UNIVERSITÀ DEGLI STUDI DI PARMA
Dipartimento di Matematica e Informatica

Corso di Laurea in Informatica

Implementazione del crivello quadratico per la fattorizzazione: backup, benchmark e comunicazione di rete

Candidato: Ayoub Ouarrak
Matricola: 239228

Relatore: Prof. Alessandro Zaccagnini
Correlatore: Prof. Enea Zaffanella

Anno Accademico 2014/2015

Indice

Ringraziamenti

Mi sembra doveroso dedicare qualche riga a coloro che in parte o in totale mi hanno aiutato a raggiungere questo traguardo. Ringrazio in primis i miei genitori. Ringrazio il mio relatore Alessandro Zaccagnini, Enea Zaffanella, e in generale tutti i miei professori. Ringrazio i miei amici, compagni di corso e colleghi, in particolare Giuseppe Petrosino, uno dei migliori sviluppatori con cui ho lavorato, uomo dalle innovative idee, Lorenzo Pattarini, Victoria, Silvia Balzani, Andrea Chiastra e Paolo Bandini per il loro aiuto, Marco Grillo e in generale tutti gli altri miei compagni che hanno trovato il tempo di leggere queste righe. Ringrazio Luca Da Rin Fioretto per i suoi preziosi consigli.

"Computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty"

- Donald Knuth -

Introduzione

Fin dall'antichità l'uomo ha avuto la necessità di scambiare messaggi nascosti, necessità nata sopratutto per mantenere le informazioni segrete in conflitti militari, evitando che i propri messaggi vengano intercettati e decodificati dai nemici. Abbiamo testimonianze di sistemi crittografici già dai tempi dei Greci. Si racconta che il governo spartano trasmetteva messaggi ai suoi generali nel modo seguente: mittente e destinatario avevano due bastoni di eguale diametro. Il mittente avvolgeva intorno a tale bastone una pergamena su cui veniva poi scritto il messaggio in righe longitudinali. Una volta srotolato, il messaggio si presentava come una sequenza di lettere senza ordine, e poteva essere letto solo riavvolgendo la pergamena intorno a un bastone di uguale diametro. Altro esempio di sistema crittografico celebre è quello di Cesare che consisteva nel sostituire a ogni lettera del testo in chiaro la lettera che la segue di 3 posti. Per esempio, si sostituisce alla A la lettera D, alla B la lettera E e così via.

0.1 Crittografia a chiave pubblica

I sistemi elencati sopra vengono denominati sistemi **simmetrici o a chiave privata**, in quanto per decifrare il testo basta essere in possesso della chiave, nel primo caso la chiave è il diametro del bastone mentre nel secondo caso la chiave è 3, cioè il numero di posti in cui si trasla ogni lettera. Tali sistemi sono facilmente attaccabile.
Con i secoli sono stati introdotti diversi tipi di sistemi crittografici, nella metà degli anni $\acute{7}0$ è stato introdotto un nuovo sistema di **crittografia asimmetrica o a chiave pubblica**. Il sistema si basa sull'esistenza di due chiavi distinte, una viene usata per cifrare e l'altra per decifrare. Anche se entrambe le chiavi sono dipendenti tra loro non è possibile risalire ad una chiave conoscendo l'altra. Questi sistemi basano la loro sicurezza sull'elevata complessità computazionale di **fattorizzare** in numeri primi.

0.1.1 RSA

RSA è un algoritmo a chiave pubblica inventato nel 1977 da *Ronald Rivest, Adi Shamir e Leonard Adleman*, il cui funzionamento è il seguente:

1. Si scelgono a caso due numeri primi p e q grandi in modo da garantire la sicurezza dell'algoritmo
2. Si calcola il prodotto $n = pq$ e il prodotto $\varphi(n) = (p-1)(q-1)$
3. Si sceglie e coprimo con $\varphi(n)$ e più piccolo di $\varphi(n)$
4. Si calcola $d \mid ed \equiv 1 \mod \varphi(n)$

La **chiave pubblica** è (n, e), mentre la **chiave privata** è (n, d). Nonostante la coppia (n, e) sia pubblica, per ricavare d non è sufficiente essere in possesso solo di n, ma è necessario avere anche $\varphi(n) = (p-1)(q-1)$; tale calcolo, con i calcolatori attuali, richiede una quantità di tempo che va oltre le nostre aspettative, si parla di milioni di anni per una chiave di lunghezza RSA-2048. La sicurezza di RSA risiede nell'estrema quantità di tempo necessaria per fattorizzare in numeri primi (cioè scomporre un numero nei suoi divisori primi). Con l'aumentare delle performance dei moderni computer e con i miglioramenti apportati agli algoritmi di fattorizzazione, l'abilità di fattorizzare grossi numeri è aumentata. La robustezza di criptazione è direttamente collegata alla grandezza delle chiavi, duplicando la lunghezza delle chiavi si ottiene un aumento esponenziale, a discapito di una diminuzione delle performance. Se N ha una lunghezza uguale o inferiore a 300 bits, può essere facilmente fattorizzabile in un paio d'ore sulle macchine che ognuno di noi possiede usando software di pubblico dominio. Il più recente esempio di attacco al criptosistema RSA è quello avvenuto nel 2010, dove il numero RSA-768 è stato fattorizzato con successo dopo due anni di calcolo distribuito su centinaia di macchine. Alcuni esperti sostengono che anche la chiave 1024 bits sarà attaccabile in un futuro molto vicino, ecco perché molti enti governativi e aziende hanno cominciato ad usare chiavi con lunghezza minima di 2048 bits.

Capitolo 1

Crivello quadratico e fattorizzazione

Il crivello quadratico è un algoritmo per fattorizzazione di interi sviluppato da Carl Pomerance nel 1981, era l'algoritmo più veloce finché non è stato sorpassato nel 1993 dal crivello dei campi di numeri. Il crivello quadratico resta comunque il più veloce per fattorizzazioni di numeri inferiori alle 110 cifre. Quella che implementeremo è una versione parallela dell'algoritmo del crivello quadratico, seguendo lo stile Object Oriented del C++11.

1.1 Funzionamento

Sia N il numero da fattorizzare, il crivello quadratico cerca x e y tale che

$$x \not\equiv \pm y \mod n \quad \wedge \quad x^2 \equiv y^2 \mod n$$

Questo implica:

$$(x-y)(x+y) \equiv 0 \mod n$$

Quindi calcoliamo $(x-y, n)$ usando l'algoritmo di Euclide per vedere se ci sono divisori non banali. Sappiamo che la possibilità che sia un fattore non banale è $\frac{1}{2}$. Definiamo quindi:

$$Q(x) = (x + \lfloor \sqrt{n} \rfloor)^2 - n = \tilde{x}^2 - n$$

E calcoliamo $Q(x_1), Q(x_2), ..., Q(x_k)$. Dalla valutazione di $Q(x)$ vogliamo ottenere un sottoinsieme tale che $Q(x_{i_1})Q(x_{i_2})...Q(x_{i_r})$ sia un quadrato perfetto, y^2. Notando che $Q(x) \equiv \tilde{x}^2 \mod n$ quello che otteniamo è

$$Q(x_{i_1})Q(x_{i_2})...Q(x_{i_r}) \equiv (x_{i_1}x_{i_2}...x_{i_r})^2 \mod n$$

E se la condizione sopra vale, allora abbiamo fattori di N.

1.1.1 Base di fattori

Nel paragrafo precedente abbiamo detto che vogliamo calcolare $Q(x_1), Q(x_2)$, $..., Q(x_k)$ affinché sia possibile ottenere dal prodotto dei $Q(x_i)$ un sottoinsieme che sia un quadrato perfetto. Per ottenere questo risultato è necessario che gli esponenti dei fattori primi del prodotto siano pari, piccoli e fattorizzabili su in fissato insieme di numeri primi piccoli, che chiameremo **base di fattori**. Affinché $Q(x)$ sia piccolo, è necessario usare un x vicino a 0, fissiamo quindi un limite M e consideriamo soltanto i valori di x sull'intervallo di *sieving* $[-M, M]$. Se x si trova nell'intervallo, e se qualche primo p divide $Q(x)$, allora:

$$(x - \lfloor\sqrt{n}\rfloor)^2 \equiv n \mod p$$

Quindi n è un residuo quadratico $\mod p$. Di conseguenza i primi nella base di fattori devono avere **simbolo di Legendre** $= 1$

$$\left(\frac{n}{p}\right) = 1$$

Una seconda condizione per questi primi è che siano inferiori a un certo limite che chiameremo B, il quale dipende dalla grandezza del numero da fattorizzare N.

1.1.2 Sieving

Una volta che abbiamo i numeri primi della base di fattori, cominciamo a prendere gli x dall'intervallo di sieving, calcoliamo $Q(x)$ e controlliamo che fattorizzi completamente sulla base di fattori. Se fattorizza completamente, allora chiamiamo questo intero **smooth**, altrimenti verrà semplicemente

scartato e si procederà a valutare il successivo elemento nell'intervallo di sieving. Essendo l'algoritmo parallelo, l'intervallo di sieving verrà diviso in sotto intervalli, e ogni processore lavorerà in questi sotto intervalli. Se p è un fattore primo di $Q(x)$, allora $p \mid Q(x+p)$. Invece se $x \equiv y \mod p$, allora $Q(x) \equiv Q(y) \mod p$. Quindi per ogni primo p nella base di fattori, risolviamo

$$Q(x) = s^2 \equiv 0 \mod p, \quad x \in \mathbb{Z}_p$$

Otterremo due soluzioni, che chiameremo s_{1_p} e $s_{2_p} = p - s_{1_p}$. Allora gli $Q(x_i)$ con x_i nell'intervallo di sieving, sono divisibili per p quando $x_i = s_{1_p}, s_{2_p} + pk$ per qualche intero k.

Ci sono differenti modi per applicare il crivello partendo da questo punto, un modo è quello di prendere un sotto intervallo, dipendente dalla lunghezza della memoria, e inserire $Q(x_i)$ in un array per ogni x_i nel sotto intervallo. Per ogni p, partendo da s_{1_p} *e* s_{2_p} si divide la più grande potenza possibile di p per ogni elemento dell'array, salvando l'opportuna potenza $\mod 2$ di p in un vettore. Quello che si ottiene è un vettore contenente le potenze dei primi, il quale verrà inserito dentro una matrice A. Questo processo va iterato finché non si otterranno abbastanza elementi nella matrice A.

1.1.3 Costruzione della matrice

Se $Q(x)$ fattorizza completamente, allora inseriremo gli esponenti $\mod 2$ dei primi nella base di fattori, in un vettore come descritto sopra. Aggiungeremo tutti questi vettori in una matrice A, in modo che le righe rappresentino i $Q(x_i)$ e le colonne gli esponenti $\mod 2$ dei primi nella base di fattori. Ricordiamo che vogliamo un prodotto dei $Q(x_i)$ che dia un quadrato perfetto, quindi la somma degli esponenti di ogni fattore primo nella base di fattore deve essere pari, di conseguenza congruo a $0 \mod 2$. Quindi, dati $Q(x_1), Q(x_2), ..., Q(x_k)$ è necessario trovare le soluzioni di

$$Q(x_1)e_1 + Q(x_2)e_2 + ... + Q(x_k)e_k;$$

dove e_i sono o 0 o 1. Quindi se $\overrightarrow{a_i}$ è la riga di A corrispondente a $Q(x_i)$ allora

$$\overrightarrow{a_1}e_1 + \overrightarrow{a_2}e_2 + ... + \overrightarrow{a_k}e_k \equiv \overrightarrow{0} \mod 2$$

$$\overrightarrow{e}A = \overrightarrow{0} \mod 2$$

dove

$$\overrightarrow{e} = (e_1, e_2, ..., e_k)$$

Troviamo l'insieme delle soluzioni usando l'algoritmo di Gauss. Abbiamo bisogno di trovare tanti $Q(x_i)$ quanti sono i primi nella base di fattori. Ogni elemento dell'insieme delle soluzioni corrisponde ad un sottoinsieme di $Q(x_i)$ il cui prodotto è un quadrato perfetto. Se la base di fattori ha B elementi, e abbiamo $B + 10$ valori di $Q(x)$, allora avremo almeno una probabilità $\frac{1023}{1024}$ di trovare un opportuno fattore. Controlliamo quindi il vettore delle soluzioni per vedere se il corrispondente prodotto di $Q(x_i)$ e x_i produce un opportuno fattore di N calcolando il *GCD* tra i due. Altrimenti si controlla il successivo elemento nello spazio delle soluzioni, quando un opportuno fattore è stato trovato si fa il test di primalità.

1.2 Implementazione

Questa non è la prima implementazione del crivello quadratico e molto probabilmente non sarà l'ultima. In rete si trovato numerosi progetti di realizzazione dell'algoritmo. In questa versione, l'algoritmo è parallelo e si è cercato di rendere l'implementazione il più comprensibile e modulare possibile facendo uso del C++11 e del paradigma di programmazione orientato agli oggetti. Si potrebbe contestare questa implementazione dicendo che l'uso di classi astratte e il meccanismo di overriding riduca le prestazioni, ma nel caso dell'algoritmo del crivello quadratico questo overhead è talmente insignificante in confronto al resto dell'algoritmo che non viene considerato. I vantaggi invece sono maggior modularità e la possibilità di estendere l'algoritmo. Sono state inizialmente definite le strutture basi necessarie per il crivello quadratico, in particolare la classe *QSVector* che rappresenta vettori in $\mathbb{Z}$, la cui implementazione è basata sui *std::vector*, *QSMatrix* e *QSBitMatrix* che rappresentano rispettivamente matrici in $\mathbb{Z}$ e in $\mathbb{Z}_2$. Abbiamo inoltre le classi di "alto livello" *QSFactorBase* che contiene al suo interno un *QSVector* e rappresenta la base di fattori. Altra classe di "alto livello" è *QS* la quale

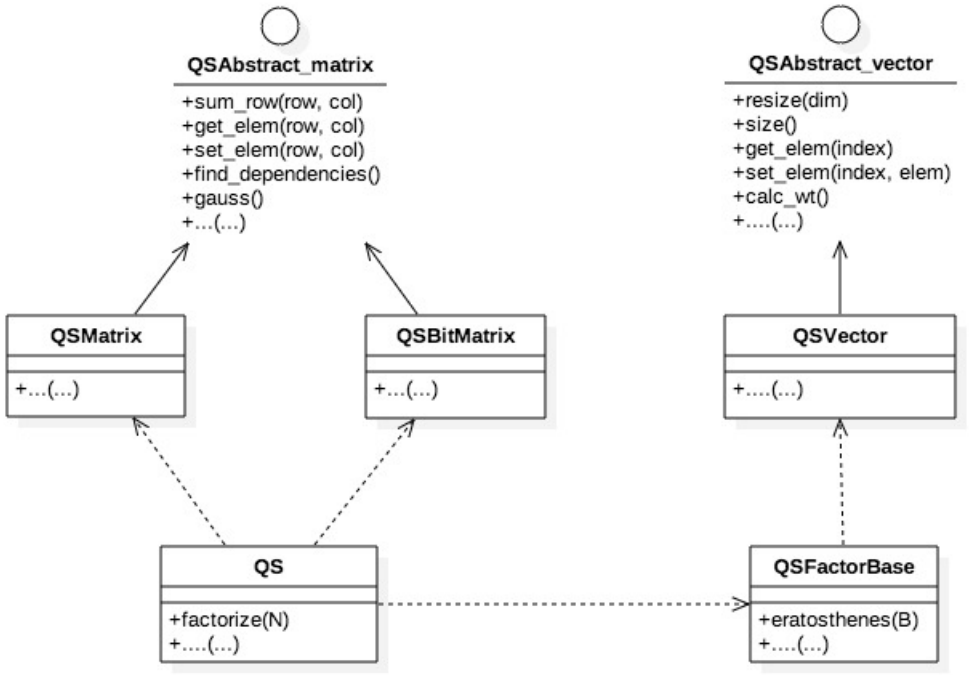

Figura 1.1: Struttura QS

contiene i metodi di gestione e configurazione dell'algoritmo. La struttura di QS è data dalla seguente figura

Per memorizzare i numeri usati nei vari oggetti è stato usato il tipo dato intero **mpz_t** della libreria **GMP**.

Capitolo 2

Backup

L'algoritmo del crivello quadratico è implementato in modo da essere parallelo, eseguibile su più macchine o su cluster. Nel caso dei cluster, questi offrono la possibilità di eseguire programmi per una durata limitata di tempo, dopo tale durata il processo viene fermato. Oppure in caso venga avviato in una sola macchina o su n macchine, rimane sempre il rischio che il programma smetta di funzionare. Questo è problematico, se il processo venisse fermato durante la normale esecuzione si rischierebbe di perdere i dati calcolati, rendendo il tempo speso nel calcolo tempo perso. C'è bisogno quindi di un sistema per salvare i dati calcolati per poi caricarli in un secondo momento.

2.1 Serializzazione

Una tecnica che ci viene in aiuto è la **serializzazione**. Questo processo permette di salvare un oggetto su un supporto fisico oppure di trasmetterlo attraverso la rete. È una tecnica ben nota in ambito di Web e viene usata quando si vuole trasmettere un oggetto nella rete tra due nodi. Esistono essenzialmente due formati per la serializzazione, **binario** e **testuale**. Il formato binario salva gli oggetti in modo non leggibile; di solito questo metodo viene preferito quando si vuole ridurre lo spazio usato per la serializzazione oppure quando si ha la necessità di avere una serializzazione performante, in quanto la serializzazione con formato binario viene completata con un numero di cicli di CPU inferiore. Il formato testo invece, salva gli oggetti in modo leggibile usando codifiche tipo JSON, XML o altro. La codifica di testo è ampiamente usata nel caso di applicazioni web con protocolli di comunicazione RESTful, dove gli oggetti vengono trasmessi sotto forma di stringhe. Un esempio di serializzazione di un oggetto *User* con codifica JSON è il seguente:

```
class User {
private:
    std::string name;
    std::string surname;
    unsigned int age;
    bool alive;
    std::vector<std::string> friends;

public:
    ....

    User() {
        name = "Ayoub";
        surname = "Ouarrak";
        age = 24;
        alive =  true;
        friends = {"Fizz", "Bizz"};
    }

    ....
};
```

Serializzando *User* usando una codifica JSON si ha:

```
{
    "name" : "Ayoub",
    "surname" : "Ouarrak",
    "age" : 24,
    "alive" : true,
    "friends" : [
        "Fizz",
        "Bizz"
    ]
}
```

Come si nota dall'esempio sopra, la serializzazione "cattura" soltanto lo stato dell'oggetto.

2.1.1 Supporto linguaggi di programmazione

Molti linguaggi orientati agli oggetti supportano nativamente la serializzazione, senza l'uso di librerie esterne. Tra questi linguaggi ricordiamo Ruby, Python, Objective-C, Java, e la famiglia .NET. Di seguito un breve esempio semplificato in Java:

```
public class Employee implements Serializable {
    .....
}

public class SerializationUtil {
    // deserialize to Object from given file
    public static Object deserialize(String fileName) {
        .....
    }

    // serialize the given object and save it to file
    public static void serialize(Object obj, String fileName) {
        .....
    }
}

public class SerializationTest {

    public static void main(String[] args) {
        // file name that contain the serialization
        String fileName = "employee.ser";
        Employee emp = new Employee();

        // insert data in emp
        ....

        // serialize to file
        SerializationUtil.serialize(emp, fileName);


        Employee empNew = null;

        // deserialize Object from file and cast to Employee
        empNew = (Employee) SerializationUtil.deserialize(
    fileName);
    }
}
```

L'esempio Java riportato sopra, mostra lo "schema" generale usato per la serializzazione in linguaggi che supportano il paradigma orientato agli oggetti.

- Si estende un'interfaccia **Serializable**
- Si definiscono i metodi **serialize** e **deserialize** che contengono il codice per la serializzazione dell'oggetto

- Si chiama il metodo **serialize** (o **deserialize**) nei momenti in cui si vuole effettivamente salvare lo stato dell'oggetto

Il linguaggio C++ non supporta la serializzazione, ma varie librerie sono presenti in rete per soddisfare questa esigenza, ognuna con una sintassi differente e con pregi e difetti; di seguito daremo una breve panoramica delle librerie che sono state prese in considerazione.

2.1.2 Boost

Boost è una celebre libreria C++ che offre tantissime funzionalità tra qui la serializzazione, il cui funzionamento è simile a quello visto in java. La serializzazione di Boost è portatile, è compatibile con i contenitori della STL e estremamente efficiente. Una panoramica della sintassi è data dal seguente esempio:

```
// class to serialize
class gps_position {
private:
    int degrees;
    int minutes;
    float seconds;

    friend class boost::serialization::access;

    template<class Archive>
    void serialize(Archive & ar) {
        ar & degrees;
        ar & minutes;
        ar & seconds;
    }

public:
    gps_position() {};
    gps_position(int d, int m, float s) :
          degrees(d), minutes(m), seconds(s) {}
};

....

int main() {
    // create and open a character archive for output
    std::ofstream ofs("filename");

    // create class instance
    const gps_position g(35, 59, 24.567f);
```

```
    // save data to archive
    boost::archive::text_oarchive oa(ofs);

    oa << g;

    ......

    // ... some time later restore the class instance to its
    orginal state
    gps_position newg;

    std::ifstream ifs("filename");
    boost::archive::text_iarchive ia(ifs);

    ia >> newg;

    return 0;
}
```

Boost, a differenza di Java, non offre la possibilità di estendere un'interfaccia *Serializable*, ma fa uso di una classe friend

```
friend class boost::serialization::access;
```

per permettere l'accesso ai dati interni della classe in modo da essere usati nel metodo

```
template<class Archive>
void serialize(Archive & ar) {
    .....
}
```

Inoltre la gestione dei file è "nascosta" dall'uso di **Archive**, idea che verrà ripresa quando introdurremo la libreria *Kairos*. La serializzazione di Boost non è stata usata per risolvere il problema di backup nell'algoritmo del crivello, perché come abbiamo detto Boost è una collezione di librerie molto grande, usarla equivale a introdurre una dipendenza ad una libreria esterna e non sappiamo se la macchina che eseguirà il crivello quadratico è compatibile con Boost.

2.1.3 Cereal

Altra libreria per la serializzazione in C++, forse meno nota di Boost, è **Cereal**. Cereal è compatibile con i vari tipi scalari e con i tipi della STL, usa codifiche in binario, JSON e XML. A differenza di Boost, Cereal non ha dipendenze esterne; un esempio semplificato d'uso è il seguente

```
struct MyRecord {
    uint8_t x, y;
  float z;

  template <class Archive>
  void serialize(Archive & ar) {
    ar(x, y, z);
  }
};

struct SomeData {
  int32_t id;
  std::shared_ptr<std::unordered_map<uint32_t, MyRecord>> data;

  template <class Archive>
  void save(Archive & ar) const {
    ar(data);
  }

  template <class Archive>
  void load(Archive & ar) {
    static int32_t idGen = 0;
    id = idGen++;
    ar(data);
  }
};

int main() {
  std::ofstream os("out.cereal", std::ios::binary);
  cereal::BinaryOutputArchive archive(os);

  SomeData myData;
  archive(myData);

  return 0;
}
```

Anche in questo caso, come Boost, non c'è nessuna estensione di un'interfaccia *Serializable*, anche Cereal fa uso della struttura *Archive* per "nascondere" la gestione dei file, e in generale il funzionamento è simile a quello di Boost, con il vantaggio di essere indipendente da librerie esterne. Vediamo dunque un'ultima libreria.

2.1.4 Autoserial

Autoserial è un'altra libreria C++ che permette la serializzazione di oggetti, ma a differenza di Boost e Cereal offre anche la reflection, funzionalità non

presente nel linguaggio C++. La reflection è una tecnica molto usata in Java e in C#. E' possibile per esempio creare in modo dinamico un'istanza di un tipo, associare il tipo a un oggetto esistente o ottenere il tipo da un oggetto esistente, nonché richiamarne i metodi o accedere ai campi e alle proprietà dell'oggetto. La reflection consente inoltre di accedere agli eventuali attributi utilizzati nel codice. Autoserial usa la reflection per serializzare e deseriallizare gli oggetti in C++, usando la seguente sintassi

```
class BasicObject : public autoserial::ISerializable {
    AS_CLASSDEF(BasicObject)
    AS_MEMBERS
        AS_PRIVATEITEM(double, a)
        AS_ARRAY(int, c, 32)
        AS_ITEM(std::vector<std::string>, strings)
        AS_ITEM(std::map<int, std::set<std::string>>, myMap)
    AS_CLASSEND;

    // Members outside macros will not be serialized
    float f;
};
```

L'esempio sopra mostra la poca eleganza sintattica di Autoserial, la quale fa ampio uso di macro, e la poca flessibilità in quanto richiede che la classe cominci sempre con **AS_CLASSDEF**.

Nel prossimo capitolo introdurremo **Kairos**, libreria C++ cross platform per la serializzazione sviluppata appositamente per essere usata nell'algoritmo del crivello quadratico.

Capitolo 3

Kairos

Kairos riprende molte idee dalle librerie viste nel capitolo precedente, in particolare verrà ripreso il concetto di **Archive** per nascondere la gestione e la scrittura su file, verrà estesa un'interfaccia **Serializable** e si implementeranno i metodi **serialize** e **deserialize**. In generale l'obbiettivo di Kairos è offrire il supporto alla serializzazione in C++, usando lo standard C++11, in modo semplice e pulito, senza l'uso di sintassi complesse e senza l'uso di librerie esterne che aggiungano dipendenze in più; è inoltre estendibile ed è possibile aggiungere altre codifiche di serializzazione alla libreria. Un breve esempio di utilizzo è dato dal seguente frammento di codice

```
class Object : public Serializable, public Serialization {
private:
    int data;
    float data1;
    char data2;

public:
    // a default constructor is necessary for deserialization
    Object() {
        // register Object, by default use text format
        registerType(this);

        // if you want to use binary format
        // registerType(this, Serialization::BINARY);
    }

    void serialize(Archive& archive) {
        archive << data << data1 << data2;
    }

    void deserialize(Archive& archive) {
        archive >> data >> data1 >> data2;
```

```
    }

};
```

Come per la serializzazione in java, anche in Kairos bisogna implementare l'interfaccia **Serializable** e estendere la classe **Serialization** e successivamente implementare i due metodi *serialize* e *deserialize* usando l'archivio che viene passato per parametro. A differenza di Boost e Cereal, i metodi non sono templatici e l'archivio si occupa di salvare i dati usando la codifica scelta in fase di registrazione del tipo. Per salvare effettivamente lo stato dell'oggetto Object, ci basta richiamare

```
....

 Object* object1 = new Object(0, 2.3, 'a');

 // createCheckpoint can throw a SerializationException*
 try {
    Serialization::createCheckpoint(object1);

 } catch (SerializationException* exp) {
    std::cout << exp->what() << std::endl;
 }

....
```

Il metodo createCheckpoint è un metodo statico della classe Serialization che salva sul supporto fisico la serializzazione di object usando il giusto formato. Il recupero dell'oggetto funziona in modo analogo, la classe Serialization si occupa di localizzare il file relativo alla serializzazione e crea l'oggetto caricando i dati letti dal file e ritorna l'oggetto

```
....
try {
    // return a map containing all objects with type "Object"
    auto objects = Serialization::restore<Object>();

    // get the instance of "object1"
    objects.at("object1")->get();

} catch(SerializationException* exp){
    std::cout << exp->what() << std::endl;
}
.....
```

L'esempio sopra mostra l'uso del metodo templatico restore, il quale deserializza tutti gli oggetti che hanno tipo compatibile con l'istanza del template. Il diagramma seguente mostra la struttura base di serializzazione in Kairos

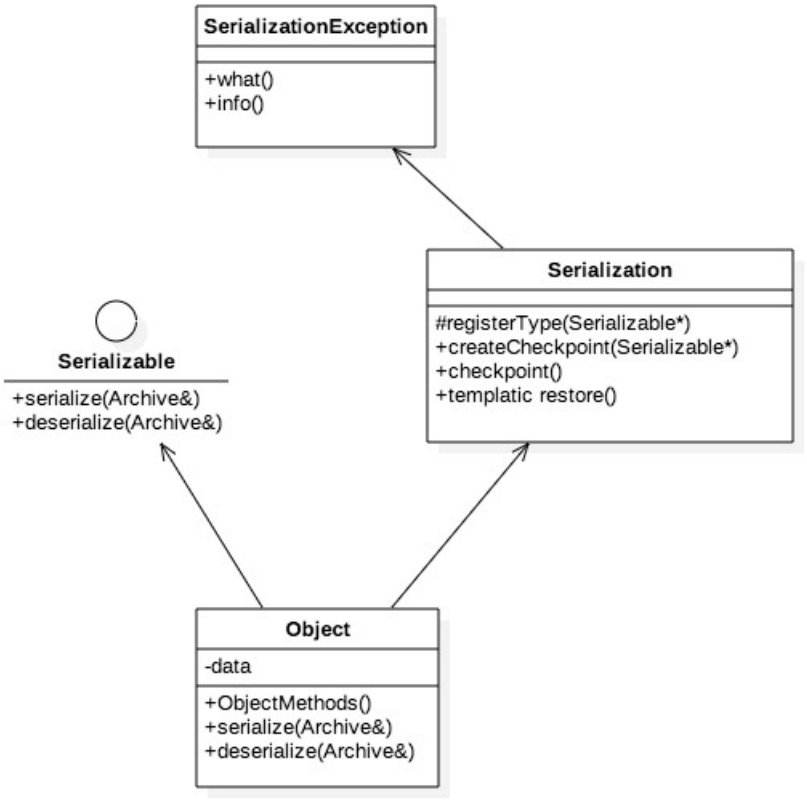

Figura 3.1: Struttura Serializable e Serialization

Dalla figura riportata sopra vediamo la struttura di un oggetto generico *Object* pronto per la serializzazione, il quale estende l'interfaccia *Serializable*, dotata di due metodi virtuali puri che vanno implementati da tutte le classi che estendono *Serializable*.

```
virtual void serialize(Archive&) = 0;
virtual void deserialize(Archive&) = 0;
```

L'estensione di **Serializable** offre anche maggior leggibilità del codice. Altra classe che viene estesa è *Serialization* la quale contiene i vari metodi che permettono la registrazione del tipo, la scelta della codifica di serializzazione e i metodi per l'avvio del processo di serializzazione.

```
static void registerType(Serializable*);
```

Il metodo *registerType* viene richiamato nel costruttore delle classi serializzabili, e genera un **id** univoco il quale viene aggiunto ad una mappa

interna a *Serialization*. È disponibile anche un altro metodo *registerType* in overloading

```
static void registerType(Serializable*, std::string);
```

Il secondo parametro permette la scelta della codifica di serializzazione che può essere

- Serialization::BINARY
- serialization::TEXT

3.1 Checkpoint

Le librerie che abbiamo visto nel capitolo precedente offrono un sistema di serializzazione basilare, non viene offerta nessuna possibilità di tenere traccia di serializzazioni passate. Quello che stiamo suggerendo è di avere uno *storico delle serializzazioni*. Con storico si intende una cronologia delle serializzazioni fatte; questo può essere utile, per esempio, se si vuole ripristinare un oggetto ad uno stato arbitrario assunto in passato, oppure può essere usato come sistema di debug. Kairos offre questa funzionalità mediante l'uso di **checkpoint**

```
....
 Object* object1 = new Object(0, 2.3, 'a');

 // createCheckpoint can throw a SerializationException*
 try {
    Serialization::createCheckpoint(object1);
....
```

La chiamata a *createCheckpoint* provoca la creazione di un nuovo file contenente la serializzazione dell'oggetto passato per parametro; il file ha la seguente struttura:

$$archive.[codifica].[classe].[idistanza]$$

nel caso di object1 sarà di questa forma:

$$archive.text.Object.1234$$

La classe *Serialization* offre anche un altro metodo per il checkpoint senza parametro:

```
....
    Serialization::checkpoint();
....
```

In questo caso viene fatto il checkpoint di tutti gli oggetti serializzabili ognuno con un suo file di serializzazione. La chiamata *Serialization::restore<Object>()* recupera l'ultimo checkpoint fatto degli oggetti con tipo Object.

3.2 Archive

La semplicità d'uso di Kairos risiede nell'uso degli **Archive**, idea presa dalla libreria Boost e Cereal. Gli Archive sono un'astrazione per memorizzare dati in codifiche differenti, per codifiche di testo e binario, l'implementazione è basata sull'uso del file del C++ come supporto di archiviazione. La struttura di Archive essendo estendibile, permette l'utilizzo di altri tipi di supporto per l'archiviazione

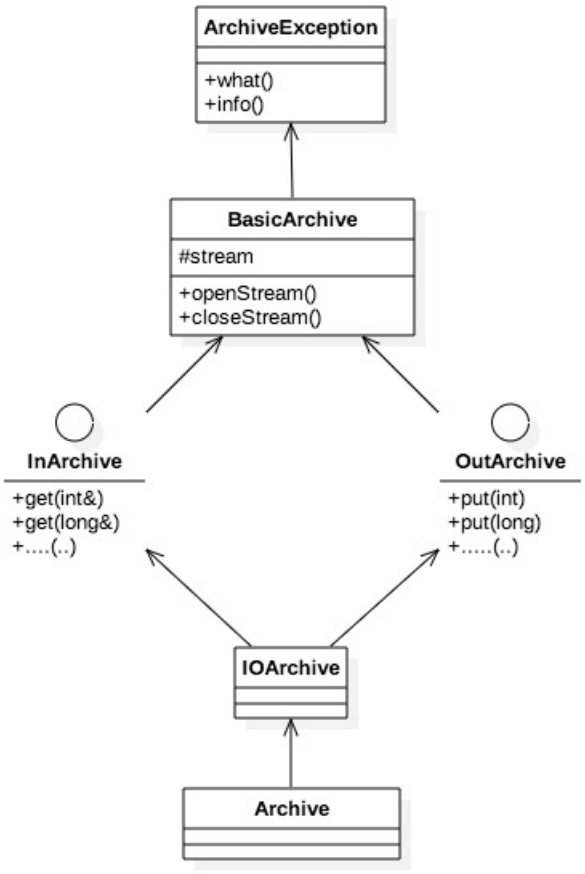

Figura 3.2: Struttura Archive

La figura mostra le relazioni tra **BasicArchive**, le due interfacce **InArchive**, **OutArchive** e la classe di alto livello **Archive**, e da come si nota Archive non è utilizzabile in quanto estende le interfacce **InArchive**, **OutArchive** ma non implementa i metodi puri

```
....
void put(int src);
void put(long src);
void put(double src);
void put(float dest);
...

void get(int& dest);
void get(long& dest);
void get(double& dest);
void get(float& dest);
....
```

Questi metodi vengono implementati dalle classi che definiscono la codifica di serializzazione, in quanto per ogni codifica i metodi *put* e *get* hanno un comportamento differente

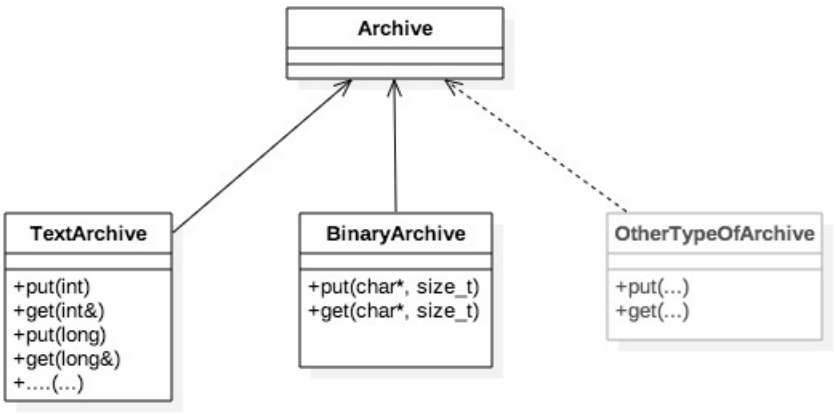

Figura 3.3: Struttura codifiche di serializzazioni

Le classi di codifica (Text, Binary, other) oltre a implementare i metodi puri, offrono un overloading degli operatore << e >> rendendo la serializzazione sintatticamente "apprezzabile". Altro vantaggio di strutturare Archive in questo modo è quello di nascondere l'uso diretto delle classi di codifiche (Text, Binary, ...). Di seguito un esempio d'utilizzo riprendendo la classe User vista nel capitolo precedente

```
class User : public Serializable, public Serialization {
private:
    std::string name;
    std::string surname;
    unsigned int age;
    bool alive;
    std::vector<std::string> friends;

    ....

public:
    User() {
        .....
        registerType(this, Serialization::TEXT);
    }

    void serialize(Archive& archive) {
        archive << name
                << surname
                << age
                << alive
                << friends;
    }

    void deserialize(Archive& archive) {
        archive >> name
                >> surname
                >> age
                >> alive
                >> friends;
    }

    .....
};
```

L'esempio sopra mostra la semplicità d'uso di Kairos, la scelta dell'archivio viene delegata alla classe *Serialization*, la quale oltre a gestire la scelta dell'archivio si occupa anche di aggiornare il file **serialization.index** contente la tabella degli oggetti serializzati e il file contenente la serializzazione. Nel caso di checkpoint, la classe *Serialization* si comporta nel seguente modo

```
void Serialization::createCheckpoint(Serializable* object) {

    std::string format = "";

    try {
        format = ObjectsFormatSerialization.at(object);
```

```
        if(format == Serialization::TEXT) {
            TextArchive archive;
            object->serialize(archive);
        }

        if(format == Serialization::BINARY) {
            BinaryArchive archive;
            object->serialize(archive);
        }

    } catch (std::out_of_range exp) {
        throw new SerializationException("object not registered")
;

    } catch (SerializationException* exp) {
        throw exp;
    }

    .....
```

Quello che fa il metodo *createCheckpoint* è ritrovare dalle mappe interne il formato di codifica scelto e in base al formato creare il tipo di archivio opportuno e richiamare il metodo *serialize* passando l'archivio. Se l'oggetto da serializzare non ha chiamato il metodo *registerType*, la ricerca dentro la mappa provoca un'eccezione di tipo *std::out_of_range* la quale viene catturata e viene lanciata una nuova eccezione di tipo *SerializationException* dal messaggio comprensibile.

3.2.1 Serializzazione tipi scalari

Come abbiamo detto inizialmente, la serializzazione è un processo che dipende dal tipo che si vuole serializzare. Kairos attualmente supporta la serializzazione di tipi scalari, array, matrici e oggetti Serializable. I tipi scalari vengono semplicemente serializzati inserendo il loro valore nel file di serializzazione separati da spazio, differente il discorso è per i tipi **float** e **double**. Per garantire una serializzazione portatile dei tipi *floating point* è stata usata una tecnica di "incapsulamento". I valori float e double non vengono serializzati direttamente su file, ma vengono prima incapsulati usando lo standard IEEE745 in un formato portatile, che nel caso dei float è **uint32** mentre per i double è **uint64**, e si serializza invece i nuovi valori ottenuti. Per la deserializzazione si usa il processo inverso, si deserializzano i tipi uint32 e uint64 e da questi si ricavano i valori originali float o double.

3.2.2 Serializzazione array

La serializzazione degli array è molto intuitiva; si scrive inizialmente la dimensione dell'array e nella riga successiva si comincia a serializzare i valori contenuti dentro l'array. Il processo di deserializzazione funziona allo stesso modo, si legge la dimensione, si crea un array allocando la giusta dimensione e si caricano i valori letti.

3.2.3 Serializzazione matrici

La serializzazione di matrici è un processo che richiede tempo ed è costoso, sopratutto nel caso del crivello quadratico dove le matrici usate sono molto grandi e *abbastanza* sparse, serializzare le matrici così come sono genererebbe inutilmente file molto grandi, e deserializzare questi file richiederebbe molto tempo. Per cercare di ottimizzare la serializzazioni di matrici si fanno dei controlli preliminari, in particolare si controlla la sparsità della matrice prima di serializzarla usando la seguente euristica

$$\frac{numberofelement(A) - nonzero(A)}{numberofelement(A)} > threshold$$

se si supera una certa soglia (di default 0.75) allora la matrice si considera sparsa, di conseguenza l'algoritmo serializza la dimensione della matrice, i valori non nulli e la loro posizione nella matrice.

$$\begin{array}{ccc} rows & cols & \\ x1 & y1 & value1 \\ x2 & y2 & value2 \\ \ldots & \ldots & \ldots \end{array}$$

La deserializzazione funziona in modo analogo, si crea una matrice nulla della stessa dimensione letta dal file $[rows, cols]$, si legge dal file i valori e la loro posizione x, y e si inseriscono nella matrice appena creata, riducendo in questo modo il tempo di caricamento.

3.2.4 Serializzazione di oggetti annidati

L'algoritmo del crivello quadratico è fatto in modo da seguire il paradigma Object Oriented; questo significa che i concetti fondamentali dell'algoritmo sono visti come oggetti. Ad alto livello abbiamo *QS* che ha il compito di

gestire tutta la logica dell'algoritmo, contiene inoltre *QSBaseFactor* che rappresenta l'astrazione della base di fattori. Identifichiamo quindi gli oggetti del crivello quadratico e in che modo sono relazionati tra loro in modo da usare la politica di serializzazione più adatta. La struttura del crivello quadratico è la seguente

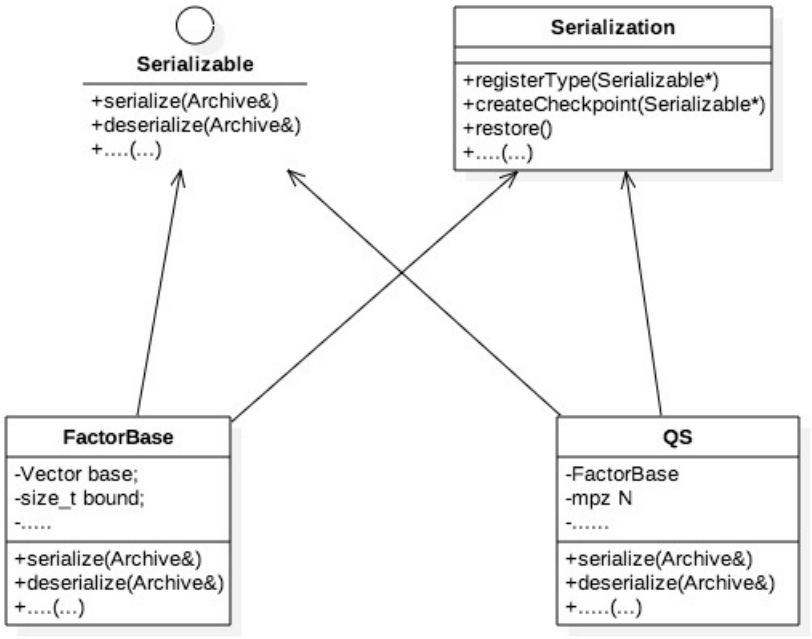

Figura 3.4: Serializzazione crivello quadratico

Traducibile nel seguente frammento di codice

```
class QSFactorBase : public Serializable, public Serialization {
private:
    std::vector<long long> base;
    std::size_t bound;
    .....

public:
    QSFactorBase() {
        .....
        registerType(this, Serialization::BINARY);
    }

    ....
```

```
    void serialize(Archive& archive) {
        archive << base << bound << ....;
    }

    void deserialize(Archive& archive) {
        archive >> base >> bound >> ....;
    }

    .....
};
```

Per la base di fattori si sceglie una serializzazione con codifica binaria (essendo più performante).

```
class QS : public Serializable, public Serialization {
private:
    QSFactorBase factorBase;
    mpz N;
    .....

public:
    QS() {
        .....
        registerType(this, Serialization::BINARY);
    }

    ....

    void serialize(Archive& archive) {
        archive << factorBase << N << ....;
    }

    void deserialize(Archive& archive) {
        archive >> factorBase >> N >> ....;
    }

    .....
};
```

In questo caso l'oggetto *QS* contiene un oggetto Serializable, quindi durante la serializzazione di *QS* viene chiamato il metodo serialize di *QSFactorBase* il quale serializza la base di fattore e gli altri dati in *QSFactorBase*, serializza N e eventualmente altri dati di *QS*.

```
QS* qs = new QS();
....

try {
```

```
    serialization::createCheckpoint(qs);

} catch(SerializationException* exp) {
    // something wrong happened during qs serialization
}
....
```

La chiamata a *createCheckpoint(qs)* provoca quindi una **serializzazione a catena** di qs e di tutti gli oggetti serializable che contiene, nel nostro caso di *FactorBase*. Il restore funziona in modo inverso

```
....
try {
     auto qs = Serialization::restore<QS>();

} catch (SerializationException* exp) {
    // whops!, something wrong happen, try to restore another
    // serialization of QS from the "Serialization History"
}
....
```

Si cerca di deserializzare partendo da QS che chiama a sua volta il metodo deserialize di FactorBase.

Capitolo 4

Benchmark

Durante la progettazione dell'algoritmo del crivello quadratico è nata la necessità di monitorare l'uso delle risorse da parte dell'algoritmo, in modo da individuare i punti in cui è possibile ottimizzare. È necessario quindi un sistema di **benchmark** per il quale è stata sviluppata una libreria per permettere la raccolta di informazioni sull'uso delle risorse e la successiva rappresentazione grafica tramite tecnologie Web.

4.1 cMark

cMark è una libreria sviluppata per permettere di monitorare l'uso delle risorse fisiche. La libreria è composta da due parti

- Raccolta informazioni e caricamento su DB
- Lettura informazioni dal DB e rappresentazione grafica dei dati

4.1.1 Raccolta informazioni

La fase iniziale consiste nella raccolta delle informazioni di sistema, che possono essere l'uso della memoria fisica, l'uso della CPU, l'uso del disco, ecc. Questa fase consiste in un *thread* sempre attivo in un ciclo teoricamente infinito, il quale dopo un tempo t (configurabile) chiama alcune routine di sistema per ottenere le informazioni sull'uso delle risorse; una volta ottenute le inserisce dentro un database SQLite. Oltre a questi dati viene aggiunto anche l'istante in cui è stata effettuata la misurazione. Le routine di sistema sono platform dependent, quindi è stata sviluppata un'interfaccia *DeviceInfo* la quale viene implementata in modo diverso in base all'architettura.

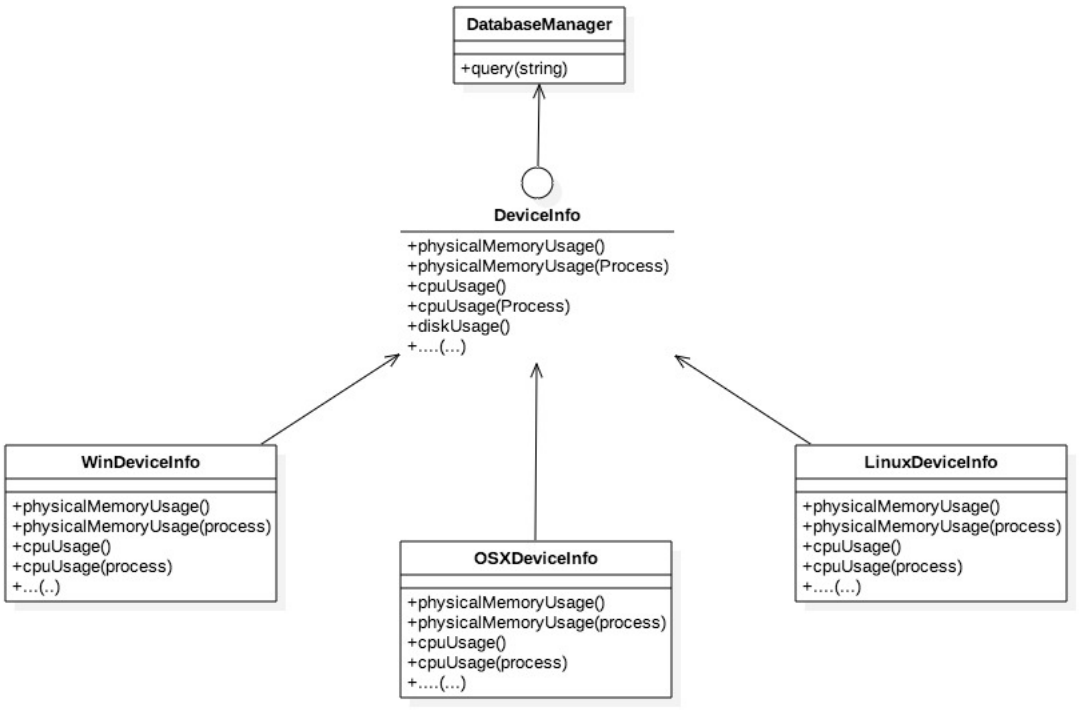

Figura 4.1: Struttura device info

L'interfaccia *DeviceInfo* fa uso di un oggetto *DatabaseManager* per interfacciarsi al database SQLite, mentre le varie implementazioni, diverse per ogni piattaforma, implementano i metodi puri offerti dalla interfaccia.

4.1.2 Rappresentazione dei dati

Salvare le informazioni sull'uso delle risorse in un database ha notevoli vantaggi, uno dei quali è la separazione tra la raccolta delle informazioni e la loro rappresentazione. Altro vantaggio è la possibilità di usare tecnologie differenti, ed è quello che è stato fatto. Per la fase di raccolta dei dati viene usato il linguaggio C++11, mentre per la parte di rappresentazione dei dati vengono usate tecnologie Web. In particolare viene usato uno script javascript per collegarsi al database SQLite e per il caricamento dei dati in vettori interni, i quali vengono passati alla libreria **Char.js** che si occupa di mostrare i grafici. Vengono usati inoltre HTML/CSS per il design del layout della pagina web contenente i grafici, il tutto impacchettato con il framework *electron* che permette di racchiudere le pagine web creando applicazioni native. Di seguito un esempio di grafico creato con *Chart.js*

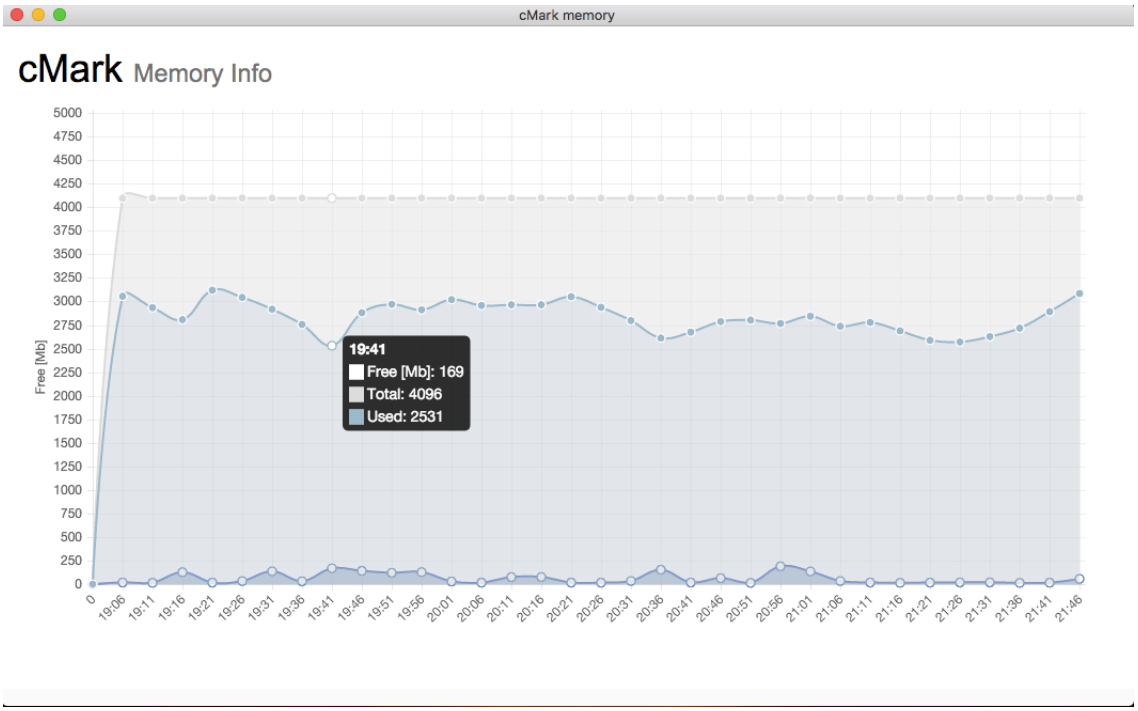

Figura 4.2: Grafico sull'uso della memoria

Come è stato detto, dividere la fase di raccolta dai dati dalla loro rappresentazione offre notevoli vantaggi. In questo modo è possibile eseguire il sistema di benchmark sul cluster in cui l'algoritmo del crivello quadratico è in esecuzione, e nello stesso momento su un macchina esterna monitorare i dati raccolti tramite un web browser, oppure tramite l'applicazione nativa "impacchetta" con *electron*.

Capitolo 5

Comunicazione di rete e parallelizzazione

Come è stato detto, questa versione del crivello quadratico è parallela. Il parallelismo è sia multi-core che multi-CPU, e quando si considera una versione parallela di un algoritmo c'è da considerare la complessità temporale delle varie parti dell'algoritmo. Nel caso del crivello quadratico la parte che richiede più tempo è la parte di *sieving*. Questa parte dell'algoritmo è ideale per la parallelizzazione in quanto ogni singolo sotto intervallo viene passato ad un processo, mentre un processo *master* si occupa di gestire il lavoro degli *slave* e di costruire la matrice A. Altra parte costosa dell'algoritmo è *l'eliminazione Gaussiana*, la quale confrontata con la parte di sieving rimane comunque di inferiore complessità computazionale.

5.1 MPI

La parallelizzazione è ottenuta tramite la libreria **OpenMPI**, che implementa lo standard Message Passing Interface. Lo scambio di messaggi può avvenire sia fra processi sulla stessa macchina (tipicamente tramite memoria condivisa), che fra processi su macchine differenti (tramite protocolli di rete come TCP/IP). Un grosso vantaggio di MPI è l'indipendenza dalla configurazione utilizzata. La libreria open source usata è in grado di gestire la comunicazione fra processi con diverse tecnologie, fra cui TCP per la comunicazione fra processi in esecuzione su macchine facenti parte di una rete LAN o WAN, e memoria condivisa per la comunicazione (molto più rapida) fra processi in esecuzione sulla stessa macchina. L'esecuzione su macchine multiple è gestita tramite il protocollo SSH: la macchina su cui viene eseguito il processo contatterà le altre tramite protocollo SSH, e lancerà opportunamen-

te il programma richiesto. La libreria funziona assumendo che le macchine abbiano un utente con lo stesso nome e con lo stesso contenuto della home (tenterà infatti di contattare tramite SSH la macchina con lo stesso nome utente da cui è stato lanciato il processo, e cercherà l'eseguibile nello stesso path della macchina di origine). Al fine di evitare possibili problemi in questo senso è opportuno utilizzare uno dei tanti sistemi in grado di sincronizzare il contenuto dei dischi, o se possibile montare la home su un file system di rete.

5.2 Parallelizzazione

Il programma parte suddividendo in più processi in base all'argomento passato come argomento, ed ogni nodo riceve un solo processo. Tutta la comunicazione avviene nell'ambiente di default **MPI_COMM_WORLD** che contiene tutti i processi; ad ogni processo viene assegnato un *rank* (identificatore numerico) e il processo con *rank* = 0 viene considerato il master, i processi con rank diverso da 0 diventano *slave*.

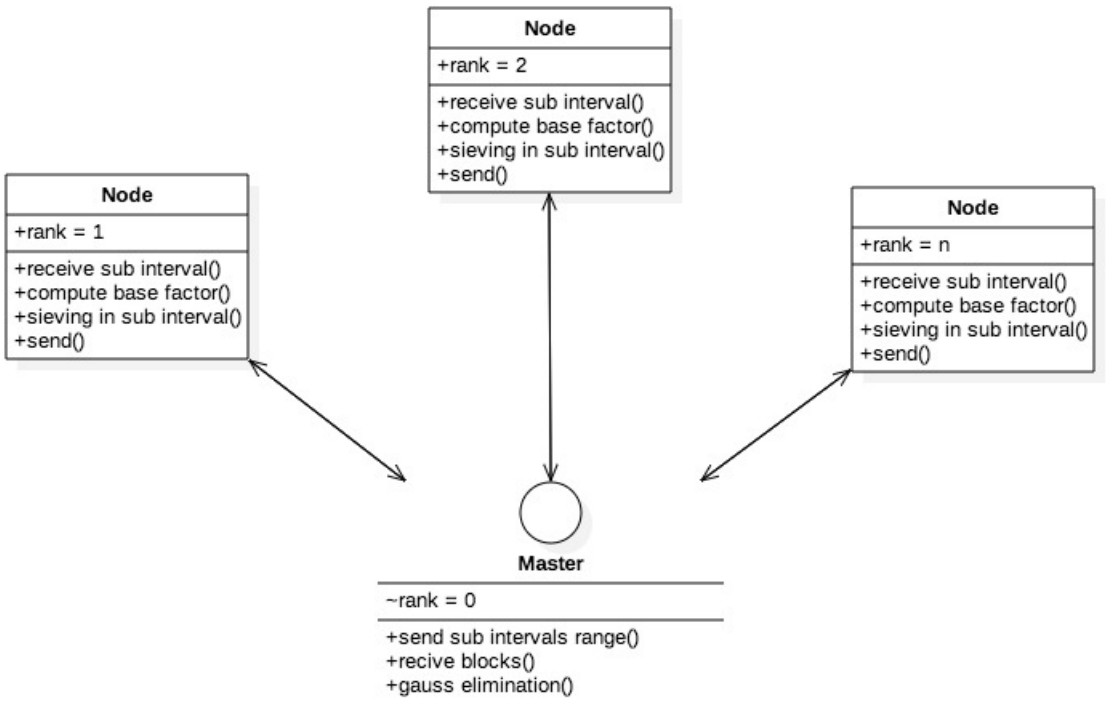

Figura 5.1: Grafico sull'uso della memoria

In codice la parte di inizializzazione si traduce nel seguente frammento di codice

```
....
/* 1st: launch the MPI processes on each node */
MPI_Init(&argc, &argv);

/* 2nd: request a thread id, sometimes called a "rank" from
        the MPI master process, which has rank or tid == 0
*/
MPI_Comm_rank(MPI_COMM_WORLD, &tid);

/* 3rd: this is often useful, get the number of threads
        or processes launched by MPI, this should be NCPUs-1
*/
MPI_Comm_size(MPI_COMM_WORLD, &nthreads);
.....
```

Successivamente viene calcolata la base di fattori, la quale è necessaria a tutti gli slave. Visto che la sua computazione è inferiore al tempo complessivo, viene fatta calcolare da ogni slave. Durante la fase di sieving, il master entra nella modalità "ricezione", mentre gli slave nella modalità "computazione-invio". Ogni nodo, in modo parallelo, si occuperà di calcolare le fattorizzazioni e inviarle al master tramite il metodo **MPI_Send**. Il master nella modalità di ricezione, chiama il metodo **MPI_Recv** per raccogliere i dati calcolati dagli slave e li serializza creando un checkpoint. Il master rimane in questo ciclo finché non riceve abbastanza $Q(x_i)$ quanti sono gli elementi nella base di fattori, dopo di che invia agli slave il segnale di fermare la computazione. A questo punto Il master avvia l'eliminazione di Gauss e viene fatta un'altra serializzazione.

Conclusioni

Concludendo, abbiamo visto una panoramica sul funzionamento del crivello quadratico, la cui dettagliata implementazione verrà trattata in un secondo lavoro di tesi. Abbiamo discusso del problema di eseguire l'algoritmo del crivello quadratico in un contesto in cui ci sia un limite di tempo di calcolo, con conseguente perdita dei dati calcolati e come è possibile risolvere questo problema.

Abbiamo discusso della serializzazione come possibile tecnica per salvare i dati. Sono state esposte alcune librerie per la serializzazione dei dati. Successivamente è stata introdotta la libreria **Kairos**, la quale permette di serializzare e deserializzare oggetti in C++, facendo vedere la struttura della libreria, le componenti fondamentali e il funzionamento con alcuni esempi. E' lecito chiedersi se era necessario scrivere una nuova libreria. Di seguito diamo alcune personali motivazioni:

- A differenza delle altre librerie viste, Kairos offre la serializzazione di matrici facendo controlli sulla sparsità
- Viene usato il concetto di checkpoint per offrire quello che abbiamo definito nei capitoli precedenti come storico delle serializzazioni
- La libreria è fatta in modo da essere facilmente estendibile
- La sintassi è di facile comprensione

Altro motivo è il fatto di aver compreso le varie tecniche di serializzazioni grazie alla realizzazione di questa libreria.

E' stato discusso un possibile metodo per fare benchmark del crivello, per questo intento è stata introdotta la libreria **cMark** la quale permette di monitorare l'uso della memoria e della cpu, mostrando successivamente come vengono rappresentati i dati raccolti con semplici grafici. Abbiamo discusso del vantaggio di usare tecnologie web moderne per rappresentare i dati raccolti.

Infine è stata data una breve panoramica di come avviene la comunicazione in rete tra il master dell'algoritmo e i vari slave.

5.3 Estensioni

Le due librerie svillupate, **Kairos** per la serializzazione e **cMark** per il benchmark, sono ancora in una fase prematura. Sono stati fatti alcuni test per verificare il corretto funzionamento ma rimangono ancora molte funzionalità da aggiungere, in particolare per **Kairos** sono in programma da aggiungere le seguenti funzionalità

- Aggiungere la compatibilità di serializzazione con altri tipo della STL
- Aggiungere altri formati per gli archivi, in particolare formati di tipo JSON, e archivi che salvano direttamente in un DB
- Aggiungere un sistema per monitorare quando un oggetto è stato modificato in modo da eseguire il checkpoint in automatico
- Serializzare oggetti in caso di dipendenza ciclica
- Aggiungere la serializzazione di oggetti non di tipi Serializable

Mentre per la libreria **cMark**, in futuro verranno sviluppate le seguenti funzionalità:

- Aggiungere il monitoraggio dell'uso della rete
- Aggiungere la possibilità di localizzare il db nella rete

Bibliografia

[1] Manuale di crittografia. Teoria, algoritmi e protocolli
http://www.amazon.it/Manuale-crittografia-Teoria-algoritmi-protocolli/dp/8820366908/ref=sr_1_1?s=books&ie=UTF8&qid=1461097908&sr=1-1

[2] Prime Numbers. A Computational Perspective
http://www.springer.com/gp/book/9780387252827

[3] Smooth numbers and the quadratic sieve https://math.dartmouth.edu/~carlp/PDF/qs08.pdf

[4] The Quadratic Sieve Factoring Algorithm
http://www.cs.virginia.edu/crab/QFS_Simple.pdf

[5] Factoring large integers using parallel Quadratic Sieve
https://www.nada.kth.se/~joel/qs.pdf

[6] Beej's Guide to Network Programming
http://beej.us/guide/bgnet/output/html/singlepage/bgnet.html

[7] Reflection support by means of template metaprogramming
http://lcgapp.cern.ch/project/architecture/ReflectionPaper.pdf

[8] ISO C++
https://isocpp.org/wiki/faq/serialization

[9] Object Oriented Design
http://www.oodesign.com

[10] cereal - A C++ library for serialization
http://uscilab.github.io/cereal/

[11] Autoserial library
http://home.gna.org/autoserial/

[12] Boost serialization
http://www.boost.org/doc/libs/1_37_0/libs/serialization/doc/index.html

[13] Chart.js
https://nnnick.github.io/Chart.js/docs-v2/

[14] Electron
http://electron.atom.io/docs/latest/tutorial/quick-start/

[15] MPI
http://www.linux-mag.com/id/5759/

[16] Bootstrap3 Framework
http://getbootstrap.com