Indice

1 Introduzione e obiettivi **1**

2 Crivello Quadratico **3**

3 Architettura del sistema di calcolo distribuito **5**
3.1 MPI 5

4 Tecnologie utilizzate **6**
4.1 OpenMPI 6

5 Strategia di parallelizzazione dell'algoritmo **7**

6 Algoritmo parallelizzato **8**

7 Risultati **15**

8 Sviluppi futuri **16**

Elenco delle figure

Elenco delle tabelle

1 Introduzione e obiettivi

Per il teorema fondamentale dell'Arimetica dato un numero N non primo, il quale possiede quindi dei divisori non banali, esiste ed è unica, prescindendo dall'ordine dei fattori, la sua fattorizzazione esprimibile come prodotto di numeri primi elevati ad opportune potenze. Il problema della fattorizzazione di numeri interi viene affrontato sin dalle elementari per trovare relazioni tra due numeri quali il massimo comun divisore o il minimo comune multiplo, ognuno di noi ha quindi presente di cosa si tratti, per lo meno ad un livello intuitivo. Quando ci si addentra nell'ostico compito di fattorizzare numeri che contengono un numero di cifre nell'ordine delle centinaia il problema, però, si dimostra essere molto più difficile di quanto si potesse pensare analizzandolo in maniera intuitiva. Sotto l'aspetto della teoria della complessità computazionale il problema risulta essere esponenziale, subesponenziale nel caso di alcuni algoritmi particolari. Sulla difficoltà di questo problema si basa un famosissimo algoritmo di cifratura: RSA.

RSA fa parte di quella branca della crittografia moderna che prende il nome di crittografia asimmetrica. A differenza della crittografia classica o simmetrica, quella asimmetrica prevede l'esistenza di due tipi di chiavi, una pubblica ed una privata. Questo approccio permette situazioni del seguente tipo:

- Alice vuole spedire un messaggio a Bob di modo che solo Bob possa leggerlo, userà quindi la chiave pubblica di Bob per cifrare il messaggio sapendo che solo Bob con la sua chiave privata potrà decifrarlo.

- Alice vuole spedire un messaggio a Bob di modo che, non solo Bob sia l'unico a poterlo leggere, ma che esso abbia la certezza che il mittente del messaggio sia proprio Alice. Alice quindi cifra il messaggio con la propria chiave privata e successivamente con quella pubblica di Bob. In questo modo all'atto della ricezione, Bob potrà applicare la propria chiave privata e la chiave pubblica di Alice per poter decifrare il messaggio essendo certo della provenienza dello stesso.

Il funzionamento di RSA non è particolarmente complesso.
Supponiamo che Alice e Bob stiano avendo un dialogo segreto, ossia non vogliono che un eventuale haker possa intercettare la loro comunicazione e comprenderne il significato.
Sia quindi M il messaggio che si vogliono scambiare.
Ognuno di loro sceglie due numeri primi p e q li moltiplica tra di loro ottenendo N=$p.q$.
In seguito calcolano $\varphi(N) = (p-1)(q-1)$.
Scelgono infine un numero e coprimo con $\varphi(N)$ e minore dello stesso e calcolano d tale per cui $e.d \equiv 1 mod(N)$.
Ora (N, e) è la chiave pubblica, mentre quella privata è (N, d).
Il messaggio visibile sulla rete è il seguente: $c = mex^e mod(N)$ che verrá poi decifrato applicando una semplice esponenziazione di esponente d, elemento della chiave pubblica del mittente, ossia $mex = c^d mod(N)$.

La sicurezza di quest'algoritmo risiede nella difficoltà computazionale di fattorizzare il numero N nei suoi fattori primi e quindi nel trovare la funzione $\varphi(N)$ che permetterebbe di trovare l'inverso moltiplicativo di e e quindi rompere i sistema.

Nel 1991 la RSA Laboratories propose come sfida la fattorizzazione di 54 semiprimi (prodotti di due primi) con un numero di cifre compreso tra 100 e 617. Ad oggi solo i 12 piú piccoli sono stati fattorizzati e, nonostante il 2007 vide la chiusura dell'RSA Challenge in molti ancora si dilettano nel tentativo di fattorizzarli.

L'algoritmo che ci proponiamo di implementare è il Crivello quadratico di Pomerance, uno dei più veloci algoritmi di fattorizzazione ad oggi conosciuto, assieme al crivello con campi di numeri. Questi algoritmi sono subesponenziali, il crivello nella fattispecie, dato N il numero da fattorizzare ha un costo temporale nell'ordine di $e * sqrt(log(N)log(log(N))$.

Se un haker riuscisse ad intercettare i messaggi crittati con RSA, conoscendo la chiave pubblica del mittente, potrebbe rompere il sistema fattorizzando il primo elemento della chiave stessa, ossia N. Fattorizzando N otterrebbe i suoi fattori primi p e q, e potrebbe calcolare $\varphi(N)$, per poter quindi calcolare l'inverso moltiplicativo della seconda parte della chiave, (e) modulo $\varphi(N)$

2 Crivello Quadratico

Il Crivello quadratico è assieme al Crivello coi Campi di Numeri l'algoritmo di fattorizzazione piú veloce ad oggi conosciuto. Il costo computazionale risulta essere asintoticamente subesponenziale nell'ordine di $O(exp(sqrt(log(N))log(log(N))))$.
Per comprendere l'algoritmo occorre presentare almeno in parte la tecnica di fattorizzazione di Fermat. Fermat osservó che per trovare una fattorizzazione di un numero N si possono trovare due numeri X e Y tali per cui $X^2 - Y^2 = N$ trovando quindi $(X+Y)(X-Y)$. Qualora $X+Y$ o $X-Y$ non risultassero fattori banali, ossia 1 o N stesso, avremmo trovato una fattorizzazione completa per N.
Il Crivello Quadratico parte da questa semplice idea sviluppandola per ottenere un algoritmo alquanto efficiente e piuttosto articolato.

Algoritmo

Dato un numero N intero, il crivello, a differenza dell'algoritmo di Fermat, cerca dei valori X e Y tali per cui valga la relazione $X \equiv Y \mod N$, successivamente ricerca il massimo comun divisore tra $(X-Y, N)$. Iniziamo calcolando una base di fattori primi FB di dimensione $k = k(N)$. Questo calcolo avviene mediante la scrematura di una base data dal crivello di eratostene. La discriminante per i fattori primi p della base è che essi abbiano simbolo di Legendre $(N|P) = 1$, ossia siano tali per cui N sia un residuo quadratico modulo p Sia ora $s = \sqrt[2]{N}$ impostiamo il seguente polinomio: $Q(A) = (A+s)^2 - N$. Siamo certi che $Q(A) \equiv N$ sia un quadrato perfetto, il lavoro ora consiste nel trovare dei valori di A tali per qui $Q(A)$ si fattorizzi completamente sulla base di fattori precedentemente calcolata. Quando uno di questi polinomi si fattorizza completamente sulla base FB creiamo un vettore $v = (\alpha_1, \alpha_2, .., \alpha_n)$ dove ogni α_i rappresenta l'esponente dell'i-esimo numero primo nella fattorizzazione di $Q(A)$. Calcoliamo quindi un altro vettore $v_2 = (\alpha_1, \alpha_2, .., \alpha_n)_2$ ossia il vettore degli esponenti in base binaria. Nel caso banale in cui v_i fosse identicamente nullo, allora ogni primo avrebbe un esponente pari, in questo caso sarebbe un quadrato perfetto e, quindi, avremmo trovato una congruenza del tipo $X^2 \equiv Y^2 \mod N$. Anche se questo accadesse potremmo aver ottenuto una fattorizzazione banale, si prosegue quindi con la parte dell'algoritmo che riguarda prettamente l'algebra lineare. I vettori $v_2 i$ vengono inseriti tutti in una grossa matrice. Dall'algebra sappiamo che, data una matrice di k colonne necessita di almeno k+1 righe per ottenere almeno una dipendenza lineare. Nel nostro caso occorrerà quindi trovare almeno $k+m$ con $m \geq 1$ per poter ottenere un numero di righe tale da permettere di trovare almeno una dipendenza lineare.
Consideriamo le seguenti matrici:

v_1	3	4	2	7
v_2	3	2	2	2
v_3	5	2	3	1
v_4	6	1	3	3
v_5	2	2	2	3

v_1^2	1	0	0	1
v_2^2	1	0	0	0
v_3^2	1	0	1	1
v_4^2	0	1	1	1
v_5^2	0	0	0	1

Notiamo che dalla combinazione lineare di $v1_2$ $v2_2$ e $v4_2$ otteniamo un vettore nullo. Andiamo quindi a considerare ora i relativi vettori non modulati $v1v2ev5$ e sommiamoli tra loro, ottenendo $v_125 = (8, 8, 6, 12)$. Se ora consideriamo la congruenza

$$Q(A_1)Q(A_2)Q(A_5) \equiv 2^8 * 3^8 * 5^6 * 7^12 \mod N$$

Considerando il membro di sinistra come la X e quello di destra come la Y della nostra relazione iniziale, possiamo ricercare la congruenza desiderata e tentare di fattorizzare N.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

3 Architettura del sistema di calcolo distribuito

3.1 MPI

MPI, acronimo per *Message Passing Interface*, è un'interfaccia che permette lo scambio di dati tramite il paradigma di scambio di messaggi fra processi. Lo scambio può avvenire sia fra processi sulla stessa macchina (tipicamente tramite memoria condivisa), che fra processi su macchine differenti (tramite qualche protocollo di rete come TCP/IP). Il principale vantaggio è l'indipendenza del codice dalla configurazione utilizzata. In altre parole, il programmatore che utilizza questa interfaccia non ha bisogno di sapere quanti processori avrà a disposizione e su quante macchine essi siano distribuiti, si dovrà semplicemente occupare di scambiare i dati con le primitive offerte dall'interfaccia, e il mezzo di comunicazione verrà poi determinato automaticamente a seconda dell'implementazione di MPI e della disponibilità di risorse.

4 Tecnologie utilizzate

4.1 OpenMPI

OpenMPI è un'implementazione open source di MPI (3.1). È in grado di gestire la comunicazione fra processi con una moltitudine di tecnologie, fra cui TCP per la comunicazione fra processi in esecuzione su macchine facenti parte di una rete di tipo classico, 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 lanciato il processo contatterà le altre tramite protocollo SSH, e lancerà opportunamente 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 orgine). 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.

Di seguito verranno esposte le primitive più importanti della libreria.

- MPI_Init: Inizializza la libreria, va chiamata prima di utilizzare qualunque altra funzione di MPI.
- MPI_Comm_size: Ritorna il numero di processi all'interno del comunicatore specificato.
- MPI_Comm_rank: Ritorna il rank del processo all'interno del comunicatore specificato.
- MPI_Abort: Termina tutti i processi nel comunicatore specificato.
- MPI_Get_processor: Ritorna il nome del processore che sta eseguendo il processo.
- MPI_Finalize: Finalizza la libreria, dopo questa chiamata non è più possibile utilizzare funzioni di libreria.
- MPI_Send: Send bloccante classica, la funzione ritorna quando il buffer contenente i dati da spedire è riutilizzabile.
- MPI_Recv: Receive bloccante classica, la funzione ritorna quando il buffer contiene i dati ricevuti ed è utilizzabile.
- MPI_Isend: Send non bloccante, ritorna un id della richiesta, utilizzabile per verificare lo stato dell'operazione.
- MPI_Irecv: Receive non bloccante, ritorna un id della richiesta, utilizzabile per verificare lo stato dell'operazione.
- MPI_Test: Controlla se l'operazione richiesta è terminata o meno.
- MPI_Wait: Attende il termine dell'operazione richiesta.
- MPI_Pack e MPI_Unpack: Impacchettano e spacchettano dati da spedire.

5 Strategia di parallelizzazione dell'algoritmo

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

6 Algoritmo parallelizzato

L'algoritmo ha una particolaritá molto utile sotto l'aspetto della complessita temporale: la sezione del codice deputata alla valutazione del polinomio $Q(A)$ è fortemente parallelizzabile. Per ottenere le congruenze necessarie dobbiamo scomporre un numero notevole di polinomi valutati sulla base di fattori FB. Questa parte del lavoro puó essere fortemente parallelizzata in quanto aumentando il lavoro demandato ai singoli processori è essenzialmente quello di valutare e scomporre i polinomi su intervalli diversi; questo permette di demandare un notevole calcolo ai nodi paralleli e quindi uno speed-up notevole all'aumentare del numero di nodi. per quanto la parte appena successiva, quella di eliminazione gaussiana è intrinsecamente seriale.

```
#include "../include/smart_sieve.h"
#include "../include/linear_algebra.h"
#include "../include/matrix.h"

unsigned int smart_sieve(mpz_t n,
                         unsigned int* factor_base,
                         unsigned int base_dim,
                         pair* solutions,
                         mpz_t begin,
                         unsigned int interval,
                         unsigned int block_size,
                         unsigned int max_fact) {
  mpz_t end;
  mpz_init(end);

  // questo processo mpz deve prendere A in [begin, end]
  mpz_add_ui(end, begin, interval); // end = begin + interval

  mpz_t n_root;
  mpz_init(n_root);
  mpz_sqrt(n_root, n);

  int stop_flag = 0;
  char stop_signal;
  MPI_Request request;
  MPI_Status status;
  MPI_Irecv(&stop_signal, 1, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &
      request);

  /* Inizio della parte di codice eseguita da ogni thread */
  #pragma omp parallel
  {

    /* Dichiarazione dei buffer per la trasmissione */
    unsigned int* buffer; // buffer per le fattorizzazioni
    init_vector(&buffer, base_dim); // dim+1 per il flag di
        controllo della fine
    unsigned char* buffer_as; // buffer per (A + s)
    buffer_as = malloc(sizeof(unsigned char) * BUFFER_DIM);

    /* Dichiarazione strutture dati per raccolta risultati */
```

```
    unsigned int** exponents; // Matrice temporanea degli
        esponenti
    init_matrix(&exponents, block_size, base_dim);
    word** used_rows; // Vettore che segna quali esponenti sono
        stati inizializzati
    init_matrix_l(&used_rows, 1, (block_size / N_BITS) + 1);
    mpz_t* evaluated_poly; // Vettore temporaneo dei Q(A)
        valutati
    init_vector_mpz(&evaluated_poly, block_size);
    mpz_t* As; // A + s temporanei
    init_vector_mpz(&As, block_size);

    // Ogni thread calcola per A in [begin_thread, end_thread]
    mpz_t begin_thread;
    mpz_init(begin_thread);
    mpz_t end_thread;
    mpz_init(end_thread);

    mpz_t last_block;
    mpz_init(last_block);
    mpz_t end_block;
    mpz_init(end_block);

    mpz_t intermed; // Valore appoggio
    mpz_init(intermed);

    mpz_t A; // Valore di A di Q(A) = (A + s)^2
    mpz_init(A);

    mpz_t l;
    mpz_init(l);

    mpz_t j;
    mpz_init(j);

    mpz_t begin_solution1;
    mpz_init(begin_solution1);
    mpz_t begin_solution2;
    mpz_init(begin_solution2);

    // Indice per accedere a Q(A) memorizzato in posizione (A -
        offset) tale che index < block_size
    mpz_t index_mpz;
    mpz_init(index_mpz);
    unsigned int index;

    int n_bytes; // num bit per vettore righe usate nella matrice

    unsigned int i; // Indice generico
    unsigned char go_on = 1;
    unsigned int h; // Usato per copiare gli base_dim esponenti

    unsigned int fact_count = 0; // Numero fattorizzazioni
        trovate
```

```
    unsigned long k; // Indici generici

    /*
        ************************************************************************
        */

    max_fact += base_dim; // Le k + n fattorizzazioni da trovare

    int threads = omp_get_num_threads();
    int thread_id = omp_get_thread_num();

    unsigned int dom_decomp = interval / (threads);
    mpz_add_ui(begin_thread, begin, dom_decomp * thread_id); //
        begin_thread = begin + (dom_decomp * thread_id)
    mpz_add_ui(end_thread, begin_thread, dom_decomp); //
        end_thread = begin_thread + dom_decomp
    //gmp_printf("begin=%Zd, end=%Zd, interval=%d, block_size=%d\
        n", begin, end, interval, block_size);
    //gmp_printf("%d) begin_thread=%Zd, end_thread=%Zd, dom_decmp
        =%d\n", thread_id, begin_thread, end_thread, dom_decomp);
    //printf("###originali\n");
    mpz_pair * solutions_ = malloc(sizeof(mpz_pair) * base_dim);
    for(i = 0; i < base_dim; ++i) {
      mpz_init(solutions_[i].sol1);
      mpz_init(solutions_[i].sol2);

      mpz_set_ui(solutions_[i].sol1, solutions[i].sol1);
      mpz_set_ui(solutions_[i].sol2, solutions[i].sol2);

      //gmp_printf("x_%d = %Zd, ", factor_base[i], solutions_[i].
          sol1);
      //gmp_printf("y_%d = %Zd\n", factor_base[i], solutions_[i].
          sol2);

    }
    //printf("###ricalcolate\n");
    if(mpz_cmp_ui(begin_thread, 0) != 0)
      for(i = 0; i < base_dim; ++i) {
        //unsigned int old1 = solutions_[i].sol1;
        //unsigned int old2 = solutions_[i].sol2;

        //unsigned int f = (startfrom - solutions[i].sol1) /
            factor_base[i] + 1;
        //solutions[i].sol1 = solutions[i].sol1 + f * factor_base
            [i];
        //f = (startfrom - solutions[i].sol2) / factor_base[i] +
            1;
        //solutions[i].sol2 = solutions[i].sol2 + f * factor_base
            [i];

        while(mpz_cmp(solutions_[i].sol1, begin_thread) < 0)
          mpz_add_ui(solutions_[i].sol1, solutions_[i].sol1,
              factor_base[i]);
        while(mpz_cmp(solutions_[i].sol2, begin_thread) < 0)
```

```
          mpz_add_ui(solutions_[i].sol2, solutions_[i].sol2,
              factor_base[i]);

        //gmp_printf("x_%d = %Zd, ", factor_base[i], solutions_[i
            ].sol1);
        //gmp_printf("y_%d = %Zd\n", factor_base[i], solutions_[i
            ].sol2);
      }
    //printf("###fine calcolo soluzioni \n");

    mpz_sub_ui(last_block, end_thread, block_size); // last_block
        = end_thread - block_size

    // for(l = begin_thread; l < last_block && go_on; l +=
        block_size)
    for(mpz_set(l, begin_thread);
       (mpz_cmp(l, last_block) < 0) && go_on && !stop_flag;
       mpz_add_ui(l, l, block_size)) {

      for(i = 0; i < ((block_size / N_BITS) + 1); ++i) { // Reset
          righe usate
        set_matrix_l(used_rows, 0, i, 0);
      }

      mpz_add_ui(end_block, l, block_size); // end_block = l +
          block_size
      //gmp_printf("l=%Zd < %Zd [%Zd, %Zd)\n", l, last_block, l,
          end_block);

      for(i = 0; i < block_size; ++i) { // Calcolo Q(A) e (A + s)
          per A in [l, l + block_size]
        mpz_add_ui(A, l, i); // A = i + l
        mpz_add(intermed, n_root, A); // A + s
        mpz_set(As[i], intermed);
        mpz_mul(intermed, intermed, intermed); // (A + s)^2
        mpz_sub(evaluated_poly[i], intermed, n);

        //gmp_printf("Q(%Zd)=%Zd, ", A, evaluated_poly[i]);
      }
      //printf("\n");

      for(i = 0; i < base_dim && go_on && !stop_flag; ++i) {
        /* Sieve con Xp */
        // for(j = solutions_[i].sol1; j < end_block && go_on; j
            += factor_base[i])
        for(mpz_set(j, solutions_[i].sol1);
           (mpz_cmp(j, end_block) < 0) && go_on && !stop_flag;
           mpz_add_ui(j, j, factor_base[i])) {

          //gmp_printf("\txp) j=%Zd < %Zd [+=%d](j, end_block)\n
              ", j, end_block, factor_base[i]);
          mpz_sub(index_mpz, j, l);
          index = mpz_get_ui(index_mpz); // Siccome (j - l) <
              block_size    un uint sicuro
```

```
        while(mpz_divisible_ui_p(evaluated_poly[index],
            factor_base[i])) {
          //gmp_printf("\t\tQ(A) = %Zd / %d = ", evaluated_poly
              [index], factor_base[i]);
          if(get_k_i(used_rows, 0, index) == 0) { // Se non
              sono mai stati usati gli esponenti
            for(k = 0; k < base_dim; ++k)
              set_matrix(exponents, index, k, 0);
            set_k_i(used_rows, 0, index, 1);
          }

          set_matrix(exponents, index, i, get_matrix(exponents,
               index, i) + 1); // ++exponents[j][i]
          mpz_divexact_ui(evaluated_poly[index], evaluated_poly
              [index], factor_base[i]); // Q(A) = Q(A) / p
          //gmp_printf("%Zd  (poly[%d])\n", evaluated_poly[
              index], index);
        }
      }
      mpz_set(solutions_[i].sol1, j); // solutions[i].sol1 = j;
           // Al prossimo giro ricominciamo da dove abbiamo
          finito

      /* Sieve con Yp */
      // for(j = solutions_[i].sol2; j < end_block && go_on; j
          += factor_base[i])
      for(mpz_set(j, solutions_[i].sol2);
          factor_base[i] != 2 && (mpz_cmp(j, end_block) < 0) &&
               go_on && !stop_flag;
          mpz_add_ui(j, j, factor_base[i])) {

        //gmp_printf("\txp) j=%Zd < %Zd [+=%d](j, end_block)\n
            ", j, end_block, factor_base[i]);
        mpz_sub(index_mpz, j, l);
        index = mpz_get_ui(index_mpz); // Siccome (j - l) <
            block_size    un uint sicuro

        while(mpz_divisible_ui_p(evaluated_poly[index],
            factor_base[i])) {
          //gmp_printf("\t\tQ(A) = %Zd / %d = ", evaluated_poly
              [index], factor_base[i]);
          if(get_k_i(used_rows, 0, index) == 0) { // Se non
              sono mai stati usati gli esponenti
            for(k = 0; k < base_dim; ++k)
              set_matrix(exponents, index, k, 0);
            set_k_i(used_rows, 0, index, 1);
          }

          set_matrix(exponents, index, i, get_matrix(exponents,
               index, i) + 1); // ++exponents[j][i]
          mpz_divexact_ui(evaluated_poly[index], evaluated_poly
              [index], factor_base[i]); // Q(A) = Q(A) / p
          //gmp_printf("%Zd  (poly[%d])\n", evaluated_poly[
```

```
                index], index);
          }
        }
        mpz_set(solutions_[i].sol2, j); // solutions[i].sol2 = j;
             // Al prossimo giro ricominciamo da dove abbiamo
            finito
      }

      // Spedisco le fattorizzazioni trovate in questo blocco
      for(i = 0; i < block_size; ++i) {
        if(mpz_cmp_ui(evaluated_poly[i], 1) == 0) {
          ++fact_count;
          for(k = 0; k < base_dim; ++k)
            buffer[k] = get_matrix(exponents, i, k);

          /* MPI_Send */
          #pragma omp critical
          {
            MPI_Send(buffer, base_dim, MPI_UNSIGNED, 0, ROW_TAG,
                MPI_COMM_WORLD);
            n_bytes = (mpz_sizeinbase(As[i], 2) + 7) / 8;
            *buffer_as = 0;
            mpz_export(buffer_as, NULL, 1, 1, 1, 0, As[i]);
            MPI_Send(buffer_as, n_bytes, MPI_UNSIGNED_CHAR, 0,
                AS_TAG, MPI_COMM_WORLD);

            if(stop_flag == 0)
              MPI_Test(&request, &stop_flag, &status);
            //printf("%d) checking stop_signal = %d\n", thread_id
                , stop_signal);
          }
        }
      }

    }

    mpz_clear(begin_thread);
    mpz_clear(end_thread);
    mpz_clear(end_block);
    mpz_clear(intermed);
    mpz_clear(A);
    mpz_clear(l);
    mpz_clear(j);
    mpz_clear(begin_solution1);
    mpz_clear(begin_solution2);
    mpz_clear(index_mpz);

    free(buffer);
    free(buffer_as);

    finalize_matrix(&exponents, base_dim);
    finalize_matrix_l(&used_rows, (block_size / N_BITS) + 1);

    for(unsigned int u = 0; u < block_size; u++) {
```

```
      mpz_clear(evaluated_poly[u]);
      mpz_clear(As[u]);
    }
  }

  mpz_clear(end);
  mpz_clear(n_root);

  return stop_flag;
}
```

7 Risultati

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

3

8 Sviluppi futuri

L'algoritmo si presta a diversi sviluppi orientati sia al miglioramento delle prestazioni che al miglioramento dell'interfaccia con l'utente.

In merito alle prestazioni la prima modifica utile che potrebbe essere apportata è la seguente: l'architettura del sistema in questo momento è di tipo master slave, nella fattispecie il master svolge semplicemente il compito di suddividere i dati e di recuperarli per poi eseguire la parte seriale dell'algoritmo, ossia l'eliminazione Gaussiana. Sarebbe utile modificare l'algoritmo affinchè nel tempo di attesa anche il processo master eseguisse la parte degli slave così da ottimizzare l'uso delle cpu.

Considerando l'esecuzione dell'algoritmo in parallelo sui nodi di un supercomputer occorre considerare il problema del walltime, ossia del tempo massimo assegnato ad un processo dal gestore di code. Se il processo supera il walltime senza aver terminato correttamente l'esecuzione viene killato. Questo è un problema piuttosto notevole se si tratta di algoritmi che richiedono diversi giorni di calcolo su determinati dati; occorre quindi permettere all'algoritmo di crearsi dei checkpoint facendo un salvataggio dei dati intermedi per poi riprendere la computazione dal punto in cui era stato bloccato.

/* DETTAGLI TECNICI */