Computación Paralela

Proyecto
Paralelización de un simulador de incendios

Nombres: Emmanuel David
Apellido: Bello
Legajo: 31897
Año: 2010

INTRODUCCIÓN

En el presente trabajo se explica como realizar paralelizar un simulador de incendios.
La estrategia que se usó, fue computación mediante Master-Worker.
La paralelización consistió en probar todas las combinaciones posibles de los parámetros usados durante la simulación (modelo, velocidad del viento, etc.).
Lo que se hizo fue combinar la computación paralelo mediante el paso de mensajes con la computación miltihilo.

METODOLOGÍA DE PARALELIZACIÓN CON MPI

Como se dijo anteriormente se usó la metodología de Master-Worker. El Master se encargó de encontrar todas las combinaciones posibles de parámetros. Además carga los datos referentes a las matrices usadas en la simulación. Para ello lee dos archivos, uno con los datos de las matrices y otro con los parámetros (límite inferior, límite superior, salto). Además se calcula la cantidad de combinaciones de parámetros que le corresponderían a cada worker (el cual se encargará de realizar la simulación). En caso de que el número de combinaciones posibles no sea divisible entre el número de workers, el último worker recibirá menos cantidad de combinaciones de parámetros que el resto.
El Master le envía a cada worker un conjunto de combinaciones de parámetros, en cada instante le envía la misma cantidad de combinaciones a cada worker (a excepción del último, en caso de que no sea divisible la cantidad de combinaciones que le corresponden a cada worker con la cantidad de combinaciones por mensaje).
Una vez que un worker recibe un conjunto de combinaciones de parámetros, se encarga de realizar una simulación por cada combinación de parámetros. Por cada simulación que termina un worker, éste le envia al Master un paquete con dos mapas (el mapa de incencidios binario y el de alturas de llamas). El master una vez que recibe el dicho paquete se encarga de extraer cada mapa, y sumar cada uno de los elementos de cada mapa, al respectivo mapa final de valores.
El master mantiene dos mapas, uno de mapas de incendios y otro de alturas de llamas. Por cada mapa binario de incendios (esto es, un uno en cada celda en la que va a haber incendio, y un cero en caso contrario) que se recibe de un worker, el master suma cada valor del mapa al mapa de incendios final, que luego de recibir todos los mapas de los workers, se encarga de sacar un promedio por cada celda, de modo que en cada celda del mapa de incendios final, se encuentra la probabilidad de incendio en dicha celda. Lo mismo se hace para el caso de los mapas de alturas de llamas, con la diferencia que cada worker genera un mapa de alturas normal.
Una vez hecho esto, el master imprime los dos mapas finales en dos archivos diferentes.

USO DE HILOS EN COMBINACIÓN CON MPI

Para mejorar el rendimiento del trabajo entre las máquinas, se decidió usar tres hilos en cada worker y tres hilos en el master. De modo que en cada caso, cada hilo cumple una función diferente:
Hilos en el master:

- Hilo de envio de trabajos: este se encarga de obtener e informar a los workers la información referente a los tamaños de las matrices, calcula el todas las combinaciones posibles de parámetros y se las envia a cada worker de la forma antes descripta.

- Hilo de recepción de resultados: este hilo se encarga de recibir los mapas binarios de incendios de cada worker y los mapas de alturas de llamas que envía cada worker al master.
- Hilo de suma de resultados: este hilo suma cada uno de los mapas resultantes obtenidos a los respectivos mapas finales. Una vez hecho esto se encarga de imprimir ambos mapas finales en dos archivos de salida.

Hilos en cada worker:

- Hilo de recepción de trabajos: este se encarga de recibir los paquetes con conjuntos de combinaciones de parámtros enviados desde el master. Además, previamente, recibe el tamaño de las matrices y el número de combinaciones de que debe tratar.
- Hilo de simulación: este se encarga de realizar una simulación por cada combinación de parámetros recibida.
- Hilo de envio de resultados: este hilo se encarga de empaquetar el mapa binario de incendios y el mapa de altura de llamas obtenido en cada simulación, en un solo paquete y se lo envia al master.

PROBLEMAS DE PRODUCTOR-CONSUMIDOR

Estos problemas se deben a que hay casos en que se accede a un buffer compartido para leer y escribir datos.

Los casos en los que se presentan estos problemas son:

- Comunicación entre el hilo de simulación y el de recepción de trabajos: esto se debe a que ambos hilos escriben y leen de un mismo buffer, el cual contiene las combinaciones de parámetros que el hilo simulador tiene que usar. Debido a que ambos hilos pueden estar accediendo al mismo tiempo al buffer de entrada de combinaciones de parámetros, se usan mutex y variables de condición. Los mutexes para lograr la exclusión mutua al buffer. Las variables de condición se usan para evitar que un hilo bloquee un mutex cuando no se cumple una condición necesaria para trabajar.
- Comunicación entre el hilo de simulación y el de envío de resultados: es un caso similar al anterior. En este caso se accede a un buffer en el que se van almacenando los mapas a enviarse al master.
- Comunicación entre el hilo de recepción de resultados y el de suma de resultados: en este caso se accede mutuamente a un mismo buffer en el que se almacenan los mapas recibidos de los workers.

Como ejemplos se muestra el código del hilo de suma de resultados y el de recepción de resultados:

```
void * recibirResultados(){
   MPI_Status status;
   int indexBuffer=0,np;
   double comb=0;
   int flag,position;
   char * inBuf;//buffer de entrada
   int tam_buf,tamMap;//tamaño del buffer de entrada
   tam_buf = 2*paramMatrix->Rows*paramMatrix->Cols*sizeof(double);
   tamMap = paramMatrix->Rows*paramMatrix->Cols;
   inBuf = malloc(tam_buf);
   /*se realiza un bucle hasta que se hayan obtenidos los resultados de todas las combinaciones*/
```

```
   while(comb < combinaciones){
      /*por cada worker se verifica si ya tiene una respuesta para el master*/
      for(np=1;np < nproc;np++){
         MPI_Iprobe(np,RESULT_TAG, MPI_COMM_WORLD,&flag, &status);//se verifica si hay
nuevos datos enviados por el worker
         if(flag){
            position = 0;
            MPI_Recv(inBuf,tam_buf,MPI_PACKED,np,RESULT_TAG,MPI_COMM_WORLD,&status);
            //mapa de ignición
            pthread_mutex_lock(&mutexBufferIgnMap);
            while(condicion_buffer_ignMap == combinacionesPorWorker){//mientras el buffer esté
lleno
               pthread_cond_wait(&condpBufferIgnMap,&mutexBufferIgnMap);
            }//fin while
            printf("mapass    %d\n", tamMap);
            MPI_Unpack(inBuf,tam_buf,&position,buffer_ignMap[indexBuffer],tamMap,MPI_DOUBLE
,MPI_COMM_WORLD);

            condicion_buffer_ignMap++;//se indica que se produjo un elemento más en el buffer
            pthread_cond_signal(&condcBufferIgnMap);
            pthread_mutex_unlock(&mutexBufferIgnMap);

            //mapa de altura de llamas
            pthread_mutex_lock(&mutexBufferFlMap);
            while(condicion_buffer_flMap == combinacionesPorWorker){//mientras el buffer esté lleno
               pthread_cond_wait(&condpBufferFlMap,&mutexBufferFlMap);
            }//fin while
            MPI_Unpack(inBuf,tam_buf,&position,buffer_flMap[indexBuffer],tamMap,MPI_DOUBLE,
MPI_COMM_WORLD);
            condicion_buffer_flMap++;//se invierte la condicion
            pthread_cond_signal(&condcBufferFlMap);
            pthread_mutex_unlock(&mutexBufferFlMap);
            indexBuffer = (indexBuffer +1)%((int)combinacionesPorWorker);
            comb++;
         }//fin if
      }//fin for
   }//fin while

pthread_exit(0);
}//fin de recibirResultados

void * sumarResultados(void * filesNamesReceiver){

   char * fileName_outIncencio;
   char * fileName_outLlamas;
   fileName_outIncencio = ((ArchNames*)filesNamesReceiver)->incendiosFile;
   fileName_outLlamas = ((ArchNames*)filesNamesReceiver)->llamasFile;
```

```
double * ignMapFinal;//mapa de ignición final
double * flMapFinal;//mapa de altura de llamas final
int cantCeldas,indexBuffer = 0;//cantidad de celdas en cada mapa
double comb = 0;
int i;
cantCeldas = paramMatrix->Rows * paramMatrix->Cols;
//los mapas se inicializan todos en cero
ignMapFinal = (double*)calloc(cantCeldas,sizeof(double));
flMapFinal = (double*)calloc(cantCeldas,sizeof(double));
while(comb < combinaciones){
   //por el mapa de ignicion
   pthread_mutex_lock(&mutexBufferIgnMap);/*obtiene acceso excluiso al buffer*/
   while(condicion_buffer_ignMap == 0){//mientras no haya nuevos elementos en el buffer
      pthread_cond_wait(&condcBufferIgnMap,&mutexBufferIgnMap);
   }//fin while
   /*se suman todos los valores en el mapa final ya la ubicación actual
    en el buffer va a ser luego reutilizada*/
   for(i = 0; i < cantCeldas; i++){
      ignMapFinal[i] += buffer_ignMap[indexBuffer][i];
   }//fin for
   condicion_buffer_ignMap--;//se indica que se ha consumido un valor
   pthread_cond_signal(&condpBufferIgnMap);
   pthread_mutex_unlock(&mutexBufferIgnMap);/*libera el acceso al buffer*/

   //por el mapa de altura de llamas
   pthread_mutex_lock(&mutexBufferFlMap);/*obtiene acceso excluiso al buffer*/
   while(condicion_buffer_flMap == 0){//mientras no haya nuevos elementos en el buffer
      pthread_cond_wait(&condcBufferFlMap,&mutexBufferFlMap);
   }//fin while
   for(i = 0; i < cantCeldas; i++){
      flMapFinal[i] += buffer_flMap[indexBuffer][i];
   }//fin for
   condicion_buffer_flMap--;//se indica que se ha consumido un valor
   pthread_cond_signal(&condpBufferFlMap);
   pthread_mutex_unlock(&mutexBufferFlMap);/*libera el acceso al buffer*/
   indexBuffer = (indexBuffer +1)%((int)combinacionesPorWorker);
   comb++;
}//fin while
/*por cada mapa se calcula un promedio por cada celda*/
for(i = 0 ; i < cantCeldas;i++){
   flMapFinal[i] /= combinaciones;
   ignMapFinal[i] /= combinaciones;
}//fin for
/*se imprimen los respectivos mapas en los archivos*/
PrintMap(ignMapFinal,fileName_outIncencio);
PrintMap(flMapFinal,fileName_outLlamas);
free(ignMapFinal);
```

```
   free(flMapFinal);
   pthread_exit(0);
}//fin de sumarResultados
```

A continuación se muestran el código master:

```
void codigoMaster(char * fileName_maps, char * fileName_param, char * fileName_outIncencio,
char * fileName_outLlamas){
/*El master se encarga de generar los valores y decidir las combinaciones de parámetros que tiene que
usar cada
 worker para ejecutar la simulación. Además tiene que recibir los resultados retornados por los
workers.
 * Por último tiene que sumar y promediar los resultados retornados por los workers
 Para ello, se usan dos hilos: uno de envio tareas y otro de recepción de resultados*/
      pthread_t * sender;
      pthread_t * receiver;
      pthread_t * sumador;
      void * return_value;
      ArchNames *fileNamesReceiver;
      fileNamesReceiver = (ArchNames*)malloc(sizeof(ArchNames));
      int estado,i,j;//estado devuelto durante la creacion de un hilo

      /*se dan valores a las variables globales usadas por los hilos del master*/
      inicializarVarGl(fileName_maps,fileName_param);

      if((nroDeWorkers) > combinaciones){
         i = 0;
         /*se les avisa a los workers sobrantes que el número de combinaciones que le corresponden es
cero,
          de modo que terminan*/
         for(j = combinaciones +1; j<= nroDeWorkers;j++ ){

            MPI_Send(&i,1, MPI_INT,j,COMBPORWORKER_TAG,MPI_COMM_WORLD);
         }//fin for
         nproc = combinaciones +1;
         nroDeWorkers = nproc-1;
      }//fin if
      /*se inicializan variables de condición y los mutex*/
      pthread_mutex_init(&mutexBufferIgnMap,NULL);
      pthread_mutex_init(&mutexBufferFlMap,NULL);
      pthread_cond_init(&condcBufferIgnMap,0);
      pthread_cond_init(&condpBufferIgnMap,0);
```

```
	pthread_cond_init(&condcBufferFlMap,0);
	pthread_cond_init(&condpBufferFlMap,0);
	sender = (pthread_t *)malloc(sizeof(pthread_t));
	receiver = (pthread_t *)malloc(sizeof(pthread_t));
	sumador = (pthread_t *)malloc(sizeof(pthread_t));
	estado = pthread_create(sender,NULL,enviarTrabajos,NULL);
	if(estado){
		printf("error al crear hilo sender\n");
		exit(-1);
	}//fin if
	fileNamesReceiver->incendiosFile = fileName_outIncencio;
	fileNamesReceiver->llamasFile = fileName_outLlamas;
	/*se asigna espacio a los bufferes usados por los hilos
	 receiver y sumador*/
	buffer_ignMap =
		(double **)malloc(paramMatrix->Rows * paramMatrix->Cols
*combinacionesPorWorker*sizeof(double));
	buffer_flMap =
		(double **)malloc(paramMatrix->Rows * paramMatrix-
>Cols*combinacionesPorWorker*sizeof(double));
	estado = pthread_create(receiver,NULL,recibirResultados,NULL);
	if(estado){
		printf("error al crear hilo receiver\n");
		exit(-1);
	}//fin if
	estado = pthread_create(sumador,NULL,sumarResultados,(void *)fileNamesReceiver);
	if(estado){
		printf("error al crear hilo sumador\n");
		exit(-1);
	}//fin if
	/*se espera a que terminen de ejecutarse ambos hilos*/
	if(pthread_join(*sender, &return_value)) printf("error al terminar el hilo sender\n");
	if(pthread_join(*receiver, &return_value)) printf("error al terminar el hilo receiver\n");
	if(pthread_join(*sumador, &return_value)) printf("error al terminar el hilo sumador\n");
	//se destruyen las variables de condicion
	pthread_cond_destroy(&condcBufferIgnMap);
	pthread_cond_destroy(&condpBufferIgnMap);
	pthread_cond_destroy(&condcBufferFlMap);
	pthread_cond_destroy(&condpBufferFlMap);
	pthread_mutex_destroy(&mutexBufferIgnMap);
	pthread_mutex_destroy(&mutexBufferFlMap);
	free(sender);
	free(receiver);
	free(sumador);
	free(paramMatrix);
	/*se libera el espacio usados por los mapas*/
	for(i = 0 ; i < combinacionesPorWorker;i++){
```

```
            free(buffer_ignMap[i]);
            free(buffer_flMap[i]);
        }//fin for
        free(buffer_ignMap);
        free(buffer_flMap);
        MPI_Finalize();
}//fin codigoMaster
```

A continuación se muestra el código ejecutado por cada worker:

```
void codigoWorker(){
    MPI_Status status;
    /*El worker se encarga de realizar la simulación por cada combinación de parámetros recibida
     Una vez terminada la recepción de un mensaje, se dedica a procesar una simulación por cada combinación
     recibida en el mensaje. Además una vez efectuada la simulación debe generar dos matrices resultados,
     una correspondiente al mapa de incendios, el cual es un mapa binario que contiene un uno en cada celda
     donde se sabe que va a haber incendio, es decir, que el tiempo que tardará el fuego en llegar a dicha zona
     es distinta de infinito, y la otra correspondiente a la altura de las llamas en cada celda*/
    /*Para realizar todos su trabajo, el nodo worker tiene:
     * un hilo receptor de mensajes
     * un hilo que se encarga de realizar las simulaciones
     * un hilo que se encarga de enviar todos los resultados al master
     */
    pthread_t * receiver;//hilo receptor de mensajes
    pthread_t * simulator;//hilo simulador
    pthread_t * sender;//hilo enviador de mensajes
    int tam_in ;
    char * buf_in;
    void * return_value;
    int estado, position = 0;//estado devuelto durante la creacion de un hilo
        /*primero se recibe la cantidad de combinaciones con que tiene que tratar este worker*/
    MPI_Recv(&combinacionesPorWorker,1,MPI_INT,NODO_MASTER,COMBPORWORKER_TAG,MPI_COMM_WORLD,&status);
    if(combinacionesPorWorker != 0){//si es cero se termina inmediatamente la ejecución de este worker
        /*se recibe los parámetros acerca de las matrices a usar*/
        tam_in = sizeof(ParamMatrix);
        buf_in = (char*)malloc(tam_in);
```

```
	paramMatrix = (ParamMatrix*)malloc(tam_in);
	MPI_Recv(buf_in,tam_in,MPI_PACKED,NODO_MASTER,MATRIZ_TAG,MPI_COMM_WORLD
,&status);

	MPI_Unpack(buf_in,tam_in,&position,&(paramMatrix-
>Rows),1,MPI_INT,MPI_COMM_WORLD);
	MPI_Unpack(buf_in,tam_in,&position,&(paramMatrix-
>Cols),1,MPI_INT,MPI_COMM_WORLD);
	MPI_Unpack(buf_in,tam_in,&position,&(paramMatrix-
>CellHt),1,MPI_DOUBLE,MPI_COMM_WORLD);
	MPI_Unpack(buf_in,tam_in,&position,&(paramMatrix-
>CellWd),1,MPI_DOUBLE,MPI_COMM_WORLD);
	sender = (pthread_t *)malloc(sizeof(pthread_t));
	receiver = (pthread_t *)malloc(sizeof(pthread_t));
	simulator = (pthread_t *)malloc(sizeof(pthread_t));
	/*se asigna espacio a los bufferes usados por los hilos*/
	bufferDeEntrada =
(ParamValues*)malloc(combinacionesPorWorker*sizeof(ParamValues));//buffer de entrada para
simulator
	/*bufferes de salida para simulator*/
	buffer_flMap = (double **)malloc(combinacionesPorWorker*sizeof(double*));
	buffer_ignMap = (double **)malloc(combinacionesPorWorker*sizeof(double*));

	/*se inicializan variables de condición y los mutex*/
	pthread_mutex_init(&mutex_in,NULL);
	pthread_mutex_init(&mutex_out_ignMap,NULL);
	pthread_mutex_init(&mutex_out_flMap,NULL);
	pthread_cond_init(&condc_in,0);
	pthread_cond_init(&condc_out_ignMap,0);
	pthread_cond_init(&condc_out_flMap,0);
	estado = pthread_create(receiver,NULL,recibirTrabajos,NULL);
	if(estado){
		printf("error al crear hilo receiver\n");
		exit(-1);
	}//fin if

	estado = pthread_create(simulator,NULL,simular,NULL);
	if(estado){
		printf("error al crear hilo simulator\n");
		exit(-1);
	}//fin if

	estado = pthread_create(sender,NULL,enviarResultados,NULL);
	if(estado){
		printf("error al crear hilo sender\n");
		exit(-1);
	}//fin if
```

```
    /*se espera a que terminen de ejecutarse los tres hilos*/
    if(pthread_join(*receiver, &return_value)) printf("error al terminar el hilo receiver\n");
    if(pthread_join(*simulator, &return_value)) printf("error al terminar el hilo simulator\n");
    if(pthread_join(*sender, &return_value)) printf("error al terminar el hilo sender\n");
    //se destruyen las variables de condicion
    pthread_cond_destroy(&condc_in);
    pthread_cond_destroy(&condc_out_ignMap);
    pthread_cond_destroy(&condc_out_flMap);
    pthread_mutex_destroy(&mutex_in);
    pthread_mutex_destroy(&mutex_out_ignMap);
    pthread_mutex_destroy(&mutex_out_flMap);
    free(receiver);
    free(simulator);
    free(sender);
    free(bufferDeEntrada);
    free(buffer_ignMap);
    free(buffer_flMap);
  }//fin if
  MPI_Finalize();
}//fin codigoWorker
```

JUSTIFICACIÓN

Se decidió emplear hilos en cada nodo con el propósito de solapar el cómputo y la comunicación. De este modo se evitan los tiempos osciosos en que se incurre durante el paso de mensajes entre máquinas. Las situaciones donde los hilos mejoran significativamente el rendimiento son:

- Cuando el master está enviando los mensajes con combinaciones de parámetros a cada worker, cada worker tiene que aceptar inmediatamente este mensaje para que el master puede continuar generando los siguientes mensajes para los siguientes workers. Con el uso de un hilo receptor de trabajos en cada worker, se puede solapar la simulación que está realizando un worker en ese momento con la recepción del mensaje.
- Un worker podría no continuar con la siguiente simulación, debido a que no puede enviarle al master un paquete con los resultados obtenidos de la simulación anterior, ya el master se encuentra recibiendo los mensajes de otro worker, por ejemplo. Con el uso de un hilo emisor de mensajes y otro simulador, en cada worker, se puede comenzar con la siguiente simulación antes de que el master acepte el resultado obtenido de la simulación anterior. Además en el master el hilo receptor de resultados no tiene que esperar a que el worker n le envie sus resultados para poder recibir los resultados del worker n+1, ya que este hilo realiza un bucle comprobando si un worker tiene resultados listos (con el uso de la función no bloqueante: Iprobe), y en caso afirmativo lo acepta inmediatamente.
- El cálculo de los mapas finales en el master se debe realizar a partir de los mapas enviados por los workers. Con el uso de un hilo receptor de resultados y otro sumador de resultados en el master, se puede solapar la recepción de resultados con el cálculo de los mapas finales, sin necesidad de esperar a que todos los workers terminen de enviar todos los mensajes al master.

MEDICIONES

Para realizar las mediciones se han usado el mismo conjunto posible de parámetros. Las variables que se han tenido en cuenta durante las mediciones son:

- Número de nodos.
- Tamaño de las matrices que representan los maspas en la simulación (número de filas, número de columnas, ancho y alto de cada fila).
- Tamaño de los mensajes enviados a cada worker. El tamaño del mensaje se mide en base al número de combinaciones de parámetros que se envian en el mismo (considerarando el hecho de que los últimos mensajes podrían no tener este tamaño).

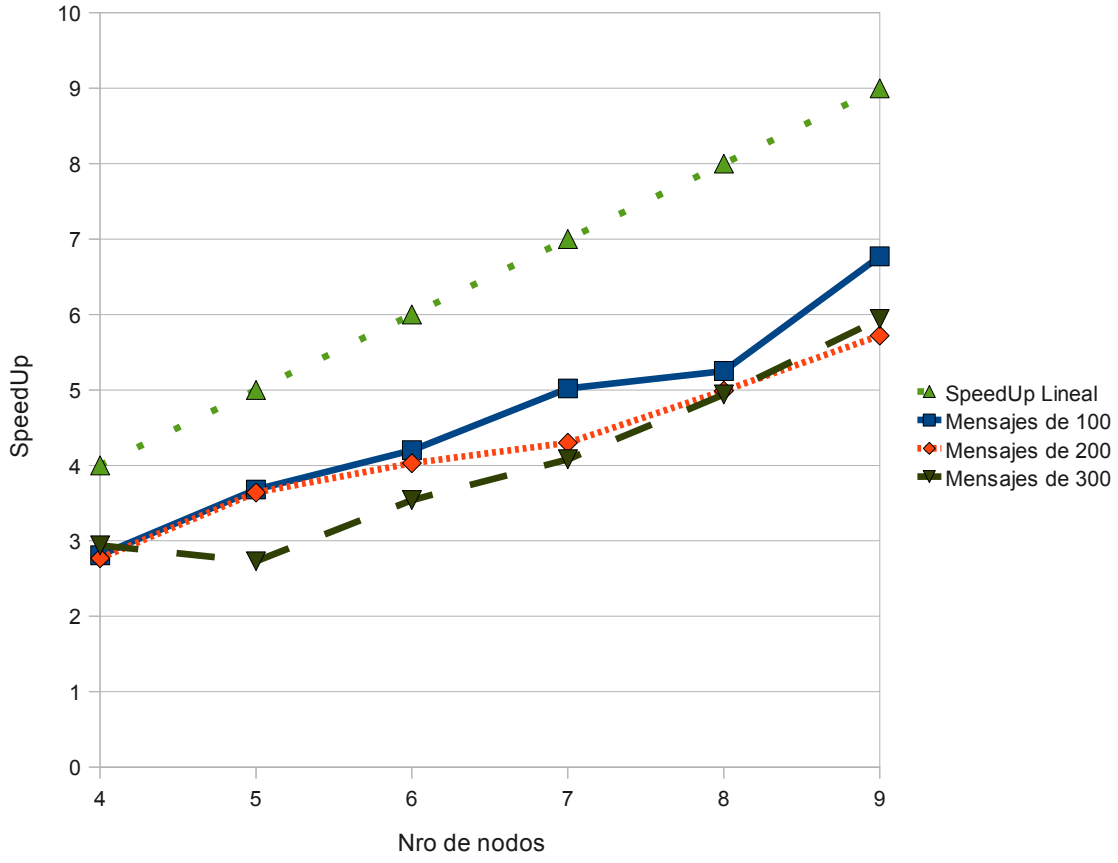
Este es el conjunto de valores que toma cada parámetro durante cada simulación:

Model: {1 ; 2 ; 3 }
WindSpd: { 4; 5 ; 6}
WindDir: { 0.0 ; 0.1; 0.2}
Slope: {0.0}
Aspect: {0.1; 0.2}
M1: {0.1; 0.2}
M10: {0.1}
M100: {0.10 ; 0.2}
Mherb: {1.00 ; 2.00 }
Mwood: { 1.50; 2.0; 2.5}

De modo que el número total de combinaciones de parámetros a probar son 1296.

En la siguiente tabla se muestra los resultados de las mediciones para el algoritmo secuencial, el algoritmo paralelo (teniendo en cuenta tres tamaños diferentes de mensajes), y teniendo en cuenta matrices de 101 filas, 101 columnas, con celdas de 100 por 100 de tamaño:

Tiempo algoritmo secuencial (segundos)	**Tiempos algoritmo paralelo (segundos)**							
	Nro de nodos	Nro de workers	Para mensajes de 100 combinaciones	**SpeedUp**	Para mensajes de 200 combinaciones	**SpeedUp**	Para mensajes de 300 combinaciones	**SpeedUp**
547.03	9	8	80.85	6.77	95.57	5.72	92.02	5.94
547.03	8	7	104.12	5.25	109.61	4.99	110.64	4.94
547.03	7	6	109.01	5.02	127.13	4.30	134.22	4.08
547.03	6	5	130.19	4.20	135.79	4.03	154.46	3.54
547.03	5	4	148.54	3.68	150.36	3.64	200.53	2.73
547.03	4	3	194.9	2.81	197.72	2.77	186.32	2.94

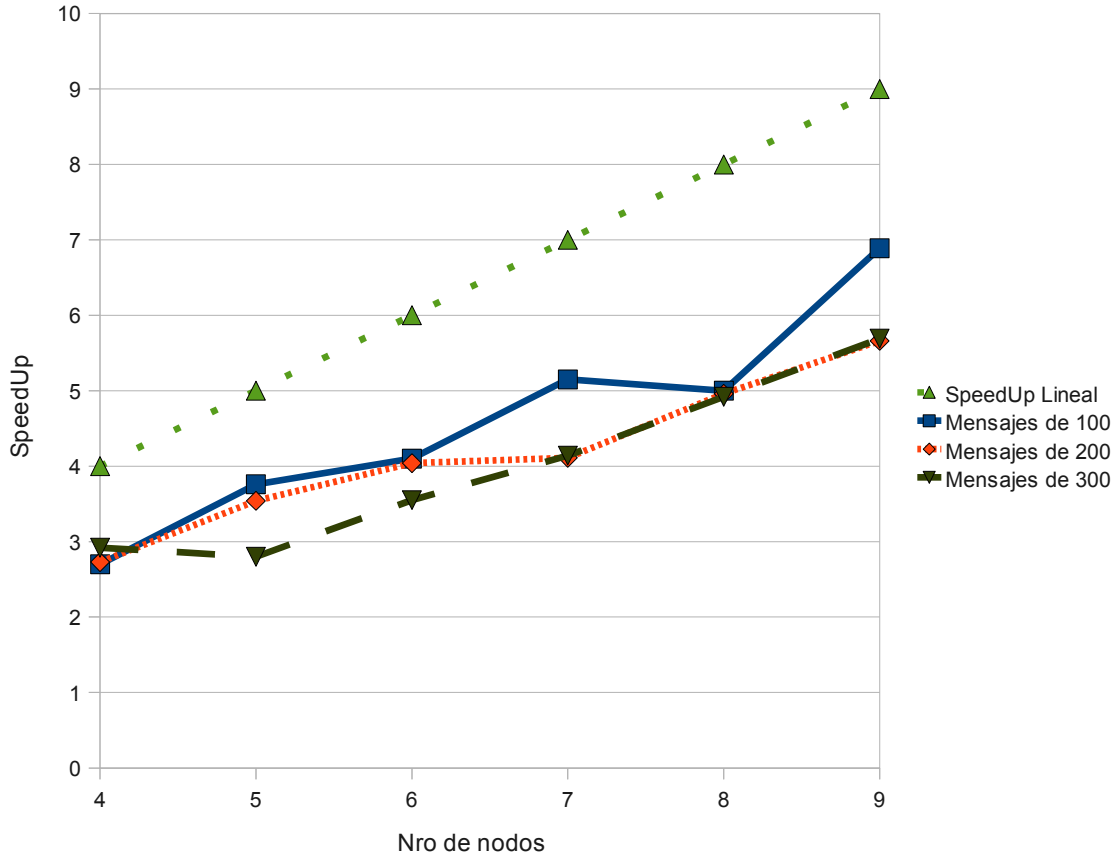


El siguiente gráfico es un gráfico de SpeedUp en base a los datos de la tabla de arriba:

En la siguiente tabla se muestra los resultados de las mediciones para el algoritmo secuencial, el algoritmo paralelo (teniendo en cuenta tres tamaños diferentes de mensajes), y teniendo en cuenta matrices de 121 filas, 121 columnas, con celdas de 100 por 100 de tamaño:

Tiempo algoritmo secuencial (segundos)	**Tiempos algoritmo paralelo (segundos)**							
	Nro de nodos	Nro de workers	Para mensajes de 100 combinaciones	**SpeedUp**	Para mensajes de 200 combinaciones	**SpeedUp**	Para mensajes de 300 combinaciones	**SpeedUp**
1099.33	9	8	159.55	6.89	194.35	5.66	193.05	5.69
1099.33	8	7	220.01	5.00	221.55	4.96	223.27	4.92
1099.33	7	6	213.27	5.15	267.2	4.11	265.53	4.14
1099.33	6	5	268.11	4.10	272.4	4.04	309.32	3.55
1099.33	5	4	292.4	3.76	310.32	3.54	392.2	2.80

1099.33	4	3	407.91	2.70	403.41	2.73	375.94	2.92



El siguiente gráfico es un gráfico de SpeedUp en base a los datos de la tabla anterior:

CONCLUSIÓN

Como se observa en las dos gráficas de SpeedUp obtenidas anteriormente se puede ver que, como se esperaba, a medida que aumente el número de nodos, mejora el speedUp. En cuanto al tamaño de los mensajes enviados se puede ver que, mientras más pequeños son los mensajes se obtienen mejores SpeedUps, a medida que aumenta el número de nodos. Esto último contradice a lo que se esperaría comúnmente, ya que se supone que mientras menor número de mensajes haya, el tiempo de respuesta será mejor; sin embargo ocurre lo contrario. Esto se puede deber a que al usar hilos en cada worker, éstos requieren de un suministro constante e inmediato de combinaciones de parámetros para comenzar las simulaciones. Entonces si se usan mensajes pequeños, el master los puede armar rápidamente y enviarlos de inmediato a cada worker, de modo que cada uno de los workers no tiene que esperar demasiado antes de recibir el primer mensaje. Para cuando se han terminado las simulaciones con las combinaciones anteriores, el hilo receptor de mensajes en el worker ya habrá recibido la siguiente tanda de combinaciones. **Es decir, que un worker tarda más tiempo en realizar una simulación con un determinado conjunto de parámetros que en recibir un mensaje del master.**
En cambio si se usan mensajes de gran tamaño, cada worker se queda esperando más tiempo hasta que

llegue un mensaje.
Finalmente, se concluye que el uso de hilos junto con el paso de mensajes, mejora significativamente el rendimiento de una aplicación. Casi no hay tiempos osciosos cuando hay comunicación, ya que se solapa con cómputo. Se estima que si se realizan simulaciones con mayor número de combinaciones, se pueden llegar a obtener mejores SpeedUp aproximadamente iguales al SpeedUp Lineal.