Paralelización de RSA con Open MPI y POSIX Threads

Christofer Chávez Carazas

17 de julio de 2017

Motivación

- Operaciones con números grandes.
- Paralelizar la exponenciación modular con *OpenMP* para reducir el tiempo de encriptado y desencriptado. del *RSA*
- Creación de hilos inecesarios.

RSA

- Sistema criptográfico asimétrico.
- Seguridad basada en dos números primos grandes.
- Usa al exponenciación modular.

Generación de llaves

1. Se escogen aleatoriamente dos números primos grandes q y p.
2. Se obtiene el módulo $n = q * p$.
3. Se calcula $\varphi(n) = (p-1)(q-1)$
4. Se selecciona un e que cumpla: $1 < e < \varphi(n)$ y que sea coprimo con $\varphi(n)$.
5. Se selecciona un d que cumpla: $1 < d < \varphi(n)$ tal que $e * d = 1 \; mod \; \varphi(n)$.

Cifrado y Descifrado

Cifrado

$$c \equiv m^e (mod\ n)$$

Donde c es el mensaje cifrado, m es el mensaje original, e es la llave pública y n es el módulo hallado anteriormente.

Descifrado

$$m \equiv c^d (mod\ n)$$

Donde m es el mensaje original, c es el mensaje cifrado, d es la llave privada y n es el módulo hallado anteriormente.

Repeated square-and-multiply Algorithm

El algoritmo se basa en la siguiente observación:

$$g^e mod\ m = (g^{e/2} * g^{e/2}) mod\ m$$

y se puede definir recursivamente de la siguiente manera:

ModExp(g,e,m) =	**1,**	**if e = 0**
	(g X ModExp(g, (e-1), m)) mod m,	**if e is odd**
	ModExp(g, e/2, m)^2 mod m,	**if e is even**

Figura: *Repeated square-and-multiply* [S. Saxena et al. 2014]

Exponenciación Modular binaria Paralela

- Se convierte el exponente a su forma binaria y se lo divide en k partes dependiendo del número de procesadores.
- La exponenciación tendría la siguiente forma:

$$g^e = g^{2^{r_k} e_k} \cdots g^{2^{r_2} e_2} * g^{2^{r_1} e_1}$$

siendo las r calculadas de la siguiente forma:

$$r_1 = 0, r_2 = \frac{n}{k}, r_3 = \frac{2n}{k}, \cdots, r_k = \frac{(k-1)n}{k}$$

- Cada parte se calcula con el *Repeated square-and-multiply Algorithm*

Propuesta

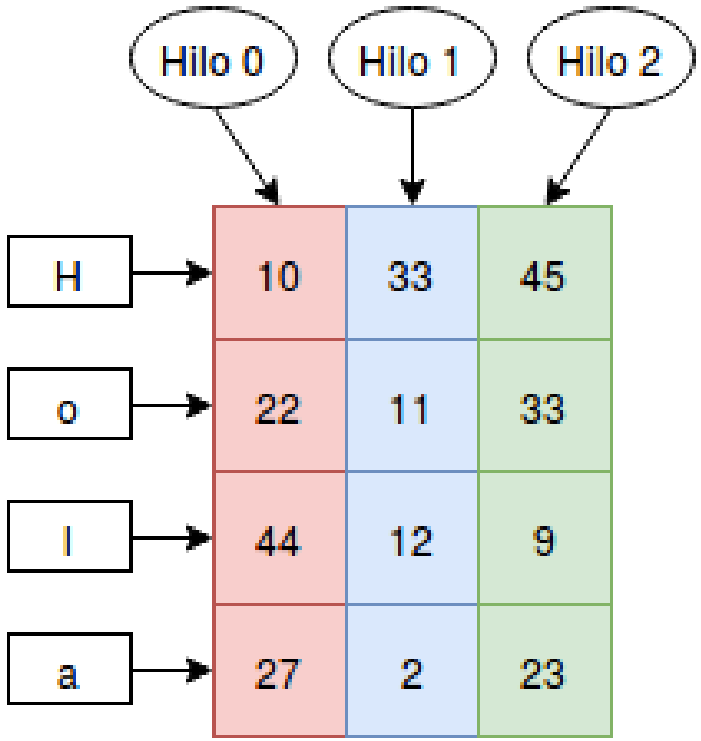
- Utilizar *Open MPI* para paralelizar el cifrado y descifrado del mensaje.
- Utilizar *POXIS Threads* para paralelizar la exponenciación modular.
- Optimizar la creación de hilos y la distribución de los datos.

Distribución en MPI

- Se distribuye el mensaje entre los procesos.
- Se convierte la key en su forma binaria y se copia en todos los procesos.

Distribución en Pthreads

- Se crea una matriz compartida entre los hilos.
- Cada hilo opera una columna de la matriz.
- Se crea un array compartido entre los hilos.
- Se reparten las filas de la matriz entre los hilos.
- El resultado de la multiplicación se guarda en el array.


Hilo 0
Hilo 1
Hilo 2
H
o
l
a
10
33
45
22
11
33
44
12
9
27
2
23

SetUp

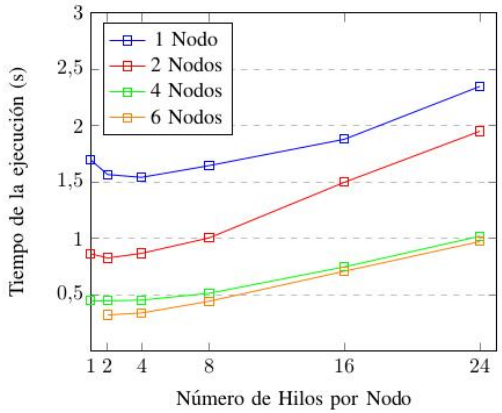
Los experimentos se realizaron en el supercomputador de la UNSA

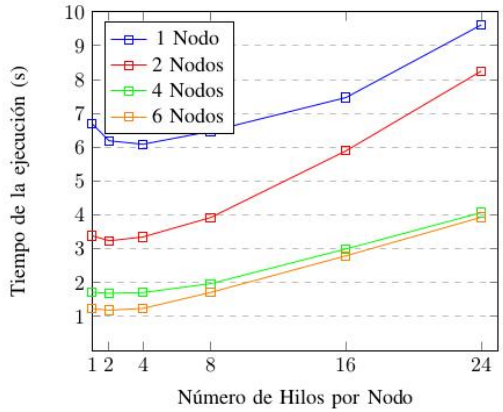
- GPUs: 24 x 2.60 GHz.
- Memory(RAM): 62.90 GB.
- SO: Linux 3.10.0-514.16.1.el7.x86 64 (x86 64).

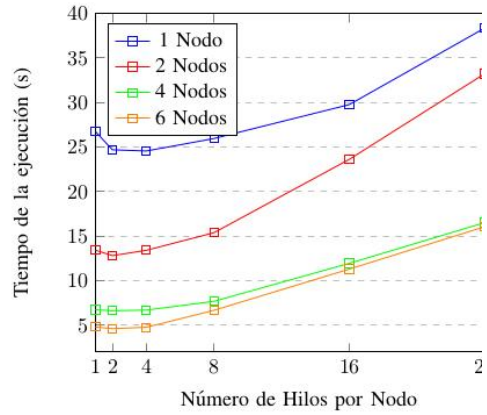
Características de los experimentos

- Llaves de 512 bits y 1024 bits.
- Combinaciones entre el número de nodos (1,2,4,6) y el número de hilos por nodo (1,2,4,8,16,24).
- Tres mensajes generados aleatoriamente de tamaños 1024, 4096, 16392 caracteres.
- Cada experimento se ejecutó 25 veces y se sacó el promedio.

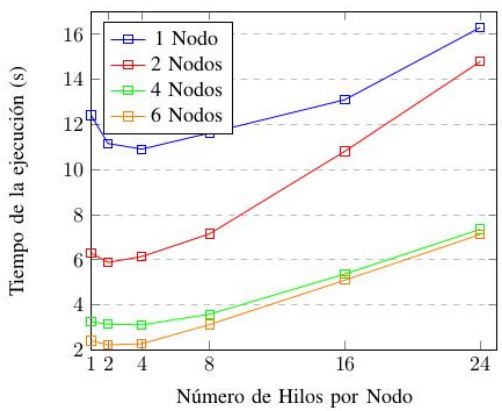
Resultados con key de 512 bits

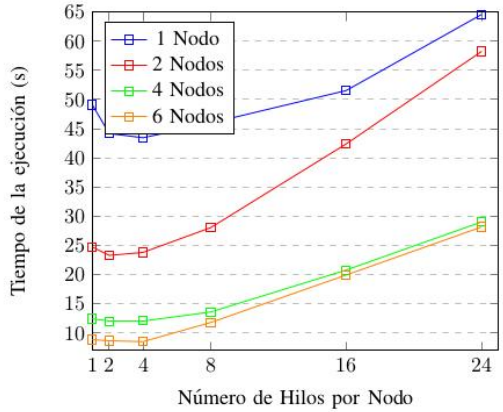


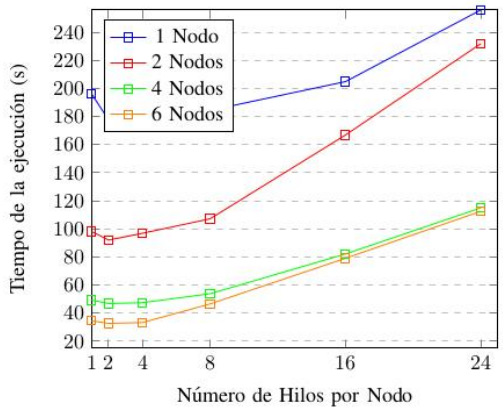




Resultados con key de 1024 bits







Conclusiones

- Se logra paralelizar el RSA de forma optima en la parte de MPI.
- Aunque se pueda ganar tiempo al no crear más hilos de los necesarios, se necesita de más memoria para guardar las matrices creadas en cada proceso.

Sapna Saxena, Bhanu Kapoor **An Efficient Parallel Algorithm for Secured Data Communications Using RSA Public Key Cryptography Method**, 2014.