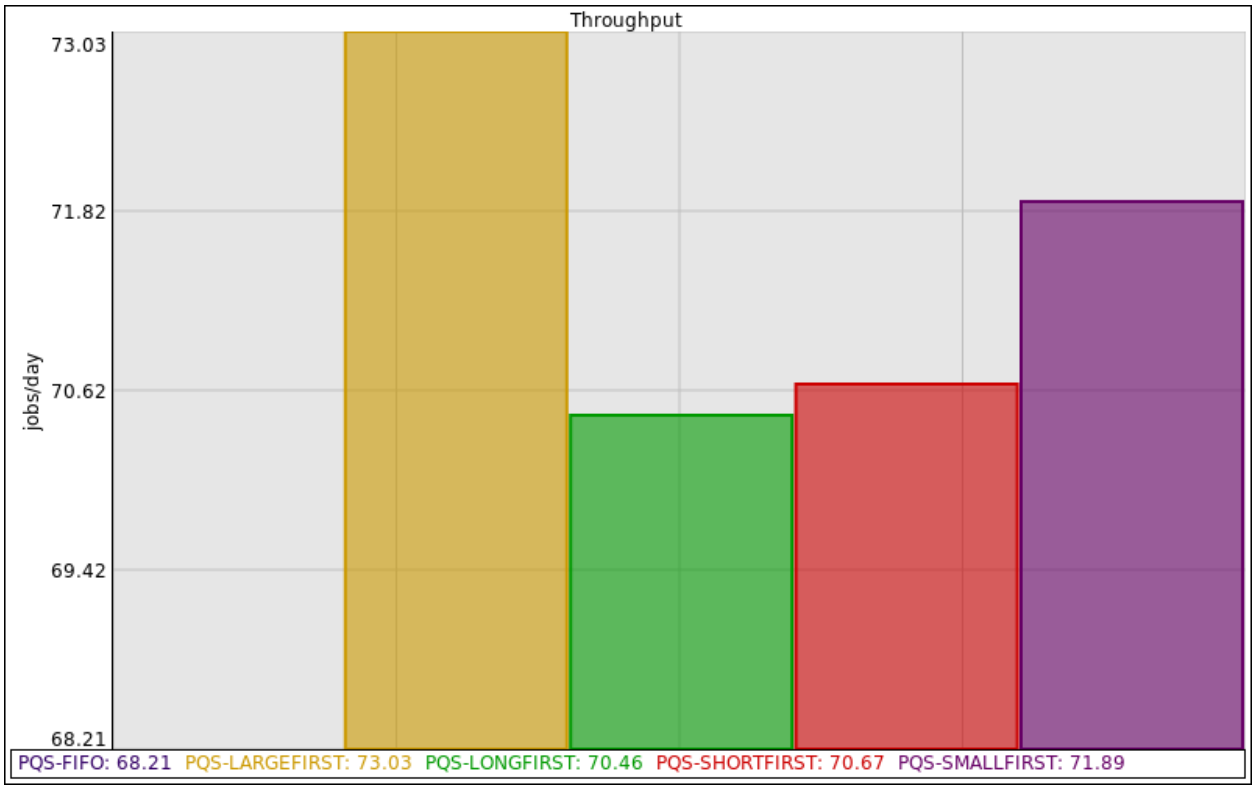


Fig 22. Resultado de la métrica *Throughput* mediante PQScheduler.

Mediante la anterior gráfica podemos observar que los valores obtenidos para todos los algoritmos son muy parecidos, diferenciados por unas pocas unidades.

PQS-LARGEFIRST se presenta como mejor valor, obteniendo un resultado de 73,03 jobs/day y PQS-FIFO como peor valor, obteniendo un resultado de 68,21 jobs/day. Estos resultados no estarían muy acordes con las estadísticas comentadas sobre este workload (ver: The San Diego Supercomputer Center (SDSC) SP2 log), relacionada con la llegada al sistema de unos 20 por día pero, se debe tener en cuenta que el procesador Thin Node POWER2 que incorporaba cada uno de los 128 del IBM SP2 disponía de una frecuencia de reloj de 66.7 MHz (se debe tener en cuenta que en este estudio no se ha tenido en cuenta la frecuencia de reloj utilizada por el simulador SST-Simulator).

PQS-LARGEFIRST, al darle prioridad a los jobs con mayor requerimiento de CPU, consigue finalizar un mayor número de trabajos por unidad de tiempo dando rápidamente salida a los procesos teóricamente de un tamaño mediano/pequeño maximizando la disponibilidad. También se da el caso, como se comenta para la métrica *uso del procesador*, que los procesos con un mayor requerimiento de CPU's provocan una menor fragmentación de recursos. Por tanto, un mayor intercambio de procesos en el sistema no introduce excesiva penalización por falta de recursos (CPU's).

Para PQS-SMALLFIRST, que obtienen la segunda mejor puntuación con 71,89 jobs/day, vemos que al seleccionar jobs por recursos de CPU requeridos –en este caso seleccionando prioritariamente los de menor requerimiento- consigue introducir poca fragmentación en el sistema por lo que, cuando un job finaliza su ejecución, liberando n CPU's, se incrementa la probabilidad de que el siguiente a ejecutar también requiera de las mismas n CPU's, no introducción *wait times* penalizando la productividad.

PQS-LONGFIRST y PQS-SHORTFIRST no presentan mucha diferencia por lo que podemos inferir que los algoritmos prioritarios que seleccionan estrictamente por requerimiento de CPU's son los que obtienen una mayor relevancia en el sistema simulado y en cuanto a productividad se refiere.

Para esta métrica, el algoritmo más simple, el PQS-FIFO obtiene el peor resultado para el workload considerado. Podría considerarse que este algoritmo es el que más penalizaciones introduce en el sistema por no evitar situaciones no deseable como pueden ser el efecto convoy y/o una excesiva fragmentación de recursos/disponibilidad de CPU's.

6.2.3 Análisis EASYScheduler

La siguiente gráfica muestra los resultados obtenidos mediante el algoritmo EASY aplicando la técnica de backfilling e implementando varios comparadores.

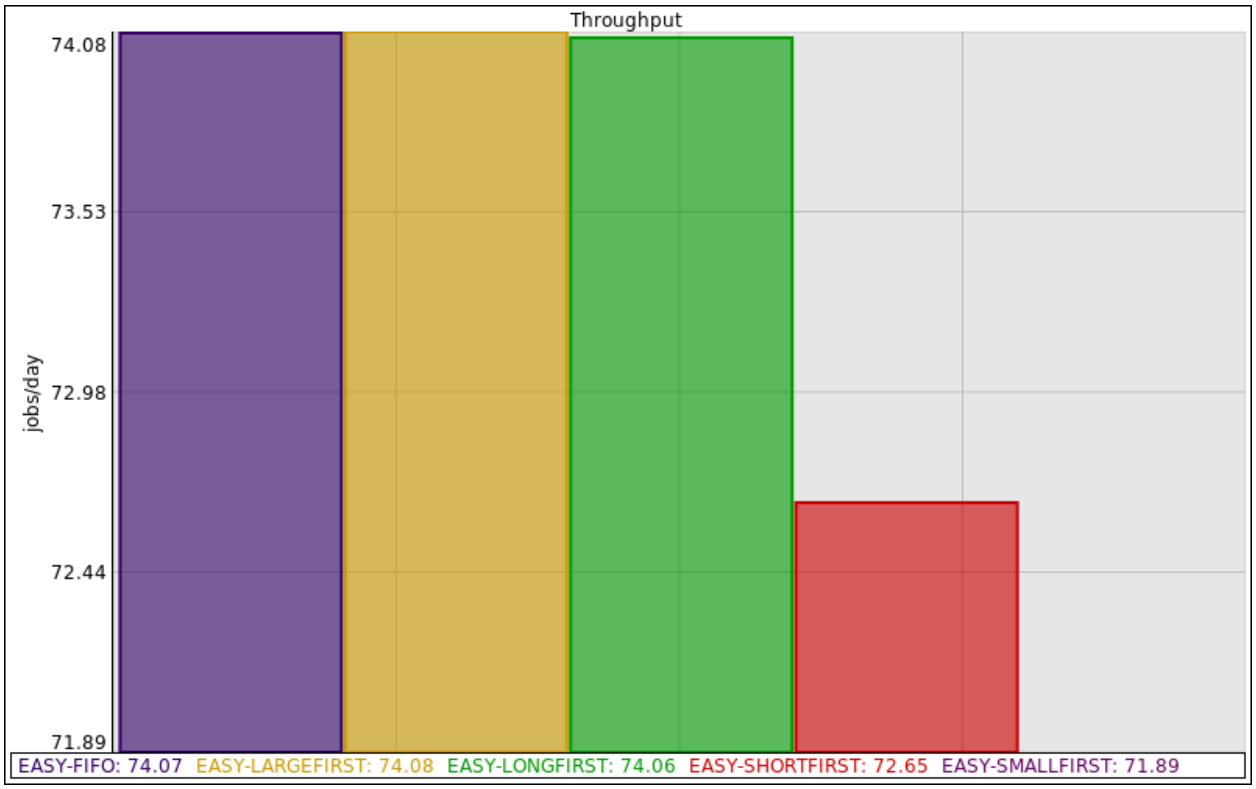


Fig 23. Resultado de la métrica *Throughput* mediante EASYScheduler.

Mediante la anterior gráfica podemos comprobar que, aplicando la técnica de backfilling, todas las políticas de job scheduling han mejorado sus resultados en relación a los obtenidos por la implementación PQScheduler (EASY-SMALLFIRST ha obtenido el mismo). Además, hay algoritmos que han experimentado un cambio en la relación que tenían con respecto a los otros. Por ejemplo, para los datos de workload que estamos analizando, vemos que EASY-FIFO obtiene mejor resultado que EASY-SHORTFIRST.

Podríamos decir que, al introducir la técnica de bacfilling, aun siendo más complejo el algoritmo de planificación, se ha conseguido aumentar el rendimiento del sistema para cualquiera de los algoritmos implementados, eliminando los tiempos de espera y acortando por tanto el tiempo total necesario para finalizar todos los trabajos.

6.3 Tiempo de retorno (Turnaround Time)

6.3.1 Consideraciones sobre la métrica

El *tiempo de retorno* es una métrica a minimizar para optimizar características del scheduler como son el repartimiento y la interactividad. La gráfica presentada muestra la media calculada de los tiempos de retorno obtenidos por cada algoritmo de planificación.

6.3.2 Análisis PQScheduler

La siguiente gráfica muestra los resultados obtenidos mediante el algoritmo Prioritario implementando varios comparadores.

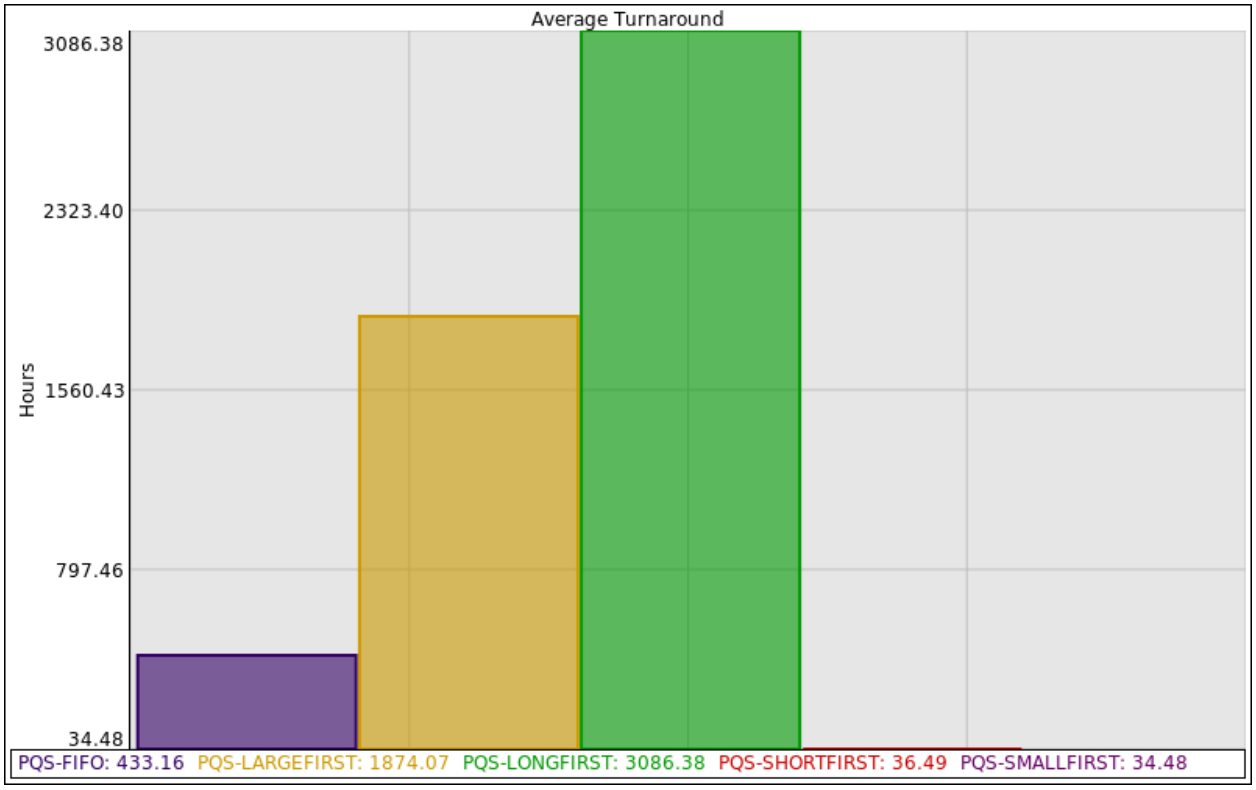


Fig 24. Resultado de la métrica *Turnaround* mediante PQScheduler.

Inicialmente podemos comprobar que los resultados obtenidos para cada una de los algoritmos analizados son muy dispares, yendo desde unas pocas horas a varios meses.

PQS-SMALLFIRST se presenta como mejor valor, obteniendo un resultado de 34,48h y PQS-LONGFIRST como peor valor, obteniendo un resultado de 3086,38h. Por tanto, podemos decir que PQS-SMALLFIRST desbanca a PQS-SHORTFIRST, que tradicionalmente es el algoritmo que obtiene mejores resultados para esta métrica (Turnaround) y, PQS-LONGFIRST mantiene su puesto como el algoritmo que más la penaliza. Tal y como se ha comentado para la métrica de productividad (Throughput), esto último podría deberse a la uniformidad de requerimientos de CPU's introducida por PQS-SMALLFIRST.

Se debe tener en cuenta que para el sistema multiprocesador implementado, el disponer de un tiempo de respuesta excesivamente alto para algún comparador no significa que su correspondiente métrica de productividad sea excesivamente baja pues, se debe tener el cuenta el grado de paralelismo del sistema que podría permitir disponer de procesos pesados ejecutando durante largos intervalos de tiempo (aumentando la media) junto con otros más ligeros no alargando excesivamente el tiempo total de procesado de todos los jobs.

Teniendo en cuenta que el tiempo de espera (waiting time) introducido por cada job es un factor muy importante para el cálculo de esta métrica, podemos corroborar, mediante la gráfica de la figura 26, que la relación es lineal, es decir, a mayor tiempo de media de *waiting time*, mayor es la media del tiempo de *turnaround*.

6.3.3 Análisis EASYScheduler

La siguiente gráfica muestra los resultados obtenidos mediante el algoritmo EASY aplicando la técnica de backfilling e implementando varios comparadores.

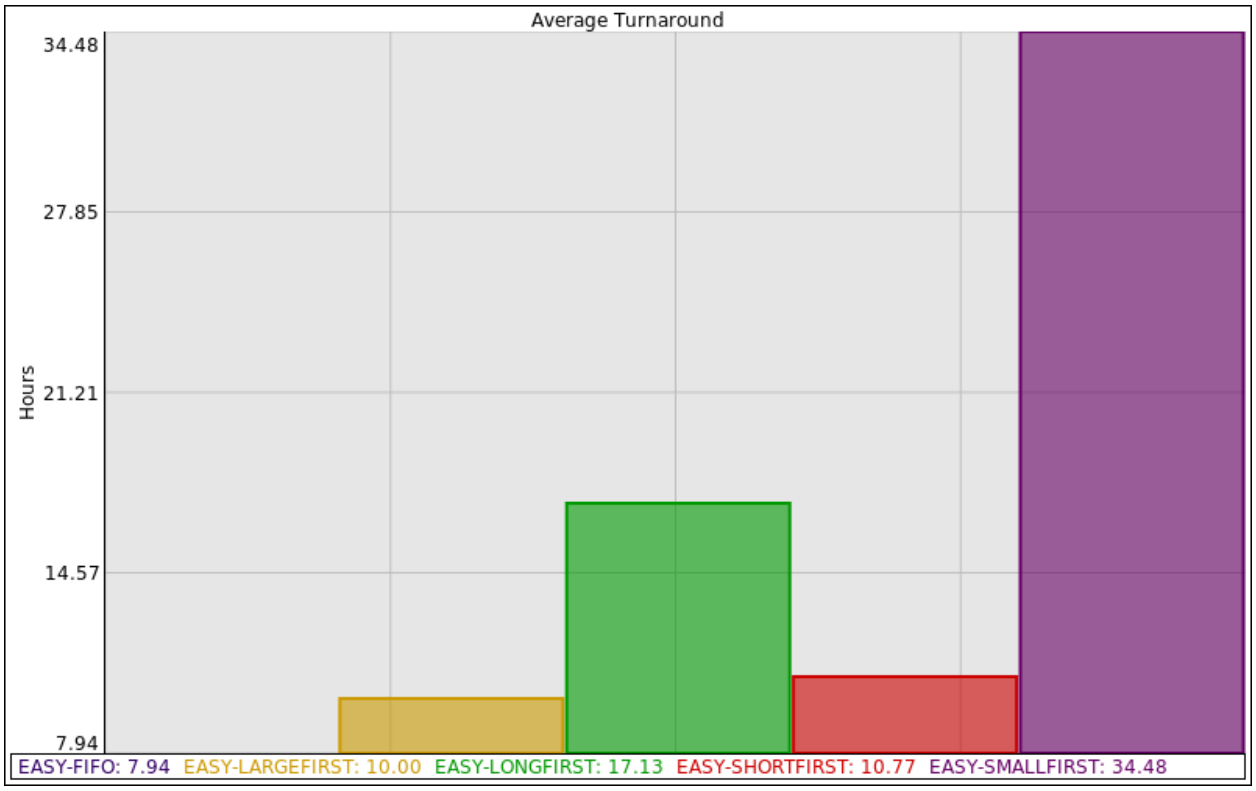


Fig 25. Resultado de la métrica *Turnaround* mediante ESYScheduler.

Mediante la anterior gráfica podemos comprobar que, aplicando la técnica de backfilling, todas las políticas de job scheduling han mejorado sus resultados, de forma espectacular, en relación a los obtenidos por la implementación PQScheduler, pudiendo pasar a hablar sólo de magnitud horas y no de meses. Además, hay algoritmos que han experimentado un cambio en la relación que tenían con respecto a los otros. Por ejemplo, para los datos de workload que estamos analizando, vemos que EASY-FIFO obtiene mejor resultado que EASY-SHORTFIRST.

Podríamos decir que, al introducir la técnica de backfilling, aun siendo más complejo el algoritmo de planificación, se ha conseguido aumentar el rendimiento del sistema para cualquiera de los algoritmos implementados, eliminando los tiempos de espera y acortando por tanto el tiempo total necesario para finalizar todos los trabajos.

6.4 Tiempo de espera (Waiting Time)

6.4.1 Consideraciones sobre la métrica

El *tiempo de espera* es una métrica a minimizar para optimizar características del scheduler como son la disponibilidad, el repartimiento y el aprovechamiento. La gráfica presentada muestra la media calculada de los tiempos de espera obtenidos por cada algoritmo de planificación.

6.4.2 Análisis PQScheduler

La siguiente gráfica muestra los resultados obtenidos mediante el algoritmo Prioritario implementando varios comparadores.

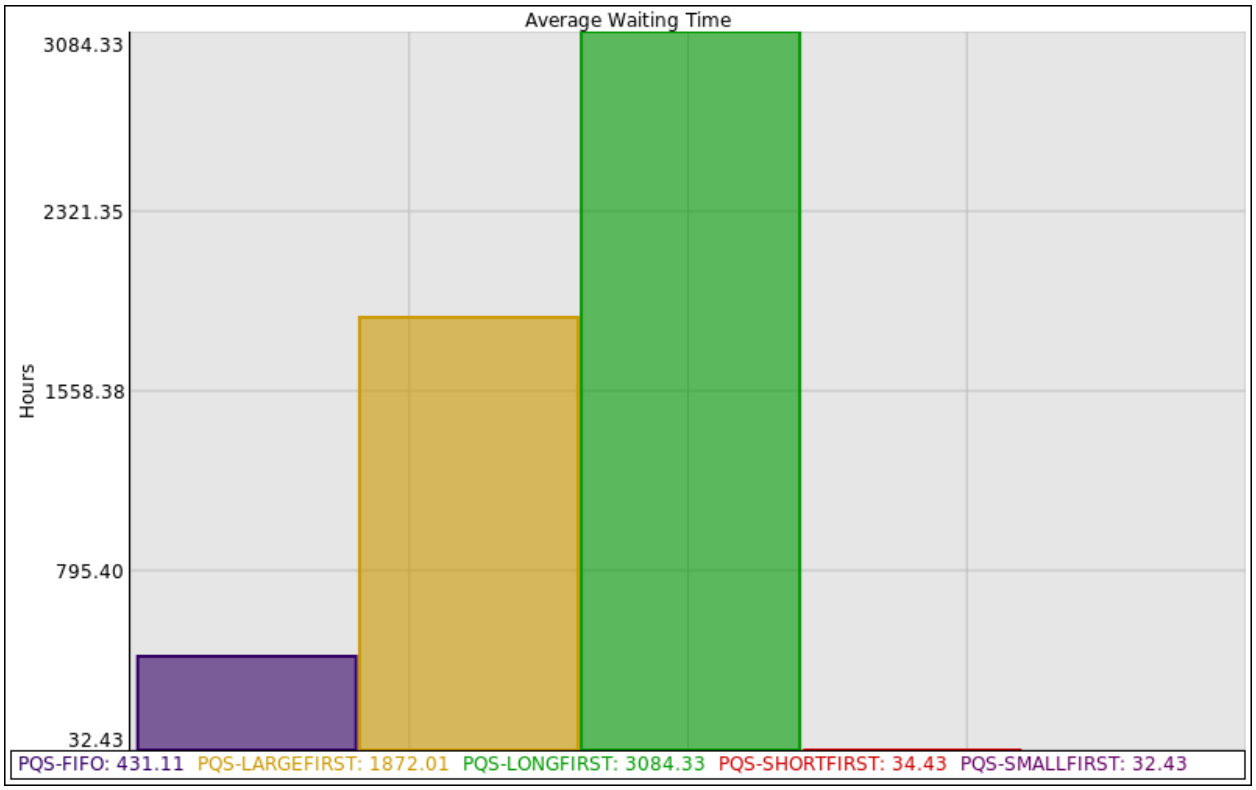


Fig 26. Resultado de la métrica *Waiting Time* mediante PQScheduler.

Al igual que con la anterior métrica (Turnaround), para la cual se ha comentado que tienen una relación lineal, podemos comprobar que los resultados obtenidos para cada una de los algoritmos analizados son muy dispares, yendo desde unas pocas horas a varios meses.

PQS-SMALLFIRST se presenta como mejor valor, obteniendo un resultado de 32,43h y PQS-LONGFIRST como peor valor, obteniendo un resultado de 3084,33h. Tal y como se ha comentado para la métrica de productividad (Throughput), este buen resultado de PQS-SMALLFIRST podría deberse a la uniformidad de requerimientos de CPU's introducida, es decir, el solicitar mismos requerimientos de CPU por unidad de tiempo, mitigando la fragmentación de recursos.

6.4.3 Análisis EASYScheduler

La siguiente gráfica muestra los resultados obtenidos mediante el algoritmo EASY aplicando la técnica de backfilling e implementando varios comparadores.

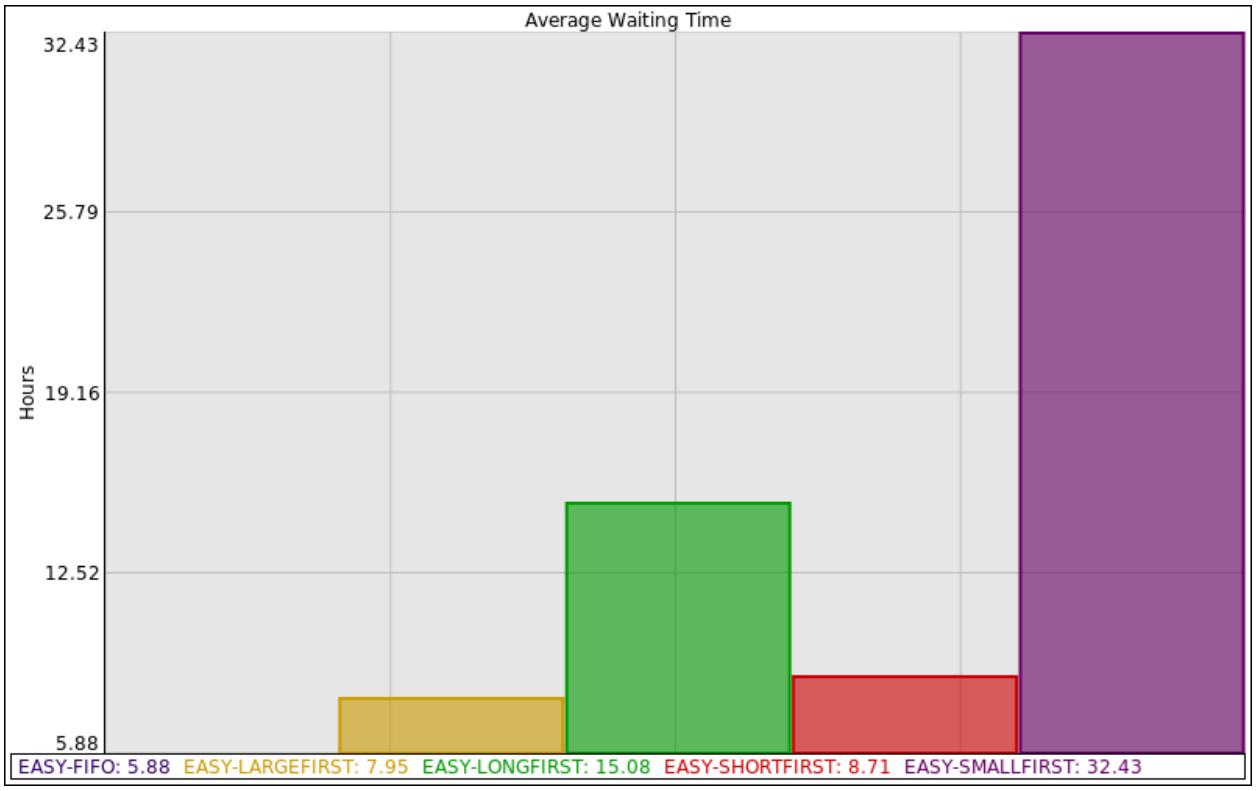


Fig 27. Resultado de la métrica *Waiting Time* mediante EASYScheduler.

Mediante la anterior gráfica podemos comprobar que, aplicando la técnica de backfilling, todas las políticas de job scheduling han mejorado sus resultados, de forma espectacular, en relación a los obtenidos por la implementación PQScheduler, pudiendo pasar a hablar sólo de magnitud horas y no de meses. Además, hay algoritmos que han experimentado un cambio en la relación que tenían con respecto a los otros. Por ejemplo, para los datos de workload que estamos analizando, vemos que EASY-FIFO obtiene mejor resultado que EASY-SHORTFIRST.

Las mejoras más importantes se dan en los comparadores EASY-LONGFIRST y EASY-LARGEFIRST, por este orden. Por tanto, podemos inferir que la técnica de backfilling consigue mitigar en gran medida la fragmentación interna introducida por PQScheduler.

Podríamos decir que, al introducir la técnica de backfilling, aun siendo más complejo el algoritmo de planificación, se ha conseguido aumentar el rendimiento del sistema para cualquiera de los algoritmos implementados, eliminando los tiempos de espera y acortando por tanto el tiempo total necesario para finalizar todos los trabajos.

7 Conclusiones

El estudio de las políticas de job scheduling en sistemas multiprocesador, en donde los trabajos sometidos son ejecutados concurrentemente, es una tarea muy compleja y está sometida a multitud de variables que deben tenerse en cuenta para poder determinar la idoneidad de una determinada política de planificación; empezando por obtener una muestra representativa de los datos a analizar. Precisamente por no disponer de todas estas variables, en la elaboración del análisis de las métricas obtenidas me he visto obligado a introducir conceptos teóricos, como el de fragmentación de recursos o grado de paralelismo, para poder explicar/desarrollar algunos resultados. Por ejemplo, para la métrica *Turnaround*, a pesar de que el algoritmo/comparador PQS-LONGFIRST obtiene un resultado mucho mayor (peor resultado) que el resto de comparadores, no obtiene un resultado muy inferior a los otros para la métrica *Throughput* (productividad). Esto ha sido posible explicarlo teniendo en cuenta el nivel de paralelismo al que han sido sometidos los datos del workload en SST-Simulator pero que, en este estudio, no se posee de dicha información.

En general, podernos convenir en que la implementación del algoritmo EASY, utilizando la técnica de backfilling, obtiene mejores resultados que el algoritmo Prioritario para todas la métricas obtenidas en el sistema simulado y, por tanto, un mejor rendimiento en su conjunto. De entre los diferentes comparadores analizados para EASY y, para el workload (SDSC) utilizado (asumimos que es una muestra representativa de la cadencia de trabajo del sistema IBM SP2 que lo generó) podemos observar que hay dos de ellos que ofrecen unos mejores resultados con respecto al resto; estos son: EASY-FIFO y EASY-LARGEFIRST, maximizando el repartimiento, disponibilidad y aprovechamiento de recursos, como demanda el sistema de procesamiento de cálculo masivo (no interactivo, etc.) estudiado. Este resultado, en parte, podría explicarse teniendo en cuenta la gráfica *job size* y *Scattered plot of runtime and job size* del workload (ver: The San Diego Supercomputer Center (SDSC) SP2 log) en donde puede observarse que existe una cierta uniformidad de recursos de CPU utilizados por los jobs, no provocando excesiva fragmentación y la apreciable tendencia de que los jobs con mayor demanda de CPU disponen de un menor *runtime*.

Para obtener las métricas, utilizadas como base del estudio comparativo, se ha realizado una aplicación en Perl para calcularlas, a partir de los ficheros .time generados por SST-Simulator al procesar el workload de SDSC y, para cada uno de los algoritmos/comparadores considerados en este estudio.

Finalizar comentando que la realización de este TFG me ha proporcionado un mayor conocimiento sobre las políticas de planificación de procesos y que ha sido muy gratificante aprender el lenguaje de computación Perl y adentrarme en el estudio de una parte tan fundamental del desarrollo de los Sistemas Operativos como son las job scheduling policies.

8 Referencias

Todas las consultas online fueron realizadas, en un momento u otro, durante el transcurso de desarrollo del proyecto, entre los meses de marzo y junio del 2015.

[AGING 1]: *Aging (scheduling)* [online], Wikimedia Foundation, <http://en.wikipedia.org/wiki/Aging_(scheduling)>.

[DES 1]: *Discrete Event Simulation* [online], Wikimedia Foundation, <http://en.wikipedia.org/wiki/Discrete_event_simulation>.

[EASY 1]: *Job Scheduler EASY* [online], PDC Center For High Performance Computing <https://www.pdc.kth.se/resources/software/job-scheduler/easy-main>.

[HPC 1]: *Supercomputer*, [online], Wikimedia Foundation, <http://en.wikipedia.org/wiki/Supercomputer>.

[JSCH 1]: *Computer scheduling*, [online], Wikimedia Foundation, <http://en.wikipedia.org/wiki/Scheduling_(computing)>.

[MESYS 1]: Mesh networking/system, [online], Wikimedia Foundation, <http://en.wikipedia.org/wiki/Mesh_networking>.

[MPI 1] *Message Passing Interface* [online], Wikimedia Foundation, <http://en.wikipedia.org/wiki/Message_Passing_Interface>.

[OPEN 1] *Open Source* [online], Open Source Initiative < http://opensource.org/>.

[Perl 1]: *Perl Computer Language* [online], Wikimedia Foundation, <http://en.wikipedia.org/wiki/Perl>.

[PNG 1]: *Portable Network Graphics* [online], Wikimedia Foundation, <http://es.wikipedia.org/wiki/Portable_Network_Graphics>.

[PREE 1]: *Preemption* [online], Wikimedia Foundation, <https://en.wikipedia.org/wiki/Preemption_(computing)> .

[PSTATE 1]: Process State [online], Wikimedia Foundation, <http://en.wikipedia.org/wiki/Process_state>.

[PWA 1]: *Parallel Workloads Archive* [online], The Hebrew University of Jerusalem <http://www.cs.huji.ac.il/labs/parallel/workload/index.html>.

[RACH 1]: University of Illinois [online], <http://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/6_CPU_Scheduling.html>.

[SST 1]: *The Structural Simulation Toolkit* [online], Sandia National Laboratories, <http://sst.sandia.gov/>.

[STARV 1]: *Resource starvation* [online], Wikimedia Foundation, <http://en.wikipedia.org/wiki/Resource_starvation>.

[VBOX 1] *Oracle VM VirtualBox* [online], Oracle Corporation, <https://www.virtualbox.org/>.

[Wiki 1]: *Systems development life cycle* [online], Wikimedia Foundation, <http://es.wikipedia.org/wiki/Systems_Development_Life_Cycle >.

[Wiki 2] *Linux operating system* [online], Wikimedia Foundation, <http://en.wikipedia.org/wiki/Linux>.

[ZOLT 1] *Parallel Partitioning, Load Balancing and Data-Management Services* [online], Sandia National Laboratories < http://www.cs.sandia.gov/zoltan/>.