Sincronización Conservadora en Simulación Paralela

Rodrigo Miranda Mauricio Marín

Departamento de Computación
Universidad de Magallanes
Punta Arenas, Chile
Contact Author: mmarin@ona.fi.umag.cl

Abstract

En este trabajo describimos y evaluamos una implementación BSP del protocolo de sincronización YAWNS para simulación discreta en paralelo y lo comparamos con un protocolo denominado Ventana Fija. Este último es la estrategia intuitiva, de fácil implementación, a la que uno recurre cuando desea simular un sistema discreto. Por otra parte, YAWNS es un algoritmo más complicado y por lo tanto es importante conocer bajo qué condiciones conviene hacer el esfuerzo de implementarlo para un sistema dado. En este artículo investigamos este tema con implementaciones reales de ambos algoritmos simulando un sistema de topología toroidal sobre un cluster de PCs. Además, presentamos una comparación entre las dos principales librerías utilizadas para implementar paralelismo en simuladores: PUB (*Paderborn University Bsp*) y MPICh (*MPI Chameleon*).

1 Introducción

La simulación computacional por eventos discretos es una técnica de análisis de sistemas utilizada cuando éstos son demasiados complejos para ser modelados en forma analítica o numérica. En estos sistemas, tanto los objetos como los eventos que ocurren a través del tiempo son representados con estructuras de datos que van siendo almacenados y ejecutados conforme avanza un tiempo virtual de simulación.

Cuando se realizan simulaciones demasiado grandes, como por ejemplo sistemas de partículas, es posible complementar estas técnicas de análisis con el uso de paralelismo, que consiste en repartir cargas de trabajo entre varios procesadores, lo que permite obtener mejores tiempos de ejecución. Se han diseñado varios algoritmos para implementar paralelismo en simuladores; cada uno de ellos permite obtener mejoras en el desempeño de la simulación bajo ciertos parámetros que varían según el algoritmo usado.

Actualmente existen ambientes de simulación que buscan ofrecer al usuario común los beneficios de realizar una simulación en forma paralela. Incluso existen aplicaciones que ofrecen al usuario la posibilidad de elegir el algoritmo que se usará para implementar el paralelismo. Sin embargo, resulta notoria la escasa difusión que ha tenido el uso del paralelismo entre los usuarios de simuladores. Es posible que esto se deba a los siguientes hechos:

- Actualmente, las aplicaciones utilizadas para realizar simulaciones están construidas sobre algoritmos que no son portables, es decir, son específicos para un tipo de arquitectura.
- En el caso de las aplicaciones que ofrecen múltiples algoritmos de simulación, estos no brindan orientación alguna al usuario sobre cual algoritmo resulta más conveniente para un modelo dado.

El objetivo de este trabajo es desarrollar una parte de un ambiente de simulación que permita elegir entre varios algoritmos de simulación. Principalmente, esta parte consiste en la implementación de dos protocolos de comunicación para simulaciones paralelas utilizando C++, y sus posteriores evaluaciones de desempeño.

Siguiendo un orden de importancia, el siguiente objetivo será evaluar dos librerías de comunicaciones frecuentemente utilizadas para el envío de mensajes entre procesos en paralelo corriendo en distintos procesadores: PUB y MPI.

2 Algoritmo YAWNS

2.1 Descripción

Este protocolo conservador basa su funcionamiento en que es posible avanzar la simulación entre sincronizaciones mientras no exista el riesgo de recibir un evento atrasado, proveniente de otro procesador. Para que esto no ocurra, YAWNS indica lo siguiente: cada vez que un objeto atiende un evento, debe realizar el cálculo del tiempo de servicio para el próximo evento que pueda ocurrir, sin simularlo. De esta forma, para un objeto de simulación i, si se definen s_i como el tiempo de servicio calculado, y $\alpha(i)$ como el tiempo de salida del próximo evento simulado, entonces:

$$\alpha(i) = max\{\beta(i), a_i\} + s_i$$

Siendo $\beta(i)$ el tiempo en que finalizará la atención del evento que actualmente se encuentre bajo simulación, a_i tiempo de llegada del próximo evento que requiera servicio. Así se tiene que el siguiente *superstep* se pueden simular todos los eventos e cuyo tiempo $e.t$ sea menor al mínimo $\alpha(i)$ de todos los objetos de simulación en el sistema.

Formalmente, el protocolo YAWNS sigue los siguientes pasos:

1. Obtener el menor de todos los $\alpha(i)$ calculados en todos los objetos de simulación a través de una sincronización de los procesos en paralelo. Este mínimo se denominará como α_m.
2. Procesar los eventos que satisfagan $\alpha(i) < \alpha_m$, siguiendo el modelo establecido.
3. Sincronizar los procesos, y asegurar la recepción de todos los mensajes en cada proceso lógico destino, es decir, que no hayan mensajes rezagados.
4. Calcular $\alpha(i)$ para cada uno de los objetos de simulación, y volver al paso 1.

2.2 Implementación

Para la implementación de este algoritmo se puso énfasis en los siguientes puntos:

- Escritura de un *kernel* para una simulación en paralelo que contemple los cálculos característicos del algoritmo YAWNS. Este *kernel* consta de tres ciclos: el primero ejecuta la simulación mientras no se haya llegado a un tiempo límite denotado por ENDSIM. El segundo se encarga de recorrer los objetos de simulación existentes en el procesador actual, y el tercero recorrerá la lista de eventos del objeto, hasta un límite dado por la ventana de simulación *TWindow*.

```
  while(GVT < ENDSIM)
  {
    // Recepción de mensajes
#include "recibe_mensajes.cc"

    // Cálculo del tiempo virtual global
#include "calcula_vt.cc"

    // Seccion del superstep
    // Se debe recorrer cada objeto
    // en este procesador
    for(it = mapa_disp.begin(); it != mapa_disp.end(); it++)
    {
      ptr = (*it).second;

      if(ptr != NULL)
      {
        // Recorrer la lista de eventos
        // del objeto
        while( ptr->GetTiempoSiguiente() <= TWindow )
        {
          // Se saca un evento de la lista
          // y se "atiende" o simula
          e = ptr->SiguienteEvento();

          ptr->Atiende(e->tiempo, e->evento, e->tipo,
                                (Mensaje *)e->msg);
          delete e;
        }
      }
    }

    com.SndMsg();

  // Sincronización necesaria para el
```

```
  // cálculo de TWindow
#include "recibe_mensajes.cc"

#include "calcula_window.cc"

    com.SndMsg();
  }
```

Es necesario incluir el archivo *recibe_mensajes.cc* por segunda vez en el ciclo, ya que se debe asegurar que todos los mensajes hayan llegado a su destino antes de empezar el cálculo de la siguiente ventana, según el algoritmo YAWNS.

El contenido del archivo *calcula_window.cc* es el siguiente:

```
// Cálculo de la siguiente ventana

  TWindow = INFINITO;
  la = INFINITO;

  map<string, Dispositivo *>::iterator it;

  // Recorrido de cada objeto en el Procesador
  for(it = mapa_disp.begin(); it != mapa_disp.end(); it++)
  {
    ptr = (*it).second;

    if(ptr != NULL)
    {
      // Obtener los parámetros que se necesitan
      st = ptr->ServiceTime();
      lst = ptr->LastServiceTime();

      // Obtener el tiempo del siguiente evento
      // de la lista de eventos, para un objeto particular
      ta = ptr->GetTiempoProximoEvento();

      // Evaluar los posibles estados de cada
      // objeto de simulación
      if(ptr->Estado() == DISPONIBLE && ta == 100.0*ENDSIM)
      {
        la = 100.0 * ENDSIM;
      }
      else if(ptr->Estado() == DISPONIBLE)
      {
        la = ta + st;
```

```
    }
    else if(ptr->NumTareas() > 0)
    {
      la = lst + st;
    }
    else if(ta == 100.0 * ENDSIM)
    {
      la = ta;
    }

    else if(ta < lst)
    {
      la = lst + st;
    }
    else
    {
      la = ta + st;
    }

    // Obtener el menor valor para TWindow
    if(TWindow > la)
    {
      TWindow = la;
    }
  }
}

// Enviar el posible valor de TWindow
// al resto de los procesadores.
for(int cont = 0; cont < NUM_PROC; cont++)
{
  if(cont != pid)
  {
    com.StoreMsg("\0", TWindow, -1, -1, NULL);
  }
}
```

En este código se obtiene el menor de los tiempos de salida de los próximos eventos que requieran atención, en cada uno de los objetos de simulación. Esto se hace a través del primer ciclo, que recorre el mapa de objetos usando un iterador. Algunas de las variables utilizadas para encontrar el mínimo tuvieron que ser inicializadas en un valor suficientemente grande, como es el caso de *TWindow*. En seguida se tiene un segundo ciclo, que se encarga de enviar el valor encontrado al resto de los procesadores.

- Escritura de código dentro de varias clases para poder facilitar los cálculos necesarios para la implementación de YAWNS, respetando el modelo orientado a objetos. Por ejemplo, para

obtener el tiempo de servicio calculado para un evento próximo en un objeto de simulación, se escribió el siguiente método en la clase *Dispositivo*:

```
// Retorna el próximo tiempo de servicio
// calculado según el protocolo YAWNS
double ServiceTime(){ return st; }
```

3 Algoritmo de Ventana Fija

Este es un algoritmo simple de simulación en paralelo en donde la ventana de simulación es incrementada en un valor constante e igual a *LookAhead*. Una vez alcanzado este límite se debe realizar una sincronización, dando a lugar el envío y recepción de datos entre procesadores.

Debido a su simplicidad, este algoritmo requiere de sólo una sincronización durante una ejecución del ciclo principal, a diferencia del protocolo YAWNS que necesita dos.

4 Evaluación Experimental

Las simulaciones se ejecutaron en equipos *Pentium III* a 700 MHz, con 128 MB de RAM, conectados en red y ejecutando *Linux RedHat 7.2*. La versión del kernel de Linux fue la 2.4.7-10. Todas las simulaciones en paralelo se realizaron utilizando cuatro de estos equipos.

La versión de las librerías PUB y MPI utilizadas fueron la 8.0 y 1.2.3, respectivamente. Ambas librerías exigen definir un programa que permita ejecutar comandos en forma remota, así que por simplicidad se eligió *rsh* debido a que ya estaba configurado de tal forma que es posible ejecutar comandos remotos sin necesidad de ingresar contraseñas.

El tiempo medido en las pruebas corresponde al tiempo de procesador utilizado por el programa; éste fue medido a través de la función *clock()*, que mide el tiempo de CPU en milisegundos. En algunos casos fue necesario valerse de la función *time()* que mide el tiempo en segundos, ya que el tiempo medido con *clock()* se reinicia cada media hora. Al menos esto sucede en la implementación de *clock()* en Linux.

Todas las pruebas se hicieron aislando los equipos del tráfico de la red del laboratorio en que se hicieron las pruebas, y evitando interferir en el proceso de recolección de datos. Este proceso se automatizó a través de un *script* en lenguaje *shell*. Este *script*, a través de un ciclo, crea un archivo de configuración del modelo de simulación (cuyo tamaño varía con cada ciclo) utilizando el programa en *perl* mencionado, y ejecuta cada simulación, utilizando los programas que correspondan y redireccionando las salidas a un archivo. El intérprete utilizado para este *script* fue *bash*.

Por último, antes de exponer los resultados obtenidos, cabe señalar que los programas fueron compilados utilizando los comandos ofrecidos por las librerías PUB y MPI. Éstos no son otra cosa que *scripts* que llaman a un compilador con varios argumentos. Para que se realice una correcta evaluación se comprobó que ambos *scripts* utilicen el mismo compilador -*gcc*- y que ambos apliquen el mismo nivel de optimización a través de la opción -O3.

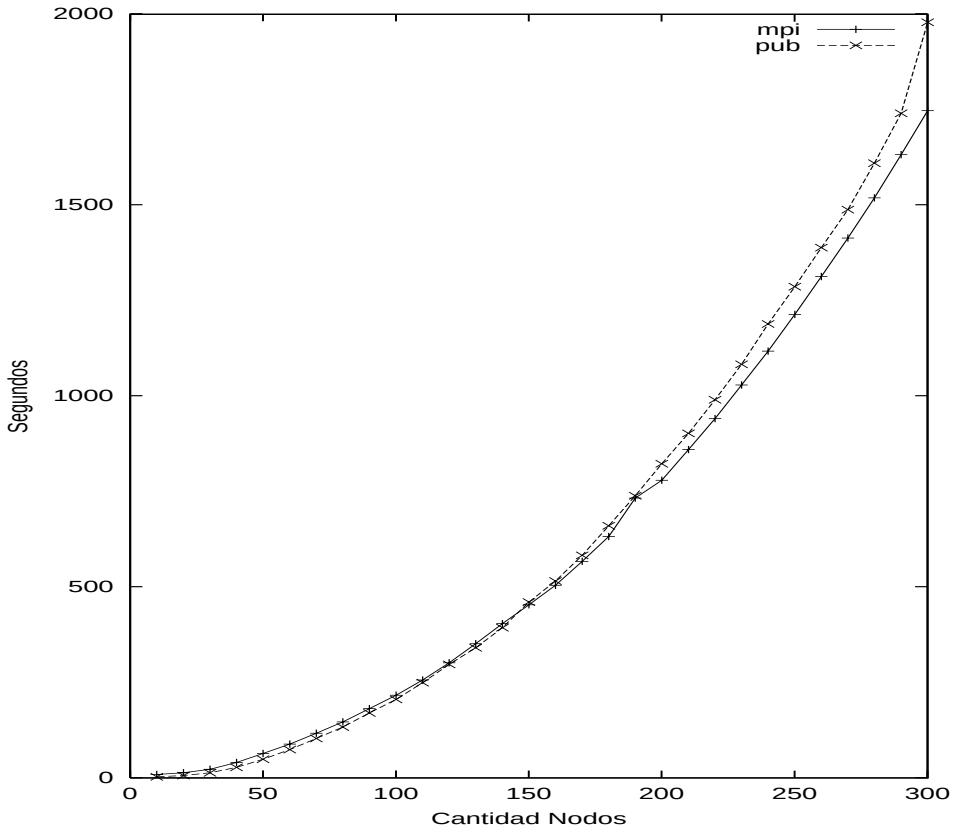

Figure 1: Tiempos de ejecución de sistemas toroidales, utilizando distintas librerías.

4.1 Comparación entre MPI y PUB

Para obtener las muestras de esta comparación se utilizó el algoritmo de Ventana Fija, con un *LookAhead* de 1.0, además del resto de las condiciones descritas al inicio de este capítulo.

La figura 1 muestra que el desempeño de cada librería varía dependiendo del tamaño del sistema a simular. En el caso del toroide, se observa que la librería PUB posee un manejo de comunicaciones levemente más rápido que MPI para sistemas de tamaño menor que 150x150. Pero para sistemas mayores MPI obtiene una modesta ventaja frente a PUB. Podemos decir que ambas librerias entregan un desempeño similar.

4.2 Comparación entre simulación secuencial y simulación paralela

A continuación se implementó un simulador secuencial -es decir, que se ejecute en un solo procesador- a partir del simulador utilizado en la comparación anterior. Para esto la función principal sufrió cambios importantes, ya que no tiene sentido hablar de comunicaciones entre procesadores si el código se debe ejecutar en una sola máquina.

En el gráfico de la figura 2 se puede observar que para simulaciones pequeñas -esto es, para

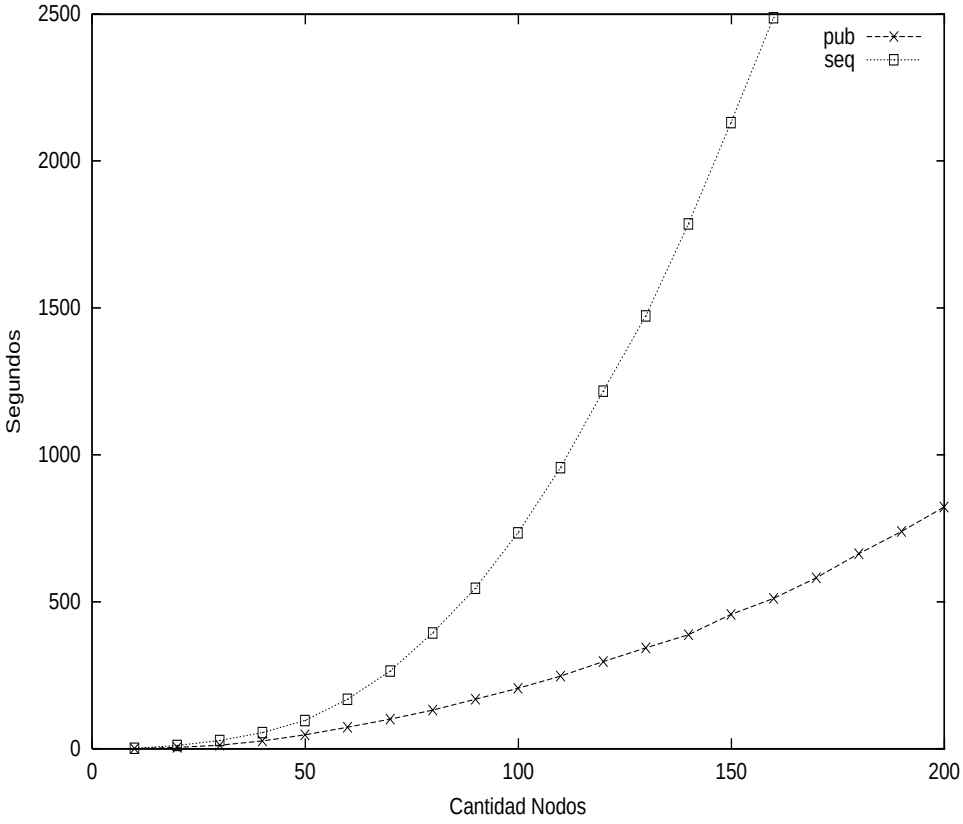

Figure 2: Comparación simulación secuencial y paralela.

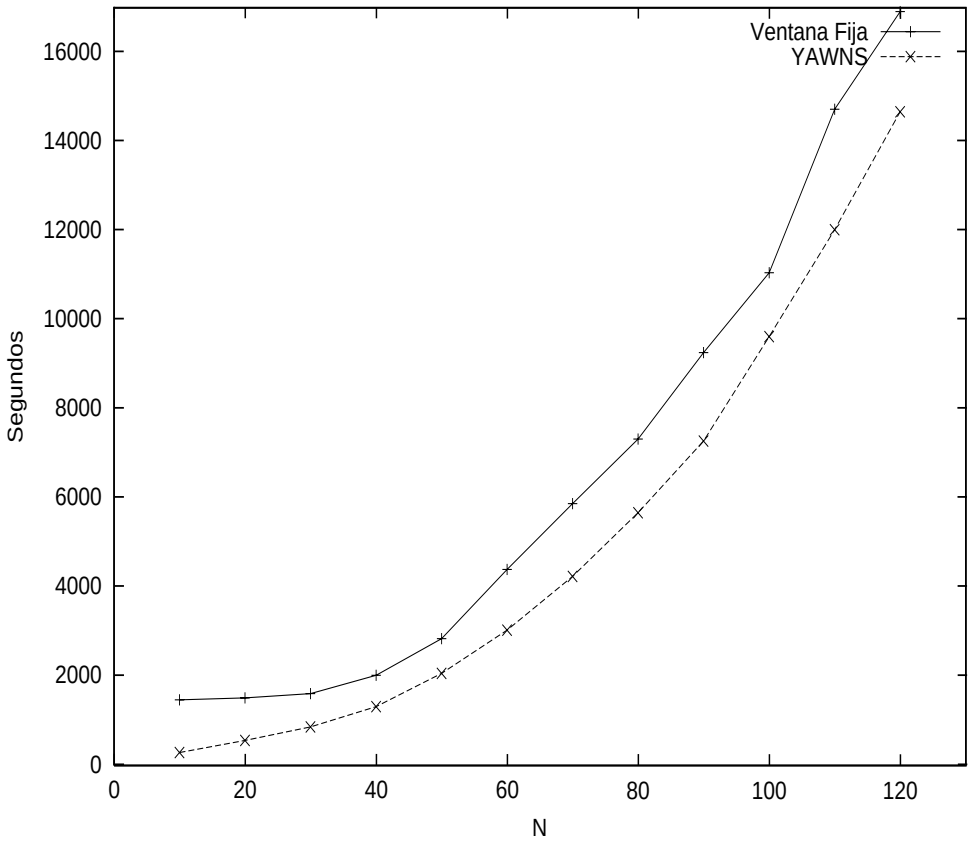

Figure 3: Tiempos de ejecución para toroides de NxN, Lk = 0.01.

toroides NxN con $N <$ 10- el algoritmo secuencial obtiene ventaja frente a los algoritmos de simulación paralela, pero que para sistemas mayores los algoritmos paralelos mantienen un mejor desempeño frente a una simulación secuencial.

4.3 Comparación entre Protocolo de Ventana Fija y Protocolo YAWNS

Para comparar el desempeño de ambos protocolos se utilizaron distintos valores para la constante *LookAhead*. El fin de la simulación se estableció en 20000, y el promedio de los tiempos de servicio para cada objeto de simulación se ajustó a 1.0.

Para un *LookAhead* igual a 0.01, la simulación se desarrolla más rápido utilizando el protocolo YAWNS que usando Ventana Fija. La figura 3 grafica los resultados obtenidos.

Si se aumenta el valor de *LookAhead* a 0.05, tenemos que el desempeño de la simulación es mejor utilizando el protocolo de Ventana Fija que con el protocolo YAWNS, como se observa en la figura 4.

Finalmente se aumentó el *LookAhead* a 1.0, y se observó que el algoritmo de Ventana Fija obtiene una gran ventaja sobre YAWNS (ver figura 5). Sin embargo, se debe tener en cuenta que

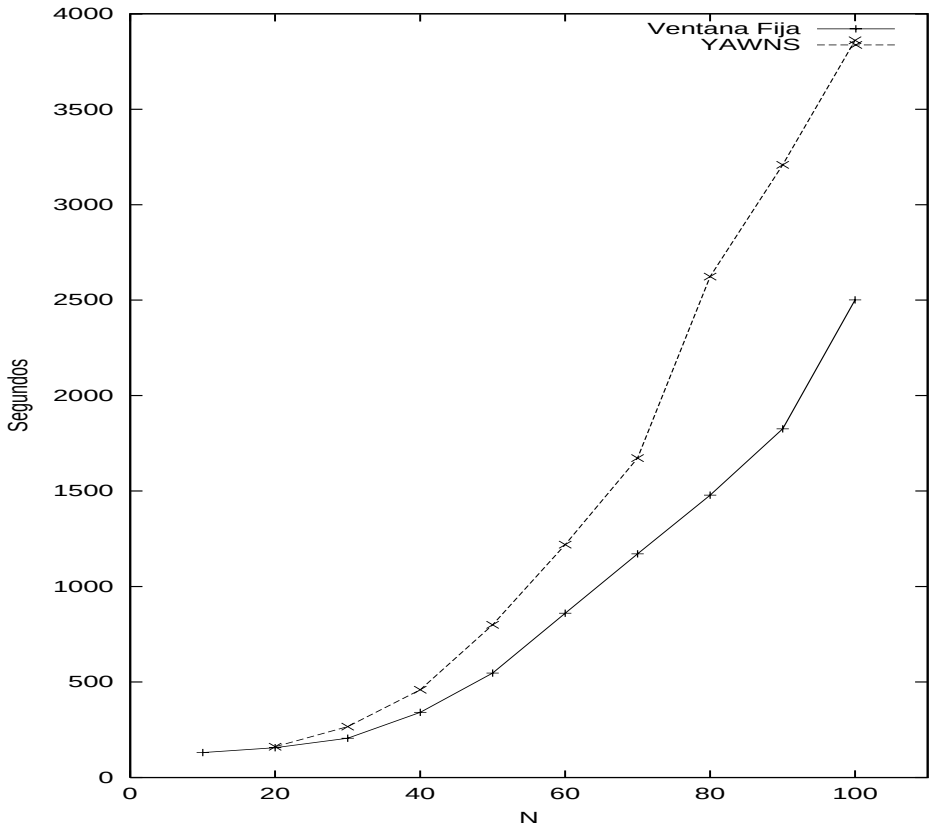

Figure 4: Tiempos de ejecución para toroides de NxN, Lk = 0.05.

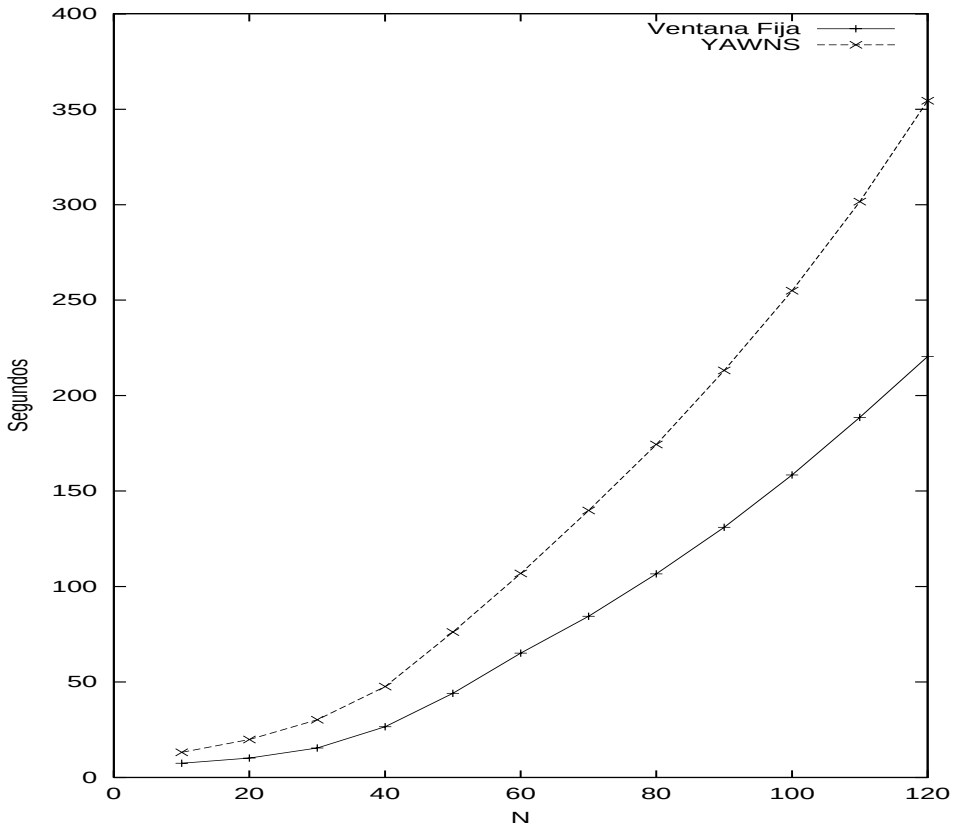

Figure 5: Tiempos de ejecución para toroides de NxN, Lk = 1.0

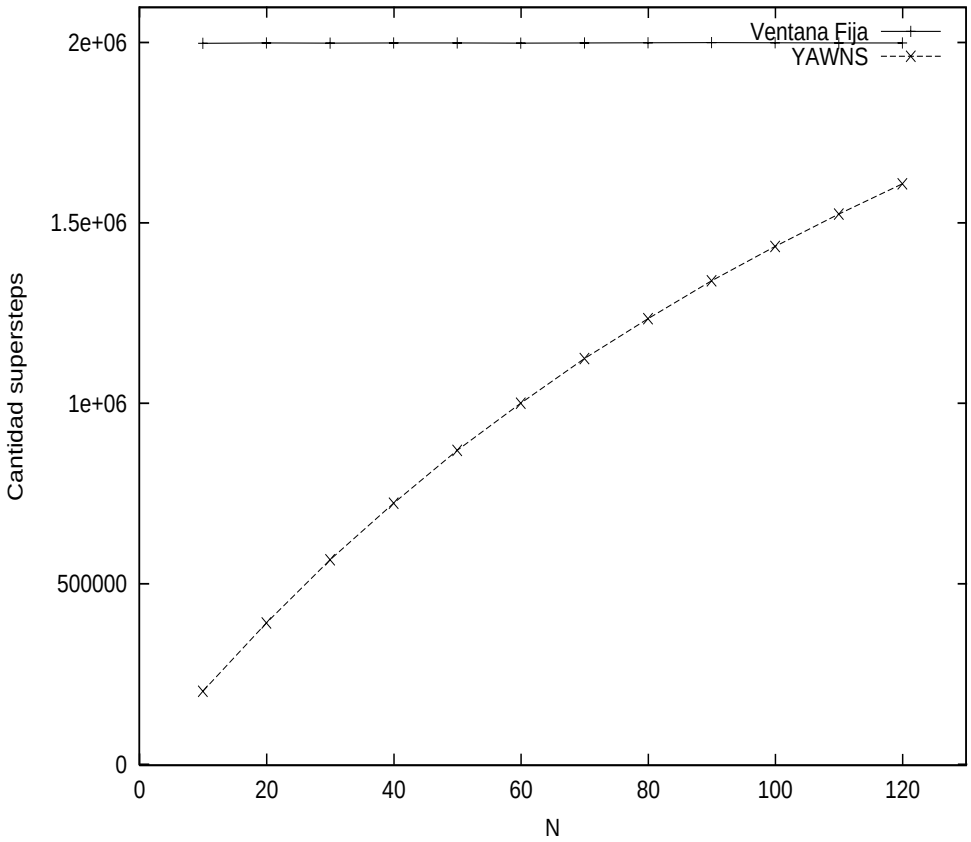

Figure 6: Cantidad de Supersteps realizados durante una simulación, según N (Lk = 0.01).

sólo en casos muy particulares se utilizan valores altos en *LookAhead*

En la figura 6 se observa un gráfico que muestra la cantidad de supersteps realizados durante una simulación para ambos algoritmos, con *LookAhead* 0.01. Es importante notar que el protocolo YAWNS exige que existan dos supersteps en cada ciclo de simulación, mientras que en el algoritmo de Ventana Fija sólo es necesario un superstep. Un gráfico similar se puede apreciar en la figura 7, pero esta vez el valor de *LookAhead* fue de 1.0. A partir de estos gráficos se puede justificar los resultados obtenidos de ambos algoritmos, ya que la completación de un superstep implica la ejecución de cierta cantidad de código más o menos constante, correspondiente al envío y recepción de mensajes entre procesadores. Por lo tanto, el protocolo YAWNS debe ejecutar al menos el doble de código en cada ciclo, en comparación con el protocolo de Ventana Fija, así que necesitaría ejecutar una cantidad de ciclos menor que el algoritmo Ventana Fija para poder obtener ventaja.

5 Conclusiones

Durante el desarrollo de este trabajo se demostró que se puede obtener mejor desempeño en simulaciones computacionales aplicando técnicas de paralelismo. Sin embargo existen ciertos parámetros

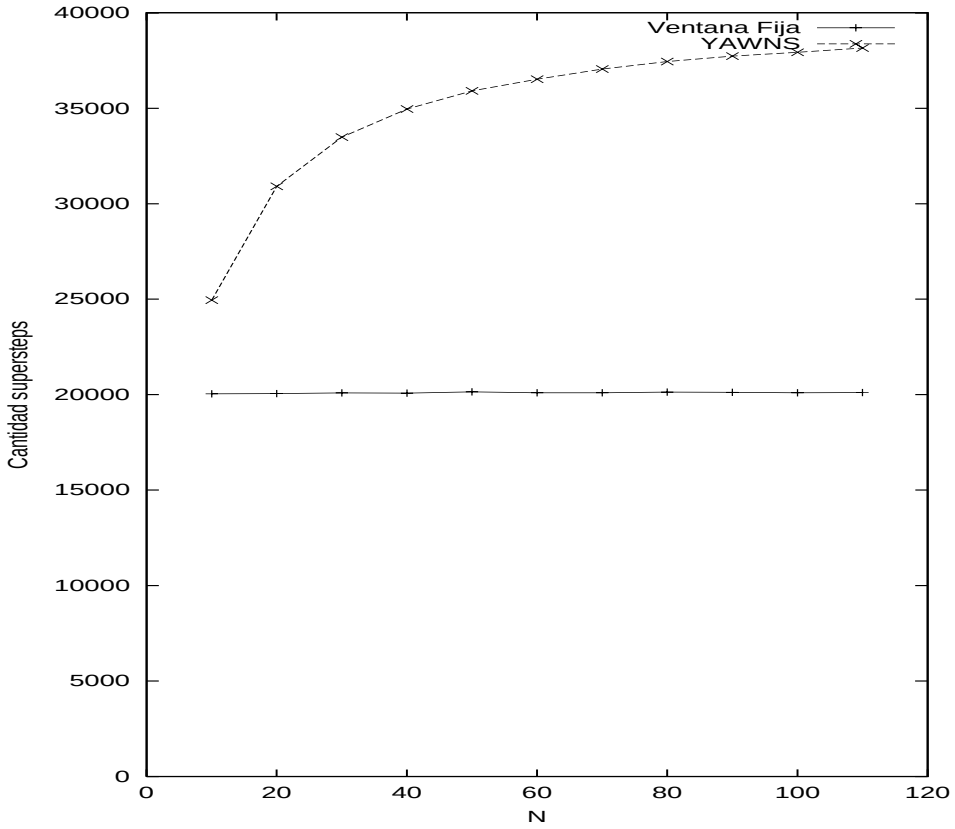

Figure 7: Cantidad de Supersteps realizados durante una simulación, según N (Lk = 1.0).

que se deben tener en cuenta al evaluar la aplicación de algoritmos paralelos en un sistema:

1. Para simulaciones pequeñas, el uso de algoritmos paralelos puede igualar o empeorar el desempeño de un sistema en comparación con un algoritmo secuencial.

2. Las librerías PUB y MPI alcanzan un desempeño similar en este tipo de aplicaciones de computación paralela.

3. En cuanto a los protocolos de simulación en paralelo evaluados en este trabajo, el valor de la constante *LookAhead* es un factor importante para decidir su aplicación sobre un sistema. En la mayoría de los casos el valor del *LookAhead* es bastante pequeño, por lo tanto conviene utilizar el protocolo YAWNS. En cambio, cuando se utilizan valores grandes de *LookAhead*, conviene utilizar un algoritmo simple, como el de Ventana Fija propuesto durante el desarrollo de este trabajo.

   Este comportamiento es justificable si tenemos en cuenta que YAWNS requiere de dos supersteps por cada ciclo de simulación, al contrario del algoritmo de Ventana Fija que sólo requiere de un superstep.

6 Bibliografía

- Cornell Theory Center, FAQ: Derived Datatypes, *http://www.pdc.kth.se/ training/FAQ/derived.html*, 1996.

- Gropp William, Tutorial on MPI: The Message-Passing Interface, *http://www-unix.mcs.anl.gov/ mpi/tutorial/gropp/talk.html*, 2003.

- Jha, V., *Simultaneous Events and Lookahead in Simulations Protocols*, Computer Science Department, University of California, Los Angeles, CA 90024.

- Joint Institute for Computational Science, Introduction to Message Passing Interface, *http://www.jics.utk.edu/SP_LECT/sp_mpi/index.htm*, 1997.

- Marín, M. *Discrete-Event Simulation On The Bulk-Synchronous Parallel Model*, Computing Laboratory, Programming Research Group, Oxford University, Dec. 1998.

- MPICH - A portable MPI implementation (Página principal), *http://www-unix.mcs.anl.gov/ mpi/mpich/*, 2003.

- Nicol, D., Parallel JTeD: The YAWNS Protocol, *http://www.cs.dartmouth.edu/˜ nicol/S3/yawns.html*, 2003.

- Page, Hernest H, *Zero Lookahead in a Distributed Time-Stepped Simulation*, The MITRE Corporation, 1820 Dolley Madison Blvd. McLean, VA 22102.

- PUB-Library (Página principal), *http://www.uni-paderborn.de/˜ bsp/*, 2003.

- Snir, M., *MPI: The Complete Reference*, The MIT Press, Cambridge, Massachusetts, London, England, 1996

- Wang, Jain J., *The Impact of Lookahead on Conservative Simulation*, Departament of Computing Science, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061-0106, February 24, 1995.