Enseñanza de Programación Paralela y Distribuida en las Carreras de grado de Computación

Autor
Marcelo Arroyo
Universidad Nacional de Río Cuarto
marroyo@dc.exa.unrc.edu.ar

Resumen

En general la enseñanza de concurrencia, paralelismo y sistemas distribuidos se dicta a un nivel introductorio en las carreras de computación en casi todo el mundo. Los temas avanzados sobre estos tópicos se abordan o en cursos avanzados optativos o de posgrado específicos. En este artículo se justifica la necesidad de lograr el *"parallel thinking"* desde el inicio de la enseñanza en carreras de computación y se propone la inclusión de un conjunto de temas relacionados a estos conceptos en forma transversal en cualquier plan de estudios vigente de una carrera de grado en ciencias de la computación o similar.

Palabras clave: Programación, Concurrencia, Paralelismo,Sistemas Distribuidos, Educación.

Introducción

Tradicionalmente, las carreras de ciencias de la computación, ingeniería de software y carreras afines, enfocan la enseñanza de la concurrencia y paralelismo y sus problemas relacionados, como el abordaje de estos conceptos en cursos avanzados específicos tales como sistemas operativos, sistemas distribuidos, ingeniería de software avanzados (ejemplo: mediante la enseñanza de patrones de diseño concurrentes) y a veces como un paradigma de programación adicional a los demás paradigmas clásicos de programación como por ejemplo, el imperativo, funcional, orientado a objetos y lógico.

Los avances tecnológicos en el campo del hardware requiere que actualmente se tengan que revisar los planes de estudio de las carreras de computación ya que deberíamos preparar a las nuevas generaciones de profesionales, docentes y hasta los usuarios con una visión que un sistema de computación actual es naturalmente paralelo y su programación , como también su enseñanza, debe dejar de ser principalmente secuencial.

Sin embargo, es difícil encontrar que conceptos de concurrencia, paralelismo y sistemas distribuidos se incluyan en cursos de grado como estructuras de datos y algoritmos y otros cursos como técnicas de resolución de problemas mediante algoritmos en general, cursos que generalmente se dictan en los primeros años de las carreras de computación.

Esto es comprensible ya que tiene su razón histórica: Las computadoras personales y mini-computadoras tenían hasta hace pocos años pocas facilidades para resolver problemas en forma paralela, por lo que las arquitecturas paralelas eran algo difícil de encontrar en el ámbito académico y profesional. Esto hacía que la enseñanza de la programación paralela y distribuida fuese considerado un tema muy específico de interés sólo a aquellos que podían acceder a hardware específico no muy común de encontrar por la mayoría de profesionales, docentes y alumnos de esos años.

Actualmente, esto ha cambiado drásticamente. La situación prácticamente se ha invertido y sucede que desde una PC, pasando por las tabletas, consolas de juegos y aún los teléfonos inteligentes, podemos encontrar hardware

paralelo (multicores, hyperthreading, pipelines, procesadores con instrucciones vectoriales, jerarquías de memoria, diferentes tipos de interfaces de red, etc) y software de base brindando facilidades para la programación concurrente y paralela.

Los avances en el software, principalmente en los lenguajes de programación, han evolucionado más lentamente con respecto a los avances del hardware, muchas veces como extensiones algo artificiales a lenguajes de programación originalmente diseñados para el procesamiento secuencial.

El concepto de programación paralela, es la forma de explicitar cómo diferentes secciones de un programa pueden ser ejecutadas en diferentes unidades de procesamiento.

El desarrollo de nuevas abstracciones de control para sistemas concurrentes es un tema de activa investigación en todo el mundo y hasta ahora no hay un consenso global sobre los modelos y mecanismos mas adecuados para el desarrollo de sistemas paralelos. Esto hace que existan diferentes modelos de programación para arquitecturas paralelas y distribuidas. Principalmente, los modelos propuestos se debaten en aquellos basados en memoria compartida (la cual constituye un mecanismo de comunicación entre procesos o threads) y los sistemas distribuidos (comunicaciones entre procesos por medio de mensajes).

A pesar que casi todos los lenguajes de programación modernos (C++11, Java y otros) ofrecen abstracciones y mecanismos para la implementación de sistemas concurrentes y/o paralelos, es curioso lo poco que se aplican en el desarrollo de talleres y prácticos de laboratorio en la mayoría de los cursos de las carreras de grado.

Una visión de un sistema de computación como un sistema naturalmente paralelo requiere que los estudiantes comiencen resolver problemas algorítmicos con una mentalidad mas natural y alejada del clásico pensamiento secuencial. En las naciones con mayor desarrollo tecnológico se habla de conseguir esta mentalidad bajo el término de *parallel thinking*.

El proyecto de la NFS/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing[1] – en el cual el autor participa como un *early adopter* desde 2013[8] [9] – tiene como objetivo proponer un núcleo de temas curriculares en computación paralela y distribuida (PDC) que deberían cubrirse en cualquier carrera de grado en ciencias de la computación e ingeniería en computadoras. En el marco de esta iniciativa, en el que participan docentes e investigadores de todo el mundo, se han desarrollado un conjunto de documentos que contienen definiciones y propuestas de cómo conseguir en los estudiantes de computación el *parallel thinking* en la enseñanza de grado.

En este sentido, creemos que la enseñanza impartida a los futuros desarrolladores de software debe ir a un enfoque mas natural del pensamiento de resolución de problemas en forma paralela y distribuida desde los inicios de su formación.

Para lograr estos objetivos, es necesario revisar los contenidos de los planes de estudio y la forma en que enseñamos computación y resolución de problemas mediante algoritmos desde los primeros momentos.

En las próximas secciones se abordarán temas sobre como lograr el *thinking in parallel*, sus posibles abordajes desde los inicios de una carrera de grado, los tópicos clásicos que afectan y cómo habilitan la inclusión directa en la enseñanza de conceptos sobre concurrencia y paralelismo, para finalmente proponer un modelo de inclusión de conceptos de concurrencia, paralelismo y sistemas distribuidos en forma transversal en un plan de estudio estándar actual de ciencias de la computación u otras carreras afines.

Pensamiento secuencial

A pesar que los problemas de la concurrencia y el paralelismo fueron uno de los primeros temas de investigación en las ciencias de la computación, ya que surgió como necesidad, principalmente en el campo de los sistemas operativos, los avances en la temática específica y los aprendizajes obtenidos no se han volcado naturalmente y con la profundidad requerida en las currículas de grado en las carreras de computación. Generalmente esos tópicos se abordan en cursos específicos avanzados o en cursos electivos.
En cursos de programación funcional o algorítmica es común describir informalmente la semántica de una operación *map f c* (la cual aplica la función *f* a cada elemento del contenedor *c)* en términos de una similaridad con una sentencia de iteración definida en un lenguaje de programación imperativo (C, Pascal, Java, ...), como por ejemplo:

for i:=0 to i<N do f c[i]

Esta descripción, basada en una semántica operacional secuencial, es absolutamente artificial ya que asume que será ejecutada o evaluada por un único procesador secuencial.
La visión mas natural (y mejor fundamentada conceptualmente) es que *f* se aplica en paralelo a cada elemento ya que no hay ninguna dependencia de datos que lo impida.

Análogamente, si le planteáramos a un grupo de niños que deben tomar su lápiz y papel y comenzar a dibujar un círculo, a ninguno se le ocurriría que debería esperar a los que están a su lado a que terminen para comenzar con su trabajo[1], ya que ninguno requiere acceder a los recursos de otro.

El pensamiento secuencial surge de la hipótesis que hay un único procesador que ejecuta instrucciones. Hipótesis que cada vez es menos válida con los sistemas de computación actuales y que ya prácticamente son cosas del pasado.

Sin embargo, a pesar de esta realidad, generalmente seguimos enseñando resolución de problemas mediante algoritmos en forma estrictamente secuencial.

¿Qué sucedería si desde el comienzo se le presentara a los alumnos que un sistema de computación paralelo? (lo que simplemente requeriría presentarle cualquier sistema de cómputo que conozca). Esto haría que los alumnos cuestionaran la secuencialidad de determinadas soluciones y de una manera más natural surgieran soluciones basadas en algoritmos paralelos.

Esto nos obligaría (a los docentes) a repensar la enseñanza de ciertos tópicos, como algoritmos y estructuras de datos, arquitecturas de sistemas de computación, paradigmas de programación, semántica de lenguajes de programación, compiladores, etc.

Por supuesto que la incorporación del pensamiento paralelo requiere prestar atención a los problemas que acarrea la concurrencia y el paralelismo, como lo son las condiciones de carrera, deadlocks, livelocks, y otros problemas conocidos.

Según la experiencia del autor - quien ha dictado cursos de paradigmas de lenguajes de programación, sistemas operativos y sistemas distribuidos en cursos obligatorios de grado y cursos de programación paralela de posgrado - mientras estos conceptos se demoren más en presentarse, es mas difícil que el alumno adquiera el pensamiento paralelo.

Otros investigadores han comprado la situación actual del problema del *shifting to parallel programming* con la resistencia a la adopción de la programación estructurada hace unos 40 años atrás.

1 Problema que algorítmicamente puede modelarse con *map*.

El desarrollo de software actual requiere del dominio de conceptos como arquitecturas cliente-servidor y otras arquitecturas de software distribuido, alta disponibilidad, escalabilidad ante grandes demandas de requerimientos, entre otros. Esto requiere que un profesional o técnico en computación pueda dominar claramente conceptos de concurrencia, paralelismo y sistemas distribuidos. Las companías están adoptando cada vez con mayor naturalidad sistemas en la web de la forma *software as a service*, computación colaborativa, bases de datos distribuidas y replicadas, sistemas tolerantes a fallas, sistemas de control con redes de sensores y actuadores y otros tipos de sistemas complejos. El desarrollo de software en estos escenarios requiere el desarrollo y uso de complejos frameworks que incluyen mecanismos de concurrencia, control de acceso a recursos, distribución de computaciones y el uso de paradigmas de programación concurrentes y distribuidos.

El conocimiento sobre ciertos conceptos sobre PDC y la adquisición de habilidades de su aplicación permitirá que un egresado pueda no sólo utilizar las abstracciones que ofrece un framework o lenguaje de programación paralela y/o distribuida, sino también desarrollar otras abstracciones necesarias para la solución de problemas específicos.

Generalmente, los objetivos de un plan de estudio de una carrera de grado es la formación de un profesional, docente o investigador que pueda perdurar en el tiempo a lo largo de toda su carrera.
Tal vez sea tiempo de pensar que debemos revisar algunos aspectos de la formación de grado.

En la próxima sección se describen algunos conceptos de PDC que deberían incluirse en algún punto de un plan de estudios de una carrera de grado de computación.
En las secciones siguientes se propone un modelo de inclusión de temas de PDC en cursos de cualquier plan de estudios estándar y algunas consideraciones sobre su enfoque con el objetivo de no incrementar las cargas horarias que impidan su implementación.

Conceptos sobre PDC

A continuación se enumeran los principales conceptos y términos sobre PDC requeridos que cualquier egresado de una carrera de grado debería conocer con algún grado de profundidad:

- *Arquitecturas paralelas*: Clasificación de Flynn y su relación a arquitecturas modernas. Jerarquías de memoria. Buses y protocolos de de comunicación.
- *Tarea, proceso, thread*: Sección independiente de ejecución que realiza un trabajo específico.
- *Recurso*: objeto utilizado por una tarea, proceso o thread. Un recurso puede ser un área de memoria (ej: una variable de programa), un dispositivo, un mensaje, etc.
- *Tareas independientes y cooperativas o comunicantes*: Un grupo de tareas cooperativas o comunicantes comparten recursos y/o intercambian información para lograr un objetivo común.
- *Planificación de uso de CPU (scheduling)*: Asignación de las CPUs a las tareas. Progreso.
- *Ejecución paralela y concurrente*: Interleaving, trazas de ejecución, no-determininismo. Race conditions.
- *Sincronización*: Las tareas cooperativas o comunicantes requieren de algún mecanismo de control de acceso a los recursos compartidos. Deadlocks, livelocks.
- *Aceleración*: razón entre el tiempo de ejecución secuencial y el tiempo de ejecución paralela.
- *Sobrecarga*: Trabajo extra requerido por el sistema para la creación,

destrucción, planificación y sincronización de tareas.
- *Escalabilidad*: Posibilidad de obtener ganancias de aceleración con una mayor cantidad de componentes paralelos.
- *Programación concurrente y distribuida*: Modelos de memoria compartida vs sistemas distribuidas (basados en mensajes). Comparación y equivalencia de modelos.

Integración de conceptos de PDC en un plan de estudios de grado

Los temas descriptos en la sección anterior deberían incluirse en algunos cursos de un plan de estudios estándar de una carrera de computación.
A continuación se presenta una relación entre temas de PDC con cursos de un plan de estudios estándar. Además se incluyen algunas recomendaciones sobre el abordaje de algunso temas.

Arquitecturas de computadoras:

- ✔ Clasificación de Flynn y reconocimiento de componentes paralelos en arquitecturas modernas (ej: GPUs como SIMD. Multicores o multiprocesadores como MIMD. Pipelines. Hyperthreading).
- ✔ Jerarquías de memoria: Memoria RAM. Niveles de cachés. Impacto de caché en la localidad de instrucciones y datos. Impacto en el scheduling de sistemas de multiprocesamiento con respecto a la caché. Problemas de afinidad de cpu.
- ✔ Modelos de hardware con memoria compartida y E/S dedicada. Comparación con modelos de programación de memoria compartida y basado en mensajes.
- ✔ Buses de comunicación. Protocolos de bajo nivel de comunicación entre dispositivos. Interconectividad de dispositivos (buses y protocolos).
- ✔ Instrucciones atómicas (ej: swap en x86). Atomicidad en sistemas multiprocesadores.
- ✔ Visión del diseño de un sistema moderno (PC, smartphone, consola de juegos, etc) como sistemas de computación paralelo y su relación con la clasificación de Flynn.

Diseño de algoritmos y estructuras de datos:

- ✔ Grafo de tareas. Dependencias de datos.
- ✔ Técnicas de diseño de algoritmos y oportunidades de concurrencia como divide-and-conquer y su relación con técnicas de paralelización basadas en descomposición recursiva (de datos o computaciones), algoritmo voraces (y sus posibles implementaciones paralelas), algoritmos de búsqueda (resaltar sus posibilidades de paralelización en cada uno de ellos), programación dinámica, etc.
- ✔ Acceso concurrente a estructuras de datos. Sincronización en el acceso a recursos compartidos. Invariantes de representación en presencia de concurrencia. Estructuras de datos thread-safe. Consideraciones de diseño de estructuras de datos lock-free.

Lenguajes de programación:

- ✔ Abstracciones linguísticas de gestión de tareas, threads y procesos.
- ✔ Abstracciones sintácticas y semántica de canales de comunicación.
- ✔ Patrones de control paralelos. Ejemplos: fork-join, C++11 *futures*, etc.
- ✔ Oportunidades de implementación paralela de abstracciones de control. Ejemplos: *map* en programación funcional, ciclos, funciones o bloques independientes en programación imperativa y orientada a objetos.

- ✔ Implementación de abstracciones paralelas y/o distribuidas en el paradigma de programación genérica. Ejemplo: implementación o uso de parallel/distributed skelletons con C++ templates.

Ingeniería de software:

- ✔ Patrones de diseño concurrentes. Ejemplos: Map-reduce, fork-join, productor-consumidor, pipeline, thread-pool y otros.
- ✔ Arquitecturas de Sistemas distribuidos: Master-slave, cliente-servidor, thread-pool, peer-to-peer, etc.
- ✔ Nociones de escalabilidad.
- ✔ Sistemas tolerantes a fallas.
- ✔ Modelado de sistemas concurrentes y distribuidos.
- ✔ Uso de herramientas de control de versiones distribuidas. Análisis de su diseño y funcionamiento. Ejemplo: Git.

Bases de datos:

- ✔ Procesamiento concurrente de consultas.
- ✔ Bases de datos distribuidas.
- ✔ Replicación y consistencia.
- ✔ Transacciones concurrentes y distribuidas.

Compiladores:

- ✔ Optimizaciones para concurrencia: Dependencia de datos (análisis data-flow). Reordenamiento de instrucciones (para hyperthreading). Implementación de primitivas de sincronización.
- ✔ Generación de código concurrente: Basado en directivas (ejemplo: OpenMP) o implícita (paralelización de ciclos y bloques secuenciales independientes).
- ✔ Generación de código SIMD. Ejemplos: generación de instrucciones SSE en x86 o código CUDA/OpenCL para GP-GPUs.
- ✔ Localidad de datos e instrucciones. Impacto en el uso eficiente de caché.

Sistemas operativos:

- ✔ Arquitecturas paralelas. SMP y AMP.
- ✔ Jerarquías de memoria: UMA y NUMA.
- ✔ Implementación de bajo nivel de mecanismos de sincronización. Exclusión mutua, variables de condición, semáforos, monitores.
- ✔ Problemas ocasionados por la concurrencia: deadlock, livelocks, starvation.
- ✔ Modularidad y primitivas de sincronización.
- ✔ Planificación de uso de CPU: scheduling.
- ✔ Planificación de uso de recursos compartidos: Algoritmos de elevador, spoolers (como instancias concretas del patrón productor-consumidor).
- ✔ Multiplexado.
- ✔ Arquitecturas de software distribuidas.
- ✔ Máquinas virtuales.

Redes y sistemas distribuidos:

- ✔ Protocolos de comunicación.
- ✔ Arquitecturas paralelas alta y débilmente acopladas. Multiprocesadores y clusters.
- ✔ Implementación de primitivas de comunicación: sincrónicas y asincrónicas. Uso de mecanismos de mensajes sincrónicos para sincronización de computaciones.
- ✔ Sistemas distribuidos. Arquitecturas de software: cliente-servidor, master-slaves, thread-pools, peer-to-peer.
- ✔ Identificación de procesos.
- ✔ Patrones de comunicación: map-reduce, scatter-gather y otros.
- ✔ Middleware: MPI, CORBA, J2EE, RPC, etc.

- ✔ Algoritmos distribuidos: exclusión mutua, memoria compartida distribuida, estado global, toma de instantáneas, transacciones distribuidas, replicación y consistencia, tolerancia a fallas.
- ✔ Balance de carga.
- ✔ Estructuras de datos distribuidas.
- ✔ Análisis de algoritmos distribuidos. Ley de Amdahl y escalabilidad.

Teoría de lenguajes, computabilidad y complejidad:

- ✔ Modelos teóricos de computación paralela (PRAM y otros).
- ✔ Análisis de algoritmos paralelos.
- ✔ Métricas de complejidad para algoritmos paralelos y distribuidos.

Otros temas podrán dictarse con mayor profundidad en cursos electivos o específicos sobre tópicos de PDC. La inclusión de los temas descriptos permitirá que el estudiante de grado adquiera los conceptos básicos requeridos para lograr de forma incremental el *pensamiento en paralelo* a partir de su formación inicial.

Impacto de los modelos de programación paralela y distribuida en otras áreas

En los últimos años se han propuesto y desarrollado muchos modelos y estilos de programación para los diferentes arquitecturas paralelas y distribuida. Los primeros mecanismos propuestos y ampliamente utilizados como threads, uso de locks, semáforos, monitores y mensajes parecen ser de demasiado bajo nivel y poco adecuados para la formulación y descripción de algoritmos y sistemas complejos. Otros modelos se basan en que los aspectos de concurrencia y distribución de datos y computaciones deberían ser implícitas o descriptas exteriormente al programa en algún lenguaje de *configuración del sistema*. Ambos enfoques parecen no cubrir las necesidades prácticas. Los mecanismos de bajo nivel permiten la inclusión de errores muy frecuentemente. De hecho, los problemas como condiciones de carrera son los más frecuentes en los errores reportados en sistemas operativos modernos como GNU-Linux, MS-Windows y otros. Además estos mecanismos generalmente afectan a la portabilidad de las aplicaciones. Es muy común ver que las aplicaciones que explotan el hardware de GP-GPUs sólo están orientadas a un modelo de hardware en particular, inclusive con lenguajes de programación propietarios desarrollados por los fabricantes de esos dispositivos.

Los modelos de alto nivel no logran conseguir el rendimiento esperado que puede lograrse con paralelismo explícito.

Esto motiva el hecho que los modelos, mecanismos y abstracciones necesarias para la programación de sistemas paralelos y distribuidos sea una de las áreas mas activas de investigación actual.

La tendencia actual es el desarrollo de abstracciones de alto nivel que se basan en conceptos de la programación funcional como modelo de programación mas adecuada para sistemas masivamente paralelos. El estilo de programación con transparencia referencial es mas adecuado para describir computaciones paralelas ya que simplifica la detección de dependencias de datos para lograr el mayor paralelismo posible y permite un razonamiento ecuacional en el análisis de programas.

Algunos nuevos paradigmas de programación, tales como la programación genérica (o programación generativa), están tomando interés en forma creciente ya que permiten describir *patrones* de computaciones con instanciaciones específicas que resultan en optimizaciones para la plataforma de hardware-software destino. Estas optimizaciones son generalmente a nivel de aplicación y permite que el código fuente sea altamente portable. Es común encontrar bibliotecas de plantillas C++ que implementan

esas abstracciones como los *parallel standard containers* de la biblioteca estándar de GNU-GCC[3].

Otras bibliotecas de alto nivel se presentan como esqueletos paralelos (*parallel skeletons*) [4] los cuales son plantillas (templates) de abstracciones de alto nivel (estructuras de datos y algoritmos paralelos/distribuidos) los cuales, utilizando técnicas de meta-programación, permiten la composición de programas paralelos y la generación de código eficiente.

Estas bibliotecas basadas en programación genérica utilizan un estilo funcional de alto orden y parecen ser las abstracciones más prometedoras para la implementación de programas complejos y masivamente paralelos.

El estilo utilizado por estas bibliotecas y los nuevos mecanismos que podemos encontrar en otros lenguajes de programación modernos como Erlang[5], por ejemplo, proponen un estilo de programación funcional (computaciones sin efectos colaterales y uso de alto orden), el cual parece ser el paradigma de programación mas adecuado de programación paralela.

Las implementaciones de mecanismos de concurrencia en lenguajes de programación funcional modernos como Haskell, ML, Erlang, Scala, etc, y su rápida adopción en la academia e industria permiten concluir que el paradigma funcional tendrá cada vez más mayor protagonismo en el desarrollo de sistemas paralelos y distribuidos.
Esto nos debería hacer reflexionar sobre la inclusión y la vigencia de contenidos de programación funcional moderna en cualquier plan de estudios de grado de una carrera de computación. Los alumnos que conocen los principios básicos de la programación funcional y han adquirido las habilidades prácticas de desarrollo de programas, encuentran en estas herramientas (lenguajes de programación y bibliotecas) un estilo de programación familar.

Algunas consideraciones sobre la enseñanza de PDC

Sería recomendable que desde el primer año de la enseñanza de la programación no se fuerce el pensamiento secuencial artificial. A modo de ejemplo, es común encontrar que se desarrollan algoritmos de ordenamiento. Algunos de ellos, como por ejemplo, el *quicksort*, el cual contiene pasos naturalmente paralelos, ya que su diseño se basa en una descomposición recursiva de operaciones, lo cual es compatible directamente con una de las principales técnicas de diseño de algoritmos paralelos.

No se pretende que los alumnos tengan que implementar versiones paralelas, pero sí sería deseable hacerles notar que podría ejecutarse en forma paralela, en caso de disponer de una plataforma multiprocesador y poder analizar la ganancia potencial.

En los cursos más avanzados de diseño de algoritmos y estructuras de datos sería deseable que se introduzca al menos el modelo de programación de memoria compartida y se realicen algunas prácticas de programación utilizando lenguajes, bibliotecas y/o frameworks de programación concurrente.

En cursos de paradigmas de programación sería deseable el estudio de la concurrencia no como un paradigma alternativo sino como algo transversal a todos los demás paradigmas existentes. Además, en este tipo de cursos es donde sería altamente recomendable plantear las oportunidades de implementación paralela de abstracciones y mecanismos de control en diferentes paradigmas. Por ejemplo en programación funcional, funciones de alto orden como *map* y *fold* y algún posible esquema de su implementación paralela.
Del mismo modo se podría hacer notar que en un lenguaje relacional como prolog, se podrían

evaluar cláusulas en paralelo si sus argumentos son independientes y que la dependencia de datos (argumentos de cláusulas) imponen restricciones secuenciales.

En los cursos sobre arquitecturas de procesadores se deberían presentar temas y hacer prácticas sobre instrucciones vectoriales, el impacto de la localidad de datos e instrucciones en la performance, gracias a tener un mayor número de *hits* y conceptos generales de arquitecturas paralelas como multicores, pipelines e hyperthreaading.
También sería conveniente la introducción de estructura y funcionamiento de las jerarquías de memoria y los mecanismos de interrupciones en sistemas multiprocesadores.

En cursos de Sistemas Operativos, la práctica con implementaciones concretas utilizando sistemas operativos educativos como xv6[6] o similares, permite que el alumno reconozca las dificultades que genera la concurrencia y el paralelismo y comprenda en detalle los mecanismos provistos por el hardware y las implementaciones de primitivas en software para la gestión de procesos, mecanismos de comunicación entre procesos, uso de recursos compartidos, etc.

La enseñanza de estos conceptos en forma abstracta dista mucho de una formación integral por lo que debería estar complementada con actividades prácticas y de laboratorio que permiten afianzar la comprensión de los conceptos y las dificultades de su correcta y eficiente implementación.

En la Universidad Nacional de Río Cuarto, el autor ha estado utilizando xv6, un sistema operativo tipo UNIX desarrollado en el MIT con fines educativos para el desarrollo del taller en el curso Sistemas Operativos desde hace tres años con resultados muy alentadores, ya que los alumnos integran muchos conceptos de otros cursos anteriores, como estructuras de datos y algoritmos, arquitectura de procesadores, paradigmas y lenguajes de programación, compiladores y otros, para poder resolver los problemas planteados en el transcurso de la materia. Los problemas asociados a la concurrencia son los que les acarrean mayor dificultad aunque los conceptos generales ya habían sido cubiertos en cursos anteriores, lo cual deja en evidencia que la práctica concreta es muy importante. En este curso, los conceptos se afianzan porque los problemas de la concurrencia emergen como errores comunes, ante los cuales los alumnos tienen que diagnosticar y resolver.

Conclusiones

Los avances vertiginosos de la tecnología nos obliga a incluir temas en un plan de estudios de una carrera de grado ya que se tornan comunes en cualquier sistema de cómputo actual.

Las arquitecturas paralelas, y con diferentes modelos de paralelismo (MIMD, SIMD, etc), ya las tenemos al alcance de todos en nuestras Pcs, laptops, smartphones, tablets, consolas de juegos y otros sistemas de computación utilizados habitualmente.

La existencia de esta tecnología requiere que las técnicas y modelos de programación de estos sistemas de cómputo ya no sean una cosa que se dictaba en cursos avanzados o de posgrado, sino que deberían incluirse en cualquier plan de estudios de una carrera de grado en computación o similar.

Dentro de la comunidad de investigadores y educadores en computación hay un consenso generalizado que los temas de PDC no deberían incluirse sólo en cursos específicos sobre PDC, sino que además, debería incluirse el pensamiento del paralelismo en forma transversal a lo largo de todo el plan de estudios y desde los primeros cursos profundizando los conceptos de manera incremental. En la bibliografía se incluyen otros artículos que describen experiencias en la implementación del *thinking in parallel* en carreras de grado.

En este artículo se presentaron las motivaciones y algunas recomendaciones para lograr el *parallel thiking* en una carrera de grado desde el inicio. Además se presentó una lista de temas relacionados con PDC y sus relaciones con los cursos existentes en un plan de estudios estándar, proponiendo de qué manera se pueden incluir esos temas en las asignaturas correspondientes.

En nuestra opinión, el abordaje de esos temas no debería agregar más horas a los cursos actuales, sino que deberían ser incluidos revisando los objetivos de enseñanza-aprendizaje logrando planes de estudio con contenidos mas cercanos a las demandas y realidades actuales.

Bibliografía

[1] NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing – Core Topics for Undergraduates. URL:http://www.cs.gsu.edu/~tcpp/curriculum/

[2] NSF/IEEE–TCPP Curriculum Initiative. (2013) Call for proposals: Book project on parallel and distributed computing topics for undergraduate core courses. [Online]. Available: http://www.cs.gsu.edu/~tcpp/curriculum/?q=CFP_Book_Project

[3] GNU-GCC libstdc++ parallel mode. https://gcc.gnu.org/onlinedocs/libstdc++/manual/parallel_mode.html

[4] Parallel Skeletons.

[5] Programming Erlang: Software for a Concurrent World. Second Edition. Joe Ambstrong. ISBN: 978-1937785536. Pragmatic Programmers. 2013.

[6] XV6, a simple Unix-like teaching operating system. Russ Cox, F. Kaashoek, R. Morris. 2014. http://pdos.csail.mit.edu/6.828/2014/xv6.html

[7] Integrating Parallel Programming Techniques into Traditional Computer Science Curricula. John R. Graham. ACM, Inroads, SIGCSE Bulletin. Volume 39, number 4, pp. 75-78. Dec, 2007.

[8] K. Prasad, A. Gupta, K. Kant, A. Lumsdaine, D. Padua, Y. Robert, A. Rosenberg, A. Sussman, C. Weems et al. "Literacy for all in parallel and distributed computing: guidelines for an undergraduate core curriculum," 2012.

[9] Marcelo Arroyo. "Teaching Parallel and Distributed Computing topics for the Undergraduate Computer Science Student". Third NSF/TCPP Workshop on Parallel and Distributed Computing Education (EduPar-13). Boston, USA. May 20, 2013.

[10] E. Orozco, R. Arce-Nazario, J. Ortiz-Ubarri, H. Ortiz-Zuazaga. "A Curricular Experience With Parallel Computational Thinking: A Four Years Journey". EduPDHPC. Workshop en Education for High Performance Computing. 2013.