Implementació de l'algorisme de Wiener i Oorschot en un entorn MPI

Escola Politècnica Superior

Universitat de Lleida

Javier Majadas Hernández

Directors: Josep Maria Miret Biosca
Francesc Sebé Freixas

Setembre 2009

Agraïments

Primer de tot, voldria agrair al Francesc Sebé i Josep Maria Miret per l'ajuda que m'han donat, sobretot en els últims mesos per poder acabar aquest projecte. Agrair-los també l'oportunitat de poder realitzar una projecte com aquest i amb uns equips que d'una altra manera no hages pogut emprar.

També donar les gràcies a tots aquells professors que m'han ajudat per poder entendre certs aspectes del projecte que no acabava de veure clar, especialment els professors Fernando Cores, Francesc Giné, Xavier Domínguez , entre d'altres. Sense ells tampoc estaria aquí. No voldria oblidar-me dels meus companys, que també m'han ajudat en la mesura que han pogut, i en particular el Pere Santallucia, que m'ha dedicat part del seu temps, tot i tenir la seva feina i un altre noi tutelat.

I per últim, no voldria deixar d'agrair a la meva família, que han estat al meu costat des del principi i que sempre m'han animat quan semblava que tot fallava.

Pròleg

Aquest projecte presenta una breu introducció a la criptologia. S'expliquen principis fonamentals, com què és la criptografia i el criptoanàlisi i els mètodes més rellevants de cada cas. Això servirà com a base teòrica per estudiar el funcionament del criptosistema de ElGamal, la seguretat del qual es basa en la dificultat de resoldre el problema del logaritme discret. Un cop tenim clar el problema del logaritme discret, s'implementarà una aplicació que el resolgui, mitjançant l'algorisme Rho de Pollard. Aquesta aplicació contarà amb el suport de la llibreria NTL, llibreria de nombres gegants, per poder implementar-la.

Per acabar, i com a principal objectiu, el que es pretén és implementar una aplicació paral·lela que resolgui el problema del logaritme discret en un entorn multicomputador utilitzant la proposta de Wiener i Oorschot.

Índex

Índex de taules

Capítol 1

Objectius del projecte

L'objectiu d'aquest treball és implementar la proposta de Wiener i Oorschot per resoldre el problema del logaritme discret en un cluster que funciona utilitzant la llibreria MPI. Per fer això ens hem marcat els següents objectius:

- Cerca informació bàsica sobre criptologia.
- Instal·lar, entendre i fer anar la llibreria NTL, que ofereix la possibilitat d'emprar nombres gegants necessaris per la implementació dels algorismes.
- Implementar un programa que resolgui el problema del logaritme discret mitjançant l'algorisme Rho de Pollard utilitzant la llibreria NTL.
- Instal·lar, entendre i fer anar la llibreria MPI, que ofereix totes les eines de paral·lelització necessàries per la implementació de l'aplicació final.
- Entendre el funcionament de la proposta de Wiener i Oorschot, per la resolució paral·lela del problema del logaritme discret.
- Efectuar proves comparant l'aplicació en paral·lel amb l'aplicació no paral·lela per veure les millores que aporta.

Capítol 2

Preliminars matemàtics

En aquest apartat s'expliquen tots aquells conceptes matemàtics necessaris per poder entendre les explicacions dels algorismes. La majoria de les següents definicions han sigut extretes de [3].

Conjunt: Agrupació d'objectes, anomenats elements. Per objecte entenem no només ens físics, sinó també ens abstractes. La relació de pertinència entre els elements i els conjunts sempre és perfectament discernible, és a dir, si un objecte pertany a un conjunt o no sempre pot qualificar-se com verdader o fals.

Grup: Un conjunt G dotat d'una operació interna * té estructura algebraica de grup o, dit més simple, (G, *) és un grup si se satisfan les propietats següents:

1. Associativa;
2. Existeix element neutre;
3. Tot element de G té simètric.

Si, a més, se satisfà la propietat commutativa, es diu que (G, *) és un grup abelià. Sigui (G, *) un grup finit, és a dir, G és un conjunt finit que té estructura de grup per l'operació *. Aleshores:

- L'ordre de G és el nombre d'elements de G.
- L'orde d'un element a que pertany a G és el més petit dels exponents n que pertanyen a $\mathbb{N}$ tals que

$$a^n = a^* ...^n a = e,$$

- La del grup finit (G, *), on $G = \{a_1, a_n, ..., a_n\}$, s'obté de la manera següent:

*	a_1	a_2	$\dots$	a_j	$\dots$	a_n
a_1	b_{11}	b_{12}	$\dots$	b_{1j}	$\dots$	b_{1n}
a_2	b_{21}	b_{22}	$\dots$	b_{2j}	$\dots$	b_{2n}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
a_i	b_{i1}	b_{i2}	$\dots$	b_{ij}	$\dots$	b_{in}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
a_n	b_{n1}	b_{n2}	$\dots$	b_{nj}	$\dots$	b_{nn}

on l'element b_{ij} és l'element de G resultant d'operar a_i per a_j, és dir, $b_{ij} = a_i * a_j$. A la taula anterior també se l'anomena taula de Cayley de (G, *).

Anell: Un conjunt A dotat de dues operacions internes + i · té estructura d'anell o, dit més simplement, (A, +, ·) és un anell, si:

1. (A, +) és un grup abelià;
2. L'operació · és associativa;
3. L'operació · és distributiva respecte de l'operació +.

- L'anell (A, +, ·) és unitari si té element neutre respecte de l'operació ·, anomenat unitat.
- L'anell (A, +, ·) és commutatiu si l'operació · satisfà la propietat commutativa.
- Es diu que $a \in A - \{0\}$ és un *divisor de zero* d'un anell (A, +, ·), on 0 és el neutre de (A, +), si existeix un element $b \in A - \{0\}$ tal que $a \cdot b = 0$ o $b \cdot a = 0$. Un anell commutatiu unitari sense divisors de zero s'anomena *domini d'integritat*.

Cos: és un conjunt K dotat de dues operacions internes + i· si:

- (K, +,·) és un anell unitari;
- Tot element de K diferent del 0, on 0 és el neutre de (K, +), és invertible, és a dir

$$\forall a \in K - \{0\} \, \exists a^{-1} \in K \text{ tal que } a \cdot a^{-1} = a^{-1} \cdot a = 1$$

on 1 es el neutre de (K, ·) anomenat *unitat.* L'element a^{-1}es diu que és l'invers de a.

Si, a més, l'operació·satisfà la propietat commutativa, es diu que (K, +,·) és un *cos commutatiu*.

Factorització: és expressar un objecte o número com a producte d'altres objectes més petits (factors), en el cas de números hem d'utilitzar nombres primers, que al multiplicar-los tots, resulta l'objecte original. Factoritzar un número té un cost sub-exponencial, però per la contra, agafar factors i multiplicar-los per obtenir un únic número, té un cost polinòmic. Aquesta doble besant de la factorització fa que sigui molt utilitzat en criptografia. Això es degut a que multiplicar factors per generar un únic número és fàcil, però agafar un número i factoritzar-lo, sense cap més dada, és molt complicat. I com més gran sigui el número encara més. Avui dia encara no s'han trobat algorismes que resolguin aquest problema de forma eficient.

Logaritme discret: es coneix com a logaritme discret de x en base a mòdul n a resoldre l'equació $x = a^y \pmod n$ on x, n i a són constants i y és la incògnita. El fet d'aplicar aritmètica modular fa el problema de trobar y irresoluble en un temps raonable, per això s'utilitza en criptografia, en el mètode d'intercanvi de claus de Diffie-Hellman o el sistema de ElGamal. Això és perquè calcular la potència d'un número té un cost polinòmic, mentre que realitzar el calcul invers té un cost exponencial.

Capítol 3

Què és la Criptologia

Des de temps molt antics la humanitat ha sentit la necessitat de comunicar-se, inicialment d'una forma més arcaica com sons i gestos, fins als últims temps que només necessitem el telefon mòbil. Tot i que la forma a canviat el concepte és el mateix, transmetre una informació a una altra persona.

Paral·lelament, amb l'aparició de la necessitat de transmetre informació també apareix la necessitat d'amagar informació a un cert col·lectiu de persones. D'aquesta manera les persones van començar a desenvolupar sistemes per amagar informació per poder-la transmetre a altres persones. Però al igual que es vol amagar informació, també es vol descobrir possible informació amagada, així que apareixen sistemes per poder descobrir aquesta informació. A aquesta ciència se l'anomena Criptologia.

La Criptologia és la ciència que tracta els problemes teòrics relacionats amb la seguretat en l'intercanvi de missatges en clau entre un emissor i un receptor a través d'un canal de comunicacions. Aquesta ciència està dividida en tres grans branques: la criptografia, el criptoanàlisi, i l'esteganografia.

La Criptografia s'ocupa del disseny de procediments per a xifrar, es a dir, per emmascarar una determinada informació de caràcter confidencial. El Criptoanàlisi, per la seva part, s'ocupa de trencar aquest procediments de xifrat per així recuperar la informació original. Ambdues disciplines sempre s'han desenvolupat de forma paral·lela, degut a que qualsevol mètode de xifrat porta sempre emparellat el seu Criptoanàlisi corresponent.[4] L'esteganografia és la ciència que estudia els procediments encaminats a ocultar l'existència d'un missatge en lloc d'ocultar el seu contingut. No es tracten de ciències exclusives, sinó complementàries: ocultar un missatge redueix les possibilitats de que sigui descobert; no obstant, si es descobreix que aquell missatge hagi sigut xifrat introdueix un nivell addicional de seguretat.[6]

3.1 Criptografia

La criptografia engloba tots aquells procediments pels quals un emissor pot alterar la informació que vol transmetre fins que un receptor rep la informació i la torna a recompondre en el missatge original. Per tan podem dir que, originalment tenim un emissor que vol enviar un missatge a una persona en concret, receptor, sense que ningú el pugui llegir. Per tant aquest emissor el que fa és xifrar, mitjançant un algorisme, el missatge creant un criptograma. Un criptograma és un missatge xifrat que el seu significat resulta incoherent fins que no és desxifrat. Generalment, el contingut del missatge original és modificat seguint un determinat patró, de manera que sol és possible entendre el significat després de conèixer el patró seguit en el xifrat. Un cop el emissor té el criptograma,

l'envia al seu receptor, que ha de conèixer la manera de desxifrar-lo, és a dir, coneix la clau. Un cop rebut el missatge, el receptor ja pot desxifrar-lo i llegir el missatge que li han enviat.

D'això podem deduir que la criptografia té una finalitat múltiple. Primerament ens assegurem que la informació que enviem es totalment confidencial i que ningú la podrà llegir. En segon lloc, és un sistema d'autenticació, és a dir, només els legítims emissor i receptor poden llegir el missatge. I finalment aconseguim protegir el missatge de possibles manipulacions durant l'enviament. Englobant-ho tot, podrem dir que el conjunt de protocols, algorismes de xifrat, processos de gestió de clau i actuacions dels usuaris, és lo que constitueix en conjunt un criptosistema, que és amb lo que l'usuari final treballa i interactua.

Els mètodes criptogràfics es poden classificar de diverses maneres, però la classificació que englobaria a la resta seria segons les característiques de la seva clau de xifrat. D'aquesta manera podem diferenciar els mètodes criptogràfics de clau compartida i els mètodes criptogràfics de clau publica.

3.1.1 Criptografia de clau compartida

La criptografia de clau compartida engloba tots aquells mètodes de xifrat on la clau de xifrat és la mateixa que la clau de desxifrat. Això implica que la clau ha de ser coneguda tan per l'emissor com pel receptor. Com a resultat lògic, l'emissor i el receptor s'han tingut que ficar d'acord inicialment en quina clau utilitzarien.

Si algú volgués intentar desxifrar un missatge, primer de tot té que esbrinar quin algorisme s'ha fet servir i després quina és la clau. Aquest sistema ofereix una seguretat basada en la clau. Això vol dir que si un atacant trobés l'algorisme de xifrat i no té la clau, li resultaria impossible desxifrar el missatge. Això sembla tenir moltes avantatges, però a l'hora de la veritat no és així.

Per començar tan emissor com receptor han de ficar-se d'acord amb quina clau utilitzar, això implica un comunicació per algun canal no segur o una trobada cara a cara. En la majora de vegades la trobada cara a cara és impossible, per tan s'han de transmetre les claus per un canal no segur i vulnerable a atacs externs. Això es podria solucionar amb un centre de distribució de claus que les enviés per un canal segur.

Un altre problema que presenta aquest sistema és el número de claus que es necessita. Si prenem com exemple un grup de n persones que volen emprar el sistema de clau compartida i tenim un centre de distribució de claus, per cada parella necessitarem $\binom{n}{2}$ claus. Això pot funcionar en grups petits, però en grups grans seria impossible.

Antigament el sistema de clau compartida era l'únic utilitzat. Això va promoure l'aparició de diversos mètodes que han tingut molta rellevància històrica. Aquest sistemes són el xifrat per substitució i el xifrat per transposició.

En un sistema de xifrat per substitució, cada lletra o grup de lletres es reemplaça per una altra lletra o un grup de lletres que pot ser o no del mateix alfabet per camuflar-les. En la recepció, el legítim receptor, que coneix així mateix la correspondència establerta, la clau, substitueix cada símbol per la lletra corresponent de l'alfabet original, recuperant la informació inicial.[5]

En canvi, en un sistema de xifrat per transposició consisteix en barrejar els símbols del missatge original col·locant-los en un ordre diferent, de manera que el criptograma contingui els mateixos elements del text original, però col·locats de tal forma que que resulten incomprensibles. En la recepció, el legítim receptor, que coneix així mateix la correspondència establerta, la clau, recol·loca els símbols desordenats del criptograma en la seva posició original.[5]

Tan el xifrat per substitució com el xifrat per transposició no resulten molt efectius utilitzats individualment, no obstant això constitueixen la base de sistemes molt més difícils de criptoanalitzar. Aquest tipus de xifrat ara per ara està obsolet i ja no s'utilitzen. Els mètodes que s'utilitzen ara són el DES i el AES.

El DES, Data Encryption Standard, és un algorisme de xifrat per blocs. Aquest algorisme agafa un text de longitud fixa de bits i el transforma mitjançant una serie de complicades operacions en un altre text de la mateixa longitud. En el cas de DES la mida del bloc es de 64 bits. DES utilitza també una clau criptogràfica per modificar la transformació, de manera que el desxifrat sol pot ser realitzat per aquells que coneguin la clau concreta utilitzada únicament per comprovar la paritat, i després són descartats. Per tant, la longitud de la clau efectiva en DES es de 56 bits, i així és com se sol especificar.[5]

El AES, Advanced Encryption Standard, és un algorisme de xifrat per blocs. AES té una mesura de bloc fixa de 128 bits i té unes mesures de clau de 128, 192 o 256 bits. La majoria de càlculs de l'algorisme AES es fan amb un camp finit determinat. AES opera en una matriu 4x4 bytes, anomenada state. L'algorisme AES passa per quatre fases per generar el codi xifrat. Les fases són: substitució de bits, desplaçar files, barrejar columnes i per últim càlcul de subclaus.

Dintre de la criptografia de clau compartida podem fer una classificació en dos grans blocs, el xifrat en bloc i el xifrat en flux. Aquesta classificació ve donada pel tipus de font que genera els texts.

El xifrat en bloc és aquell que agafa el text original i el divideix en parts iguals, blocs, i els va xifrant part per part. A la recepció només s'ha d'anar desxifrant els blocs i recuperar el missatge original. Els bloc són independents els uns dels altres i gracies a aquest fet no cal tenir tot el text sencer per poder desxifrar-lo, podem anar desxifrant bloc per bloc.

El xifrat en flux s'aplica sobre texts formats per lletres, combinant-les amb un flux de bits secrets, seqüència xifrant, mitjançant un operador *. El desxifrat es realitza de forma anàloga, combinant mitjançant l'operador *^1 les lletres del text xifrat amb la seqüència xifrant, produint així les lletres del text original. La seqüència xifrant es produeix mitjançant un generador de bits.[5]

3.1.2 Criptografia de clau pública

Fins ara havíem vist sistemes criptogràfics basats en una sola clau, que és compartida, i que han de conèixer l'emissor i el receptor. Aquest sistema presenta diferents problemes, abans esmentats. Per solucionar-los va aparèixer la criptografia de clau pública.

Aquest sistema utilitza dos claus. Una clau per xifrar i una per desxifrar. El sistema està pensat per a que cada persona genera les seves dos claus. Una serà la de xifrar, que la farà pública perquè la gent pugui xifrar els missatge que li vulguin enviar, i l'altra serà una clau privada que tindrà el receptor per poder desxifrar el missatge rebut.

Aquest sistema es basa en l'existència d'un tipus de funcions unidireccionals com a funció invertible. Aquestes funcions s'utilitzen per xifrar el text, i estan pensades per fer que xifrar el text sigui fàcil i en canvi desxifrar-lo sigui molt difícil. Per poder-les utilitzar en criptografia és defineix un tipus especial de funció unidireccional que s'anomenen funcions unidireccionals amb trampa. Bàsicament són funcions unidireccionals que si tens una petita informació, la part difícil que seria el desxifrant, es torna fàcil.

D'aquesta manera tenim que cada usuari té dos claus, que estan relacionades entre elles, però sense cap més dada és impossible deduir o extreure la clau privada basant-nos en la pública. Llavors, cada persona fa pública la clau de xifrat. Així tothom podrà xifrar missatges i els podrà enviar al

receptor. Per la seva banda, el receptor rebrà missatges xifrats i només ell els podrà llegir ja que és l'únic que coneix la clau de desxifrat. Per tant, el receptor el que fa és convertir un procés difícil en fàcil perquè ell té una drecera.

Cal dir que aquest tipus de funcions no se sap si existeixen, tot i que es pressuposa la seva existència. Hi ha dues funcions candidates a ser-ho: el producte de números enters, que la seva inversa és la factorització del número obtingut i l'exponenciació discreta, que la seva inversa és el logaritme discret. Les dues funcions són fàcils de computar, mentre que les seves inverses no ho són. És a dir, donat un número n és difícil determinar la seva descomposició en factors primers, i per l'altra banda, donat un grup (G, *) tal que $b = a^n$, és difícil calcular x que fa que es compleixi: $a^x = b$.

D'aquestes dos funcions en surten els dos sistemes de xifra més coneguts, el RSA i ElGamal.

L'algorisme RSA és un algorisme de xifrat basat en la factorització. És un dels sistemes més emprats avui dia i està integrat en la majoria de navegadors d'Internet que utilitzem quotidianament.

3.1.2.1 ElGamal

ElGamal va proposar un esquema de clau pública basat en l'exponenciació discreta sobre un grup multiplicatiu d'un cos finit Zp. No obstant, aquí presentaré un protocol més general al proposat pel ElGamal sobre un grup finit (G, *).

Suposem que l'usuari A desitja enviar un missatge m a l'usuari B. El protocol mencionat anteriorment és el següent:

1. Es selecciona un element α de G.
2. Cada usuari A elegeix un número aleatori a, que serà la seva clau privada, i calcula α^a en G, que serà la seva clau pública.

Per a que un usuari A enviï un missatge, m, a un altre usuari B, suposant que els missatges son elements de G, realitza les següents operacions:

1. A genera un número aleatori v i calcula α^v en G.
2. A mira la clau pública de B, α^b, i calcula $\left(\alpha^b\right)^v$ i $m * a^{bv}$ en G.
3. A envia la parella (α^v, $m * a^{bv}$) a B.

Per recuperar el missatge original:

1. B calcula $(\alpha^v)^b$ en G.
2. B obté m només amb calcular $m * a^{bv} * a^{vb}$

Per seguretat i eficàcia, el grup G i l'element α haurien d'elegir-se de tal manera que verifiquin les següents condicions:

- Per eficàcia, l'operació * en G hauria de ser fàcil d'aplicar.
- Per seguretat, el problema del logaritme discret en el subgrup cíclic de G generat per α, $\langle\alpha\rangle$ hauria de ser difícil.

Per a simplificar el protocol anterior, podem suposar, tal i com va ser descrit per ElGamal, que el grup sobre el que es duu a terme les operacions mencionades en el protocol anterior és el grup multiplicatiu del cos $\mathbb{Z}_p$; d'aquesta manera, les potencies i productes anteriors s'efectuen mòdul un número primer p.

Per veure com funciona l'algorisme descriuré un petit exemple extret de [4]

Considerarem el grup $\mathbb{Z}^*_{15485863}$, amb $p = 15485863$ primer i un generador $\alpha = 7$.

A i B elegeixen les següents claus: a = 28236, b = 21702, $\alpha^a = 7^{28236} = 12506884$ (mod 15485863) i $\alpha^b = 7^{21702} = 8890431$ (mod 15485863), que són les claus privades i publiques de A i B, respectivament.

Suposarem que A vol enviar a B el missatge m = <HIJO>. Per duu a terme la codificació el que farem és representar la paraula canviant cada lletra pel número que ocupa la seva posició dins de l'alfabet. Un cop fet això, multiplicarem cada número per 26 elevat a una potència que dependrà de la posició que ocupa la lletra dins de la paraula. Començant per la dreta, que tindrà potència 0, fins potència 3, en aquest cas. D'aquesta manera obtenim el següent missatge:

$$\text{m = HIJO} = 7\cdot 26^3 + 8\cdot 26^2 + 9\cdot 26 + 14 = 128688$$

A elegeix un número $v = 480$ i calcule $\alpha^v = 7^{480} = 12001315$ (mod 15485863). A continuació A calcule el valor $\left(\alpha^b\right)^v = 8890431^{480} = 9846598$ (mod 15485863), el de $m\cdot\alpha^{bv} = 128688\cdot 9846598$ = 8263449 (mod 15485863) i decodifica el parell format per: $(\alpha^v, m\cdot\alpha^{bv}) = (12001215, 8263449)$, que és el missatge encriptat:

$\alpha^v = 12001315 = 1\cdot 265 + 0\cdot 264 + 6\cdot 263 + 21\cdot 262 + 11\cdot 26 + 1 = \text{BAGVLB}$,

$m\cdot\alpha^{bv} = 8263449 = 18\cdot 264 + 2\cdot 263 + 4\cdot 262 + 0\cdot 26 + 25 = \text{SCEAZ}$.

Per tant, el missatge a envia a B és: (BAGVLB, SCEAZ).

Veiem com recupere B el missatge rebut.

En primer lloc, B codifica en base 26 la parella rebuda:

$\text{BAGVLB} = 1\cdot 265 + 0\cdot 264 + 6\cdot 263 + 21\cdot 262 + 11\cdot 26 + 1 = 12001315 = \alpha^v$,

$\text{SCEAZ} = 8263449 = 18\cdot 264 + 2\cdot 263 + 4\cdot 262 + 0\cdot 26 + 25 = 8263449 = m\cdot\alpha^{bv}$.

A continuació B calcula $\left(\alpha^v\right)^b = 12001315^{21702} = 9846598$ (mod 15485863) i després té que calcular

$$\frac{m\cdot\alpha^{vb}}{\alpha^{vb}}$$

Per això ha de determinar l'invers de α^{vb} mòdul 15485863. La determinació d'aquest invers es duu a terme per mitja de l'algorisme d'Euclides estes, és a dir, com $-662582\cdot\alpha^{vb} + 421299\cdot p = 1$. Resulta que:

$$\frac{1}{\alpha^{vb}} = -66258 = 14823281 (mod 15485863)$$

$$\frac{m\cdot\alpha^{vb}}{\alpha^{vb}} = \frac{8263449}{9846598} = 8263449\cdot 14823281 = 128688 = m$$

i recupera el missatge original:

$$\text{m} = 128688 = 7\cdot 263 + 8\cdot 262 + 9\cdot 26 + 14 = \text{HIJO}$$

Si alguna persona estigués escoltant la conversa coneixeria G, n, $\alpha^a, \alpha^b, \alpha^v, m\cdot\alpha^{vb}$.

3.2 Criptoanàlisi

Fins ara hem vist tot de mètodes per xifrar informació. Però per cada mètode de xifrat existeix, com a mínim, un mètode per atacar el xifrat i desxifrar el missatge sense tenir les claus. Per tant podem dir que, el criptoanàlisi engloba tots aquells mètodes per poder atacar un criptosistema. Cada criptosistema té les seves debilitats i es vulnerable a l'hora d'atacar-lo de certa manera, per això existeixen tants sistemes de criptoanàlisi com sistemes criptogràfics.

La criptografia i el criptoanàlisi sempre s'han vist com ciències contràries, ja que el criptoanàlisi ataca sistemes criptogràfics, però seria més encertat dir que són complementaris, ja que el criptoanàlisi realment el que fa es demostrar la seva vulnerabilitat.

En aquest projecte el que es fa és estudiar un criptosistema concret, ElGamal, i veure com se'l pot atacar. Un cop tenim tota la informació de com atacar-lo el que es fa és crear un programa que executi un seguit d'algorismes per extreure la clau privada i així poder desxifrar el missatge sense ser els legítims receptors.

3.2.1 Atac al logaritme discret.

En aquest treball el que s'ha fet és estudiar ElGamal, un criptosistema que basa la seva seguretat en la dificultat de resoldre el logaritme discret. Però per resoldre el logaritme discret hi ha diferents mètodes per poder resoldre'l, que depenen del tipus de grup on estigui definit el logaritme discret. Aquest mètodes són el Baby Step-Giant Step, el Pohling-Hellman i el Index-Calculus.

3.2.2 Resolució del logaritme discret.

Després d'haver estudiat com funciona el criptosistema de ElGamal, podem veure que un observador extern pot obtenir certes dades, que en principi no li permetran recuperar el missatge enviat. Si es tenen les eines adequades i els coneixements concrets es pot intentar desxifrar un missatge que ha sigut codificat mitjançant el mètode anterior.

Per atacar un criptosistema que empra ElGamal, farem anar l'algorisme Rho de Pollard. Aquest algorisme s'utilitza per resoldre el problema del logaritme discret, en el qual es basa ElGamal. Donats un primer p i donats també dos elements g i h, aquest algorisme busca un element m tal que $g^m = h \pmod{p}$. Per implementar aquest algorisme també és necessari conèixer l'ordre de g dintre de $\mathbb{Z}_p$. A aquest odre l'anomenarem q. En el nostre cas, q = (p-1)/2.

L'algorisme va recorrent una seqüència d'elements E_i de la forma $E_i = g^{a_i} \cdot h^{b_i} \pmod{p}$. Per començar, es defineix E_0 = 1 amb lo qual $a_0 = 0$ i $b_0 = 0$. D'un determinat element E_i, se'n guarda la tupla (E_i, a_i, b_i).

Donat un element de la seqüència(E_i, a_i, b_i), se'n defineix el següent (E_{i+1}, a_{i+1}, b_{i+1}) de la següent manera:

- Si $E_i \pmod 3 = 0 \rightarrow$ (E_{i+1}, a_{i+1}, b_{i+1}) = ($g \cdot E_i \pmod{\text{p}}$, a_i+ 1 (mod q), b_i).
- Si $E_i \pmod 3 = 1 \rightarrow$($E_{i+1}$, a_{i+1}, b_{i+1}) = ($h \cdot E_i \pmod{\text{p}}$, a_i, b_i + 1 (mod q)).
- Si $E_i \pmod 3 = 2 \rightarrow$($E_{i+1}$, a_{i+1}, b_{i+1}) = ($(E_i)^2 \pmod{\text{p}}$, $2 \cdot a_i \pmod{\text{q}}$, $2 \cdot b_i \pmod{\text{q}}$).

Amb aquest salts condicionals aconseguim que els resultats obtinguts entrin en un bucle, és a dir, a partir d'un E_i concret, els resultats obtinguts s'aniran repetint. El pseudocodi que representaria tota la part de buscar la E_i seria el següent:

Tortuga:= (1, 0,0)

Llebre:= (1, 0,0)

Llebre:= Següent(Llebre)

mentre Llebre.E $\neq$ Tortuga.E fer

Tortuga:= Següent(Tortuga)

Llebre:= Següent(Llebre)

Llebre:= Següent(Llebre)

fi mentre

Escriure (Tortuga.a – Llebre.a)(Llebre.b – Tortuga.b)$^{-1}$ (mod q)

Quan tinguem el mateix E_i, però amb diferents a_i i b_i només en de seguir els passos següents:

- Tenim (E_i, a_i, b_i) i (E_j, a_j, b_j) tal que$E_i = E_j$.
- Sabem que, $E_i = g^{a_i} \cdot h^{b_i}$ (mod p) i $E_j = g^{a_j} \cdot h^{b_j}$ (mod p).
- Pe tant, podem dir que: $g^{a_i} \cdot h^{b_i} = g \cdot h^{b_j}$.
- Si agrupem les g's a un cantó i les h's en un altre tenim: $g^{a_j} \cdot g^{-a_i} = h^{b_i} \cdot h^{-b_j}$.
- Si traiem factor comú obtenim: $g^{(a_j - a_i)} = h^{(b_i - b_j)}$.
- Si ara fem el logaritme en base g als dos cantons de la igualtat: $log_g g^{(a_j - a_i)} = log_g h^{(b_i - b_j)}$.
- Sabent la propietat dels logaritmes què diu que el logaritme d'una potència és igual a l'exponent pel logaritme de la base de la potència, tenim: $(a_j – a_i) \cdot log_g g = (b_i – b_{j)} \cdot log_g h$.
- Sabem que $log_g g = 1$, per tant podem escriure :$(a_j – a_i) = (b_i – b_j) \cdot log_g h$.
- Si passem el $(b_i – b_j)$ dividint a l'altre costat de la igualtat tenim:

$$log_g h = \frac{(a_j - a_i)}{(b_i - b_j)}$$

Com que a_j, a_i, b_i i b_j són coneguts, podem calcular el $log_g h$ d'una manera indirecta i molt més ràpida.

Per dur a terme tot lo explicat anteriorment, hem de crear un codi que, primerament ens busqui dos tuples on les E_i siguin iguals. Un cop trobades l'únic que hem de fer és programar al codi els càlculs adients per trobar la clau privada i així desxifrar el missatge.

Capítol 4

Paral·lelització

L'algorisme abans esmentat no deixa de ser un algorisme seqüencial, és a dir, per ser executat de forma lineal. Però si el que volem es millorar encara més l'algorisme de Rho de Pollard el que hem de fer es adaptar el codi per que es pugui executar de forma paral·lela en diferents ordinadors.

Hem de tenir present que en paral·lel disposarem de diversos processadors, on un agafara el rol de pare i els altres de fills. La idea és la mateixa que en l'algorisme en serie, buscar dues tuples amb al mateixa E_i, però amb diferents a_i i b_i. Els fills només s'encarreguen de generar tuples, que van enviant al pare. Ell és qui s'encarrega de emmagatzemar les tuples enviades i de mirar si troba dues tuples que compleixin la condició. Per fer això, hem de plantejar que cada fill inicialitzi una tupla (E_i, a_i, b_i), cadascun en un punt diferent. Els fills no han d'enviar totes les tuples generades, perquè necessitaríem molta memòria. Les tuples enviades podem dir que estan "marcades". Els fills només han d'enviar les tuples on la E_i tingui un número concret de bits a zero en la part baixa. D'aquesta manera acotem el nombre d'enviaments.

El pare per la seva banda el que a de fer és, primer de tot crear un hash obert on es guardaran les tuples enviades pels fills. Després és queda en un estat on només rep tuples dels fills. Cada vegada que rep una tupla, comprova si té la tupla dins del hash. Si no la té la guarda i continua rebent tuples. En el cas de que ja tingui la tupla, recupera la tupla i fa parar als fills. Acte seguit començar a fer els càlculs pertinents per calcular l'exponent que estem buscant.

Algorithm 4.1 Pseudocodi del pare

```
BOOL trobat:= fals
BOOL parar:=fals
VECTOR rebut(0, 0, 0)
HASH taula
ENTER a1:=0
ENTER b1:=0
ENTER s:=0
ENTER tamany:=obtenir_tamany()
mentre trobat == fals fer
rebre _vector(rebut)
    trobat:= taula.existeix(rebut.E)
    si trobat==veritat fer
        taula.consultar(rebut.E, a1, b1)
        si rebut.a==a1 fer
            trobat:=fals
            taula.inserir(rebut)
    sinó fer
        taula.inserir(rebut)
fi mentre
taula.consultar(rebut.e, a1, b1)
para:=veritat
mentre s<tamany fer
    enviar_fills(parar)
fi mentre
Escriure (rebut.a – a1)(b1 – rebut.b)^-1 (mod q)
```

Algorithm 4.2 Pseudocodi dels fills

```
ENTER k // definida per l'usuari
PARAR:=false
VECTOR enviar(0,0,0)
VECTOR busqueda(0,0,0)
iniciar_vector(busqueda)
rebre_pare(parar)
mentre para==false fer
    zeros:= Numero_zeros(busqueda.E)
    si zeros==k fer
        enviar.E:=Recuperar_E(busqueda)
        enviar.a:=Recuperar_a(busqueda)
        enviar.b:=Recuperar_b(busqueda)
        enviar_vector(enviar)
        busqueda:=Següent(busqueda)
    sinó fer
        busqueda:=Següent(busqueda)
    Actualitzar(parar)
fi mentre
```

Capítol 5

Llibreries

Per dur a terme aquest projecte ha sigut necessari emprar un tipus concret de llibreries en C++. Aquestes llibreries ofereixen unes funcions necessàries i úniques que sense elles aquest projecte no s'hagués pogut implementar.

5.1 NTL

La llibreria NTL[1] és una llibreria en C++ portable i de gran eficiència que proporciona estructures de dades i algorismes per manipular longituds arbitràries de sencers, vectors, matrius, polinomis sobre nombres enters i sobre camps finits.

Per poder fer més segura una clau a l'hora de xifrar, un punt a tenir en compte és que sigui una clau amb molt dígits. Això sempre complica molt la feina a l'hora de descobrir-la. Amb l'aparició d'ordinadors, la potencia de càlcul ha augmentat molt, i això provoca que les claus cada cop han de ser més llargues, fins al punt de superar la llargada màxima que un ordinador normal no és capaç de calcular.

De forma natural els ordinadors normals, tenen una llargada màxima a l'hora de treballar amb números. Si volem crear programes que superin aquesta llargada ja hem d'emprar eines especials. En el cas d'aquest treball, s'ha utilitzat la llibreria NTL, compatible amb C++, que permet treballar amb números molt més grans. Aquesta llibreria implementa una classe anomenada ZZ, que és l'eix principal de tot el codi generat en aquest treball.

Aquesta classe és la que permet que es defineixin números de llargada superior a la normal. Per tant si definim una variable del tipus ZZ, podrà ser un número tan llarg com sigui necessari. No ens haurem de preocupar per si superem el màxim nombre de dígits, ja que aquesta classe no en té.

Per poder treballar amb els ZZ aquesta llibreria té sobrecarregats els operadors bàsics, és a dir, la suma, la resta, la multiplicació i la divisió. També sobrecarregats els operadors d'igualtat i desigualtat. Així que a una variable tipus ZZ la podem tractar com si fos una variable de tipus INT. I no només això, sinó que podem fer tot tipus d'operacions combinant tipus de de variables, és a dir, que un ZZ i un INT els podem sumar, multiplicar, restar i dividir. I no només amb el INT, sinó també amb altres tipus de enters.

A més a més, la llibreria NTL implementa tot un seguit de funcions d'aritmètica modular molt útils en aquest projecte donat al fet que sempre hem de treballar en un mòdul concret. Aquestes operacions faciliten la feina d'anar modulant tota la estona els resultat de les operacions bàsiques.

Les funcions pròpies de la llibreria NTL que s'utilitzen en aquest projecte són:

- void AddMod(ZZ& x, const ZZ& a, const ZZ& b, const ZZ& n), que equival a fer l'operació: x = (a+b)%n
- void MulMod(ZZ& x, const ZZ& a, const ZZ& b, const ZZ& n), que equival a fer l'operació: x = (a*b)%n
- void PowerMod(ZZ& x, const ZZ& a, const ZZ& e, const ZZ& n), que equival a fer l'operació: x = a^e % n
- void SqrMod(ZZ& x, const ZZ& a, const ZZ& n), que equival a fer l'operació: x = a^2 % n
- void InvMod(ZZ& x, const ZZ& a, const ZZ& n), que equival a fer l'operació: x = a^{-1} mod n (0 <= x < n)

A part de les funcions d'aritmètica modular abans explicades hi ha unes altres funcions que han sigut necessàries per convertir una variable ZZ a un array de UNSIGNED CHAR, a part d'una funció per calcular el número de bits necessaris per dur a terme aquesta transformació. Aquestes funcions són:

- long NumBytes(const ZZ& a), aquesta funció retornar un número de tipus LONG amb la llargada necessària que ha de tenir l'array d'UNSIGNED CHAR per poder representar un ZZ.
- void BytesFromZZ(unsigned char *p, const ZZ& a, long n), aquesta funció s'encarrega de transformar un ZZ a un array d'UNSIGNED CHAR.
- void ZZFromBytes(ZZ& x, const unsigned char *p, long n), aquesta funció retorna l'array d'UNSIGNED CHAR a ZZ.

També és necessari generar nombre aleatoris per tal d'inicialitzar de forma aleatòria certes variables, i per això fem anar les següents funcions:

- void SetSeed(const ZZ& s): aquesta funció inicialitza una llavor. Aquesta llavor es utilitzada per generar el primer estat de les crides per generar números aleatoris. L'idea és que les s que passem per paràmetre tinguin molta entropia.
- ZZ RandomBnd(const ZZ& n): aquesta funció genera un número aleatori de rang 0..n-1. Per cridar aquesta funció és necessari haver inicialitzat una llavor amb la funció SetSeed.

5.2 MPI

Molts dels algorismes de criptoanàlisi comporten una carrega computacional molt elevada, provocant que els ordinadors triguin una quantitat de temps molt elevada. Per reduir el temps dels algorismes el que es fa es distribuir-los en diferents ordinadors per poder accelerar els càlculs i així trobar la resposta molt més ràpid.

La llibreria MPI[2] és una llibreria que implementa tot un seguit de funcions per poder distribuir les tasques d'un programa per poder ser executat en diferents ordinadors. D'aquesta manera aconseguim que programes lents executats en un sol ordinador passin a millorar el seu rendiment al ser executats en diversos ordinadors.

Les funcions pròpies de la llibreria MPI que s'utilitzen en aquest projecte són:

- void MPI::Init(int& argc, char**& argv): funció que indica l'inici del codi que executaran tots els ordinadors.
- int MPI::Comm::Get_size(): funció que retornar el número d'ordinadors que estan executant el codi.
- int MPI::Comm::Get_rank(): funció que retornar l'identificador de cada ordinador.
- void MPI::Comm::Send(const void* buf, int count, const Datatype& datatype, int dest, int tag): aquesta és la funció bàsica per enviament de dades de la MPI. Les dades que li passem són, per odre, un buffer d'enviament, número de dades, tipus de dades, rang del node al que s'envie les dades i l'etiqueta que diferencia al missatge per ignorar-la en MPI_Recv.
- void MPI::Comm::Recv(void* buf, int count, const Datatype& datatype, int source, int tag, Status& status): aquesta és la funció bàsica per rebre dades de la MPI. Les dades que li passem són, per odre, un buffer de recepció, número de dades, tipus de dades, rang del node al que es rep les dades i l'etiqueta que diferencia al missatge per ignorar-la en MPI_Recv. Mentre aquesta funció no rebi cap dada, qui l'hagi cridat no farà res més que escoltar.
- Request Comm::Irecv(void* buf, int count, const Datatype& datatype, int source, int tag): aquesta funció és igual que l'anterior amb la diferencia que, aquesta funció no bloqueja el funcionament de l'algorisme. Mentre no rebi cap dada, l'execució del programa continuarà normalment. Quan rep una dada, la funció retorna un Request que es pot consultar.
- bool Request::Test(Status& status): funció que serveix per veure l'estat del Request que retorna la Irecv. Mentre la Irecv no rebi cap dada, el valor que retorna aquesta funció és fals. En el moment que la Irecv rebi una dada, aquesta funció retorna cert.
- int Status::Get_Source(): funció que retorna l'identificador del fill que ha enviat l'ultim missatge.
- void MPI::Finalize(): funció que es fica per indicar que el codi que s'ha d'executar per tots ha acabat.

Capítol 6

Implementació

En aquest apartat s'expliquen totes les decisions relacionades amb la implementació de l'algorisme de Rho de Pollard en C++.

Per començar cal explicar que dins de l'algorisme podem diferenciar dos grups de dades, el E_i i la a_i i b_i. Es poden diferenciar pel fet que E_i treballa en mòdul p i a_i i b_i treballen en mòdul q. Aquest fet a provocat certes decisions a l'hora de generar el codi. Degut a que les dades treballen en mòduls diferents comporta que sempre hem de tenir present cada dada té el seu mòdul correcte. A més, les dades que treballen en mòdul p fan anar unes operacions mentre que les que treballen en mòdul q en fan anar unes altes.

D'aquesta manera, es poden crear dos classes, que són el grup additiu i el grup multiplicatiu.

6.1 Grup Additiu

La classe grup_additiu està dividida en dos arxius, el grup_additiu.h i el grup_additiu.cpp. En el grup_additiu.h està definida com és la classe i quines operacions pròpies té, mentre que en el grup_additiu.cpp estan definides les funcions. El tipus grup_additiu està definit amb dos atributs que són un ZZ q i un ZZ element. D'aquesta manera cada element del tipus grup_additiu té el seu valor, que seria l'element, i el mòdul en el que treballa, que seria la q.

Les funcions pròpies del grup_additiu són:

- grup_additiu(): és la constructora per defecte de la classe. S'encarrega d'inicialitzar el q i l'element a 0.
- grup_additiu(ZZ q1): és la constructora per paràmetres de la classe. Continua inicialitzant l'element a 0, però inicialitza q amb el mòdul que li passem.
- void set_q(ZZ q1): funció que ens permet modificar el valor del q.
- void set_element(ZZ element1): funció que ens permet modificar el valor de l'element.
- ZZ get_q() const: funció que ens permet recuperar el valor actual del q.
- ZZ get_element() const: funció que ens permet recuperar el valor actual de l'element.
- void suma(grup_additiu a, grup_additiu b): funció que s'encarrega de sumar dos dades del tipus grup_additiu i de ficar el resultat en el mòdul q.

6.2 Grup Multiplicatiu

La classe grup_multiplicatiu que està dividida en dos arxius, el grup_multiplicatiu.h i el grup_multiplicatiu.cpp. En el grup_multiplicatiu.h està definida com és la classe i quines operacions pròpies té, mentre que en el grup_additiu.cpp estan definides les funcions. El tipus grup_multiplicatiu està definit amb dos atributs que són un ZZ p i un ZZ element. D'aquesta manera cada element del tipus grup_multiplicatiu té el seu valor, que seria l'element, i el mòdul en el que treballa, que seria la q.

Les funcions pròpies del grup_multiplicatiu són:

- grup_multiplicatiu(): és la constructora per defecte de la classe. S'encarrega d'inicialitzar el q i l'element a 0.
- grup_multiplicatiu(ZZ p1): és la constructora per paràmetres de la classe. Continua inicialitzant l'element a 0, però inicialitza p amb el mòdul que li passem.
- void set_p(ZZ p1): funció que ens permet modificar el valor del p.
- void set_element(ZZ element1): funció que ens permet modificar el valor de l'element.
- ZZ get_p() const: funció que ens permet recuperar el valor actual del p.
- ZZ get_element() const: funció que ens permet recuperar el valor actual de l'element.
- void producte(grup_multiplicatiu a, grup_multiplicatiu b): funció que s'encarrega de multiplicar dos dades del tipus grup_multiplicatiu i de ficar el resultat en el mòdul p.
- void exponent(grup_multiplicatiu a, grup_multiplicatiu b): funció que s'encarrega d'exponenciar dos dades del tipus grup_multiplicatiu i de ficar el resultat en el mòdul p.

Amb la creació d'aquestes dues classes el que aconseguim és independitzar les dades i oblidar-nos de tenir que treballar amb mòduls.

6.3 Vector

Un cop tenim creades les classes grup_additiu i grup_multiplicatiu, el que fem és crear una altra classe anomenada vector i ens servira per poder implementar les tuples formades per(E_i, a_i, b_i). La classe vector que està dividida en dos arxius, el vector.h i el vector.cpp. En el vector.h està definida com és la classe i quines operacions pròpies té, mentre que en el vector.cpp estan definides les funcions. El tipus vector està definit amb tres atributs que són un tipus grup_multiplicatiu anomenat E i dos són del tipus grup_additiu anomenats a i b.

Les funcions pròpies de la classe vector són:

- vector(): és la constructora per defecte de la classe. S'encarrega de les constructores per paràmetre del grup_multiplicatiu i del grup_additiu i les inicialitza a 0.
- vector(ZZ pa): és la constructora per paràmetres de la classe vector. Aquesta funció crida les constructores per paràmetre de les classes grup_multiplicatiu i grup_additiu. El paràmetre que li passem a la constructora li passa per paràmetre a la constructora de la classe grup_multiplicatiu. Aquesta funció també s'encarrega de calcular el paràmetre que se li passa a la constructora per paràmetre del grup_additiu.

- ~vector(): funció destructora, encarregada d'alliberar els recursos de memòria de l'ordinador.
- void set_a(ZZ pa): funció que ens permet modificar el valor del a.
- void set_b(ZZ pb): funció que ens permet modificar el valor del b.
- void set_E(ZZ pe): funció que ens permet modificar el valor del E.
- ZZ get_E() const: funció que ens permet recuperar el valor actual del E.
- ZZ get_a() const: funció que ens permet recuperar el valor actual del a.
- ZZ get_b() const: funció que ens permet recuperar el valor actual del b.
- ZZ get_p() const: funció que ens permet recuperar el valor actual del p.
- ZZ get_q() const: funció que ens permet recuperar el valor actual del q.
- void incrementar_a(): funció que augmenta en una unitat el valor de a conservant el mòdul.
- void incrementar_b(): funció que augmenta en una unitat el valor de b conservant el mòdul.
- void elevar_E(): funció que eleva a 2 el valor de E conservant el mòdul.
- int moduladora(): funció que retorna el resultat de passar el valor de E a mòdul 3.
- void multiplicar_a(): funció que multiplica per dos el valor de a conservant el mòdul.
- void multiplicar_b(): funció que multiplica per dos el valor de b conservant el mòdul.
- void següent(ZZ g1, ZZ h1): funció que calcula el següent valor que a de prendre una tupla seguint l'algorisme de Rho de Pollant. Per paràmetres rep el generador g i la clau pública h. Després crida a la funció moduladora i mira el resultat obtingut, i depenent de quin sigui, calcula E, a i b segons l'algorisme de Rho de Pollard.
- bool comparadora(vector* pv): funció que compara dos tuples i comprova si les E són iguals.

6.4 Programa en serie

Un cop tenim aquestes classes creades, només cal crear el codi del programa principal, anomenat . El programa té com entrada un primer p, un generador g i la clau pública h. El codi començarà per buscar les dues tuples on les les E siguin iguals. En principi, el codi tindria que guardar totes les tuples i comparar-les totes cada cop que se'n genera una de nova, però això comportaria una necessitat de memòria molt gran. Per solucionar això farem anar un sistema que consisteix en tenir únicament dues tuples, una anomenada llebre i una altra anomenada tortuga. Aquesta manera de solucionar-ho provoca que l'algorisme perdi certa efectivitat, ja que al no poder guardar en memòria, les dues tuples que sorgiran no seran les primeres que siguin iguals.

Nosaltres el que busquem és trobar dues tuples amb les E iguals. Sabem que en un moment o altre, per l'algorisme de Rho de Pollard, s'entrarà en un grup de resultats que s'anirà repetint i només canviarà la a i la b. El que fem és calcula el salt següent de la tupla llebre dos vegades i de la tupla tortuga només calculem el següent salt. I anem comparant el resultat obtingut. D'aquesta manera

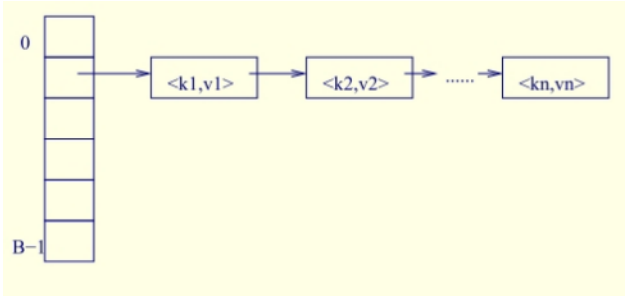
el que aconseguim es que per cada salt que fa la tortuga, la llebre en fa dos. Es podria dir que si la tortuga avança d'un en un, la llebre avança de dos en dos. Per tant si tenim present que en un moment o altre entrarem en un bucle de resultats, quan les dues tuples estiguin dins del bucle, la llebre enganxarà a la tortuga i tindrem les dues tuples amb les E iguals.

Un cop trobades les dues tuples, només cal calcular la clau pública utilitzant la formula deduïda anteriorment.

Fins aquí tindríem un programa que sabent el generador, la clau pública i el nombre primer calcula la clau privada. Però amb aquest programa no es suficient. Pel fet que les dades que utilitza per calcula la clau privada són nombres molt grans, el comput és molt elevat. Per això el següent pas es paral·lelitzar el programa per tal de millorar el seu rendiment.

6.5 Hash obert

Una de les qüestions que sorgeixen a l'hora de paral·lelitzar és que tenim que emmagatzemar certs resultats, ja que no podem fer servir la mateixa estratègia que en la versió en serie. Per això hem d'emprar una estructura de dades que sigui prou gran per encabir tota la informació i que sigui de ràpid accés. Per això farem anar l'estructura de hash, i per ser més exactes, un hash obert. Aquesta estructura ens permet guarda el nombre de dades que faci falta, ja que esta implementat de forma dinàmica. A més a més, l'estructura hash permet inserir la informació a través d'una clau. Aquesta clau, k, es genere a partir d'una de les dades que se li passa, ja que tenim una funció que transforma aquesta dada en la seva respectiva clau. Depenent com sigui la funció, podem tenir claus repetides. Però al tenir un hash obert el que fem és a la posició de la clau, k, anem inserint valors, de tal manera que, el primer valor inserit apunta al següent. El segon valor, apuntaria al tercer, i així fins arribar a l'últim. A la imatge següent podem veure un esquema de com guarda la informació un hash obert:



En el cas d'aquest projecte, s'ha implementat un has molt específic. Aquest hash està dividit en dos arxius, el hash.h i el hash.cpp. En el hash.h està definida com és la classe juntament amb les seves operacions. A part també hi ha definida la classe node i quines funcions té, ja que és una classe necessària per implementar el hash.

El tipus node està definit amb tres atributs del tipus ZZ anomenats E, a i b i amb un apuntador de tipus node que es diu següent. Les funcions que té definida la classe node són:

- node(): és la constructora per defecte de la classe. S'encarrega d'inicialitzar la E, la a i la b a 0, i l'apuntador següent a NULL
- node(ZZ* vec): és la constructora per paràmetres de la classe. Inicialitza la E amb el valor guardat a vec[0], la a amb el valor de vec[1] i la b amb el vec[2]. L'apuntador següent l'inicialitza a NULL.

- ~node(): funció destructora, encarregada d'alliberar els recursos de memòria de l'ordinador.

Per la seva part, la classe hash està implementada com un array d'apuntadors a node. I les funcions que té definides són:

- hash(): és la constructora per defecte de la classe. Aquesta funció inicialitza tots els nodes del vector cridant a la constructora per defecte de la classe node.
- int funcio_hash(ZZ E): funció que passat un valor E, retorna la seva clau k. Aquesta funció el que fa és, retornar la E en mòdul 100.
- int funcio_hash(ZZ* vec): funció que passat un vector vec, recupera el valor del vec[0] i retorna aquest valor en modul 100.
- void inserir(ZZ* vec): funció encarregada d'inserir un vector dins del hash. La funció el que fa és cridar a la funció_hash, calcular la clau k i mirar si la posició està lliure o esta ocupada. Si esta lliure ocupa la posició i acaba la inserció. En canvi, si està ocupada el que fa és inserir el nou vector el primer de la llista encadenada.
- bool existeix(ZZ E): funció que mira si existeix dins del hash una E ja inserida. Retorna cert en cas que la trobi, i fals en cas de que no hi sigui.
- void consultar(ZZ E, ZZ &a, ZZ &b): funció que passada una E la busca dins dels hash i copia la a i la b per poder-les utilitzar posteriorment.

6.6 Programa en paral·lel

Com he dit prèviament per paral·lelitzar farem anar les funcions de la llibreria MPI. Però abans de res, cal dir que aquesta llibreria només té suport per tipus bàsics de dades, és a dir, no està preparada per enviar dades del tipus ZZ. Per tant s'han creat unes funcions per solucionar aquest problema.

Les funcions creades per poder enviar dades amb la MPI estan basades en el fet que hi ha una funció de la llibreria NTL que transforma els ZZ a UNSIGNED CHAR. La MPI tampoc enviar UNSIGNED CHAR, però es el principi per començar a treballar.

Abans de res, comentar que a partir d'ara es començarà a treballar a nivell de bit. És a dir, cada UNSIGNED CHAR realment són 8 bits que codifiquen un ZZ. La codificació que fa anar no permet mostrar per pantalla el UNSIGNED CHAR, degut a que és codi intel·ligible. Si sabem que la MPI només pot enviar CHAR, el que farem és convertir el nostre array de UNSIGNED CHAR a un array de CHAR. Però per poder fer que es pugui mostrar per pantalla, no només passarem de UNSIGNED CHAR a CHAR, sinó que a més a més, farem que el codi intel·ligible passi a estar a codificació en números en hexadecimal. A més a més, també desfarem tots aquest canvis am una funció que passi de hexadecimal al codi intel·ligible i que passi de un array de char a un array de unsigned char.

Per fer tot això fer anar les funcions:

- void convertir1(unsigned char* p, char* &q, long n): aquesta funció rep com entrada un array de unsigned char, un array de char i la longitud que tenen els arrays. El que fa aquesta funció és recórrer l'array de unsined char i a cada posició la divideix en dos. Això s'aconsegueix agafant una posició i de l'array i dividir-la per 16 i guardant-la a la posició j de

l'array de char. Amb això fem que els 4 bits de major pes passin a ser el de menor pes i el de major pes passen a ser ceros. A la mateixa posició i de l'array de unsigned char ara el que fem és el mòdul en base 16 i la copiem a la posició k+1 de l'array de char. Així el que fem es guardar el 4 bits de menor pes i els de major pes es fiquen a 0. En breus paraules, cada posició de l'array de unsigned char es divideix en dos, 4 bits de menor pes i 4 bits de major pes, i es van copiant a l'array de char. Al final només cal copiar la marca de final de cadena que és el '\0'.

- void convertir2(unsigned char* &p, char* q, long n): aquesta funció el que fa és retornar el array de char a un array de unsigned char. Primer de tot, transforma el contingut hexadecimal de l'array de char al codi intel·ligible que hi havia originalment. El següent que fa és a les posicions que havia dividit entre 16, ara les multiplica per 16 per col·locar els bits de major pes al seu lloc. Un cop fet això, va sumant per parells les posicions de l'array de char i les va guardant a l'array de unsigned char. D'aquesta manera recuperem el codi intel·ligible i ordenat un altre cop.

- void mapejar: aquesta funció és l'encarregada de passa un número de hexadecimal a binari i de binari a hexadecimal. La funció mira si el caràcter correspon a un codi ASCII, que seria un caràcter en hexadecimal, i el transforma a binari. De la mateixa manera mira si el que el caràcter guardat és un número binari, i el transforma en el seu corresponent caràcter hexadecimal en codi ASCII.

Donat al fet que quan s'utilitza la llibreria MPI no es poden entrar dades per l'entrada estàndard, les dades s'han d'entrar amb la mateixa crida del programa. Aquestes dades queden guardades com un array de caràcters. Com en aquest projecte necessitem tenir ZZ, s'ha creat dos funcions per transformar el vector de caràcters a un ZZ. Les funcions són:

- void mapejar2(char p, ZZ &j): aquesta funció rep com entrades un caràcter i un ZZ on es copiara el resultat. El que fa aquesta funció és, mirar quin caràcter hi ha, i el canvia per la seva equivalència a número decimal.

- void convertir(char* v, ZZ &k): aquesta funció rep com entrades un array de caràcters i un ZZ on es copiarà el resultat. El que fa aquesta funció es transformar l'array de caràcters a un ZZ per després poder-lo utilitzar.

Les funcions que s'explicaran a continuació són funcions que han sigut creades per simplificar i fer més entenedor el codi. Aquestes funcions són:

- void enviar_vector(ZZ* vec): funció que rep com entrada el vector de ZZ que vol enviar. Aquesta funció s'encarrega de convertir el vector de ZZ a arrays de caràcters i els envia un a un al pare.

- void rebre_vector(ZZ* vec): funció que rep com entrada un vector de ZZ buit on es copiara el resultat. Aquesta funció el que fa és rebre un per un els arrays de caràcters enviats pels fills i els transforma en un vector de ZZ que copia al vector que se li ha passat a l'entrada.

- void iniciar_tupla(vector* vf, ZZ gf, ZZ hf): aquesta funció rep com a paràmetres d'entrada un vector on es copiara el resultat i dos ZZ gh i hf. Aquesta funció inicialitza de manera aleatòria els paràmetres a i b, i genera la E, seguint la formula $E = gf^a \cdot hf^b$.

Capítol 7

Resultats obtinguts

En aquest apartat veurem els resultat de diferents proves que hem realitzat amb els programes que hem creat. Com que el programa en paral·lel té un component aleatori, això provoca que podem tenir resultats diferents. Per això el resultats que es mostraran dels programes en paral·lel són la mitjana de cinc resultats obtinguts amb les mateixes dades d'entrada.

A continuació es mostra una taula on hi ha tots els resultats obtinguts en les proves efectuades. Cada columna representa quina aplicació s'ha emprat per realitzar la prova. Cada fila representa el resultat obtingut amb l'aplicació quan li passem un nombre de *n* bits, on *n* està indicat al principi de cada fila.

	Serie	**2 nodes**	**4 nodes**	**8 nodes**
44	66,10	11,87	13,23	11,47
46	90,14	18,82	11,92	17,68
48	131,78	62,43	16,47	17,88
50	278,11	122,47	60,11	44,89
52	437,55	225,55	137,33	101,64
54	3.078,49	853,96	522,18	323,75
56	7.494,26	1.021,58	386,65	349,56
58	13.181,69	998,66	386,65	787,34
60	-	-	990,05	733,22

Taula 7.1: Taula resultats dels experiments efectuats

El SpeedUp, s'obté dividint el temps de l'execució en serie pel temps de l'execució en paral·lel. Aquesta dada fa referència a la millora en temps que suposa utilitzar el programa en paral·lel respecte al programa en serie.

L'Eficiència es calcula dividint el SpeedUp pel nombre de nodes que s'han executat. Això ens aporta la informació de com millora individualment cada procés respecte a l'execució en serie.

El SpeedUp i l'Eficiència, ens donen una dada amb la que podem valorar com millore el rendiment de l'algorisme respecte l'execució en serie.

A continuació, es mostren 3 taules on es pot veure la comparació parcial entre el temps en serie i el temps d'una execució en paral·lel. A més a més, s'han afegit dues columnes més que són la del SpeedUp i l'Eficiència respectives.

En aquesta taula podem veure la comparació entre l'execució en serie i una execució en paral·lel amb dos nodes. En teoria tindríem que veure que el SpeedUp tindria que tenir un valor igual o superior a 2, i com és pot comprovar en tots els casos tret d'un es compleix. El factor probabilístic que té l'algorisme en paral·lel fa que no puguem assegurar la reducció de temps, i en un cas la reducció no es l'esperada, però podem concloure que funciona adequadament.

	Serie	**2 nodes**	**SpeedUp**	**Eficiència**
44	66,10	11,87	5,57	2,78
46	90,14	18,82	4,79	2,39
48	131,78	62,43	2,11	1,06
50	278,11	122,47	2,27	1,14
52	437,55	225,55	1,94	0,97
54	3.078,49	853,96	3,60	1,80
56	7.494,26	1.021,58	7,34	3,67
58	13.181,69	998,66	13,33	6,67

Taula 7.2: Taula de SpeedUp i Eficiència de 2 nodes

En aquesta taula podem veure la comparació entre l'execució en serie i una execució en paral·lel amb quatre nodes. En teoria tindríem que veure que el SpeedUp tindria que tenir un valor igual o superior a 4, i tornem a tenir que tret en un cas tots ho compleixen.

	Serie	**4 nodes**	**SpeedUp**	**Eficiència**
44	66,10	13,23	5,00	1,25
46	90,14	11,92	7,56	1,89
48	131,78	16,47	8,00	2,00
50	278,11	60,11	4,63	1,16
52	437,55	137,33	3,19	0,80
54	3.078,49	522,18	5,90	1,47
56	7.494,26	386,65	19,38	4,85
58	13.181,69	386,65	22,10	5,52

Taula 7.3: Taula de SpeedUp i Eficiència de 4 nodes

En aquesta taula podem veure la comparació entre l'execució en serie i una execució en paral·lel amb vuit nodes. En teoria tindríem que veure que el SpeedUp tindria que tenir un valor igual o superior a vuit, però en aquest cas només es compleix quan comparem en execucions de nombres amb 54 o més bits.

	Serie	8 nodes	SpeedUp	Eficiència
44	66,10	11,47	5,76	0,72
46	90,14	17,68	5,10	0,64
48	131,78	17,88	7,37	0,92
50	278,11	44,89	6,20	0,77
52	437,55	101,64	4,30	0,54
54	3.078,49	323,75	9,51	1,19
56	7.494,26	349,56	21,44	2,68
58	13.181,69	787,34	16,74	2,09

Taula 7.4: Taula de SpeedUp i Eficiència de 8 nodes

De les resultats obtinguts podem extreure que les versions paral·leles milloren els resultats obtinguts respecte la versió en serie. En alguns casos en trobem que la millora no és l'esperada, això és degut a l factor probabilístic de la versió en paral·lel.

Aquest factor provoca que en ocasions, tinguem una millora molt més gran que l'esperada, que seria un dels casos ideals. Però també provoca que, en el pitjor dels casos podria provocar que fos més lent que la versió en serie.

També podem veure que la utilització de vuit nodes, només genera els resultats esperats a partir de nombres amb molt bits. No obstant continua millorant els temps de la versió en serie.

Bibliografia

[1] **NTL** - http://www.shoup.net/ntl/download.html

[2] **MPI** - http://java.sun.com/j2se/1.6.0_01/docs/api/index.html

[3] J. Gimbert, X. Hernandéz, N. López, J. Miret, R. Moreno i M. Valls, *Curs pràctic d'algebra per a informàtics,* Edicions de la Universitat de Lleida, Març 2009.

[4] A. Fúster, D. Martinez, L. Hernández, F. Montoya, J. Muñoz, *Técnicas criptográficas de protección de datos*, Alfaomega Grupo Editor, Abril 2001.

[5] http://rinconquevedo.iespana.es/rinconquevedo/criptografia/introduccion.htm

[6] http://www.mmc.igeofcu.unam.mx/LuCAS/Manuales-LuCAS/doc-unixsec/unixsec-html/node1.html