Indice

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.

Implementazione seriale

Base di fattori

Divisione degli intervalli

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.

4.2 OPENMP

A differernza di OPENMPI, OPENMP è una libreria per lo sviluppo di applicazioni parallele in sistemi a memoria condivisa, nel nostro caso per processi multithread che lavoreranno su cpu multicore. Per specificare quali parti del codice debbano essere splittate sui core senza race condition, viene usata la direttiva #pragma omp parallel, per indicare una sezione critica invece viene utilizzata la direttiva #pragma omp critical

Le funzioni principali sono elencate di seguito:

- omp_get_num_procs: Restituisce il numero di processori disponibili quando viene chiamata la funzione.
- omp_get_num_threads: Restituisce il numero di thread nell'area parallela.
- omp_get_wtime: Restituisce un valore in secondi del tempo trascorso da un certo punto.

4.3 GMP

GMP é una libreria libera per l'utilizzo di aritmetica a precizione arbitraria e fa parte del progetto GNU (Gnu Multile Precision). L'unica limitazione alla dimensione dei valori assumibili da una variabile mpz è la dimensione della memoria del dispositivo. GMP viene utilizzata per algoritmi crittografici, applicazioni relative alla sicurezza delle reti e per sistemi di algebra numerica. Internamente GMP rappresenta gli interi come dei vettori. L'utilizzo di questa libreria all'interno dell'algoritmo è stato necessario sia per la rappresentazione dell'intero da fattorizzare e per i suoi fattori primi, se pensiamo che RSA utilizza numeri con più di 100 cifre viene normale utilizzare rappresentazioni in precisione arbitraria, che per gli esponenti all'interno delle fattorizzazioni: per quanto potesse sembrare poco probabile si è notato che gli esponenti nelle fattorizzazioni raggiungono dimensioni elevate, si è quindi dimostrato necessario utilizzare variabili mpz per rappresentare anche questi.

Di seguito verranno esposte le primitive più importanti della libreria.

- gmp_init: Questa funzione va chiamata ad ogni dichiarazione per inizializzare la variabile
- gmp_add: Somma due mpz
- gmp_sub: Sottrae due mpz
- gmp_mul: Moltiplica due mpz
- gmp_div: Divide due mpz
- gmp_mod: Calcola il resto della divisione intera tra due mpz
- gmp_legendre: Calcola il valore del simbolo di Legendre
- gmp_sqrt: Radice quadrata
- gmp_pow: Potenza
- gmp_gcd: Massimo Comun Divisore
- gmp_func_ui: specificando _ui indichiamo che il secondo parametro è un intero senza segno

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

Descriveremo l'architettura parallela dell'algoritmo.

6.1 Base di fattori

Il nodo master potrebbe ricercare la base di fattori ed inviare il vettore di primi validi ai restanti nodi slave del cluster. Siccome gli slave resterebbero inattivi fino alla ricezione della base, in questa implementazione parallela si è deciso di far determinare la base di fattori ad ogni nodo indipendentemente. Tutti i nodi impiegheranno un tempo circa uguale per determinarla e inizieranno subito la parte successiva dell'algoritmo evitando il trasferimento di questi dati.

6.2 Il ruolo del nodo master

Il nodo master entra in una procedura che colleziona i risultati provenienti dagli slave durante la fase di crivello. Ricordiamo che in questa parte si deve ricercare per quali valori di A il polinomio $Q(A)$ si fattorizza completamente sulla base di fattori. I nodi effettuano dunque le fattorizzazioni in simultanea ed inviano il vettore degli esponenti e il vettori con i $(A+s)$ calcolati. Sono necessarie $K+10$ fattorizzazioni per proseguire nell'algoritmo (dove K è la dimensione della base di fattori) dunque una volta collezionato tale numero di risultati, il master interrompe l'ascolto dei messaggi MPI in arrivo dagli slave ed invia un segnale di stop a tutti indicando di concludere la procedura di crivello.

Terminata la parte di "accumulo dei risultati" la parte parallela dell'algortimo finisce e l'esecuzione riprende come nella versione seriale. La parte di eliminazione gaussiana e quella finale di calcolo delle congruenze non è infatti parallelizzabile.

In dettaglio, estratto dal file `quadratic_sieve.c`, la procedura di ricezione del master:

```
while(fact_count < max_fact + base_dim) {
    /* Ricevo il vettore di esponenti */
    MPI_Recv(buffer_exp, base_dim, MPI_UNSIGNED,
             MPI_ANY_SOURCE, ROW_TAG,
             MPI_COMM_WORLD, &status);
    source = status.MPI_SOURCE;
    /* Salvo il contenuto del buffer */
    for(unsigned int i = 0; i < base_dim; ++i)
      set_matrix(exponents, fact_count, i, buffer_exp[i]);
    /* Ricevo l'mpz contenente (A + s) */
    MPI_Recv(buffer_As, BUFFER_DIM, MPI_UNSIGNED_CHAR, source,
             AS_TAG, MPI_COMM_WORLD, &status);
    /* Estrapolo la dimensione del vettore mpz */
    MPI_Get_count(&status, MPI_UNSIGNED_CHAR, &count);
    /* Lo "importo" mediante questa primitiva della libreria */
    mpz_import(As[fact_count], count, 1, 1, 1, 0, buffer_As);
    ++fact_count;
}
```

Sono presenti due tag per le chiamte a `MPI_Recv` per forzare la ricezione in sequenza di un vettore di esponenti, il `ROW_TAG`, ed un vettore di $(A+s)$, l'`AS_TAG`. Viene estratto l'id del mittente dalla ricezione del vettore degli esponenti. Questo è necessario per prelevare il vettore degli $(A+s)$ relativo agli stessi dati. Altri vettori $(A+s)$ inviati da

altri slave potrebbero essere giunti prima nella coda di ricezione rispetto ai messaggi dello slave di cui stiamo considerando il vettore degli esponenti. É fondamentale l'associazione vettore esponenti e vettore $(A + s)$.

Sorge tuttavia una problematica di uso efficiente delle risorse. Il nodo master resta in attesa delle fattorizzazioni durante la fase di crivello, queste fattorizzazione - per numeri sufficientemente grossi - sono più "rarefatte", lasciando quindi il processore del nodo inattivo per la maggior parte del tempo. Una soluzione a questo problema che non preveda alcuna modifica al codice è la seguente: si lanciano i processi mpi e si istruisce il gestore delle code (qualora preveda questa opzione) di attivare un singolo processo per nodo ad eccezione del master. Su questo sarà infatti lanciato un secondo processo mpi che occuperà i tempi di inattvità della cpu dovuti al primo processo in attesa, ricercando anch'esso fattorizzazioni per i $Q(A)$. Nell'implementazione dell'algoritmo proposta, questa soluzione non è attuabile con successo poichè `MPI_Recv` effettua un attesa attiva consumando comunque il 100% della risorsa cpu. Si potrebbe sostituire la `MPI_Recv` con la ricezione asincrona e mandare in sleep il processo per un lasso di tempo fissato forzando un attessa che, seppur sempre attiva, avvenga con frequenza inferiore.

6.3 Il ruolo dei nodi slave

Il crivello è la parte più onerosa di tutto l'algoritmo. Viene richiesto di provare svariati valori di A per tentare di fattorizzare completamente il polinomio $Q(A)$ calcolato. Possiamo considerare un dominio di valore di A che è l'intervallo $[0, M]$ e decomporlo in tante parti quanti sono i nodi slave del cluster. Lo slave i-mo effettuerà i calcoli parallelamente agli altri sui valori di A in $[B_i, F_i]$. Si calcola il polinomio $Q(A)$, si cerca di fattorizzarlo sulla base di fattori. Nella versione seriale dalla procedura di crivello, si ritornavano due strutture dati contenenti i vettori degli esponenti delle fattorizzazioni e il vettore degli $(A + s)$. Nella versione parallela memorizzeremo temporaneamente queste "soluzioni" per poi inviarle al master, tramite MPI, ogni volta che si saranno tentati n valori di A. Il comportamento della funzione di crivello parallelo è identico alla controparte parallela con la differenza che anzichè salvare i risutlati in una struttura dati si inviano mediante il protocollo MPI al nodo master.

L'architettura del nostro sistema è però ibrida: abbiamo svariati processi MPI al cui interno girano svariati thread. Analizzeremo più nel dettaglio come sarà decomposto il dominio dei valori di A per assegnare un pezzo a ciascun processo e, al loro interno, a ciascun thread.

6.4 Decomposizione del dominio

La prima suddivisione del dominio: a livello di processo MPI.

```
mpz_set_ui(begin, interval * (rank - 1));
do {
   stop_flag = smart_sieve(N, factor_base, n_primes, solutions,
                           begin, interval,
                           block_size, max_fact);

   mpz_add_ui(begin, begin, interval * (comm_size-1));
} while(!stop_flag);
```

Supponiamo di considerare lo slave i-mo. Il rank di tale slave sarà $i + 1$ (per via della presenza del master). Il parametro "interval" rappresenta il blocco di valori

di A che ogni slave utilizza alla volta. Lo slave i-mo inizialmente autodetermina in base al suo rank l'intervallo sul quale deve operare assegnando alla variabile "begin" il valore "interval * i" (si ricorda che $rank = i + 1$). La funzione `smart_sieve` opera dunque sull'intervallo $[begin, begin + interval]$. Se nessun segnale di stop è arrivato dal master lui autodetermina l'intervallo successivo da esaminare tenendo conto della presenza degli altri nodi facendo `begin = begin + (interval * comm_size - 1)`. La variabile `comm_size` contiene la dimensione del cluster alla quale va tolto il nodo master che non esegue il crivello.

Passiamo ora alla seconda suddivisione: a livello di thread.

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

  int threads = omp_get_num_threads();
  int thread_id = omp_get_thread_num();
  unsigned int dom_decomp = interval / (threads);
  /* begin_thread = begin + (dom_decomp * thread_id) */
  mpz_add_ui(begin_thread, begin, dom_decomp * thread_id);
  /* end_thread = begin_thread + dom_decomp */
  mpz_add_ui(end_thread, begin_thread, dom_decomp);

  /* Parte d crivello */
  /* ... */
}
```

Similmente a come avveniva a livello di processi MPI, l'intervallo di dati (parametro "interval") sul quale viene chiamato `smart_sieve` viene suddiviso in tanti pezzi quanti i thread disponibili nel processore.

6.5 Trasmissione dati al master

```
for(i = 0; i < block_size; ++i) {
  /* Se Q(A) e' stato fattorizzato completamente */
  if(mpz_cmp_ui(evaluated_poly[i], 1) == 0) {
    ++fact_count;
    /* Carico i dati degli esponenti nel buffer */
    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);
      /* Effettuo il controllo sul segnale di stop del master */
      if(stop_flag == 0)
```

```
        MPI_Test(&request, &stop_flag, &status);
    }
  }
}
```

L'invio del vettore degli esponenti e del vettore degli (A + s) avviene solo al termine di ogni blocco. Come nella versione seriale, per minimizzare la dimensione delle strutture dati, l'algoritmo procede per blocchi di dati di dimensione `block_size`. Siccome le primitive di MPI non sono thread safe, è stato necessario eseguire la parte di invio dei dati in mutua esclusione tra thread (omp critical). Come detto nel paragrafo sul master, si nota che sono presenti due tag MPI per discriminare quale tipo di dato è stato inviato, se il vettore degli esponenti oppure il vettore (A + s).

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.

8 Struttura del programma

8.1 Suddivisione dei sorgenti

L'implementazione del crivello quadratico proposto si suddivide in due versioni: seriale e parallelo. Entrambe le versioni hanno una struttura dei sorgenti simile. Ogni file contiene funzioni che occorrono in una determinata fase del crivello. La directory dei sorgenti è "/src". Il programma è suddiviso in 7 file:

- `base_fattori.c`
- `eratostene.c`
- `linear_algebra.c`
- `quadratic_sieve.c`
- `trivial_fact.c`
- `vector.c`
- `matrix.c`

Il file `base_fattori.c` contiene le funzioni per il calcolo della base di fattori. Il file `eratostene.c` contiene la funzione per il calcolo dei primi n numeri primi mediante il crivello di Eratostene. In `linear_algebra.c` sono contenute le funzioni per l'eliminazione gaussiana, eseguita sulla matrice degli esponenti. Nello stesso file sono presenti anche le funzioni che permettono il calcolo delle congruenze nella parte finale dell'algoritmo. `quadratic_sieve.c` è il file principale del programma. Da questo vengono richiamate le funzione che eseguono le varie fasi dell'algoritmo. `trivial_fact.c` è il file con le funzioni che effettuano la fattorizzazione per tentativi necessarie nella parte iniziale dell'algoritmo. `vector.c` e `matrix.c` contengono funzioni di servizio per la gestione delle strutture dati utilizzati. `vector.c` gestisce la struttura dati vettore dinamico istanziata per vari tipi di dato: `unsigned int`, `unsigned long` e `mpz_t`. `matrix.c` gestisce matrici di dimensione dinamica in entrambe le dimensioni. Come per i vettori è presente la variante `unsigned int`, `unsigned long` e `mpz_t`. Sono presenti header file per ognuno dei sorgenti c nella directory "/include".

8.2 Compilazione

Per la compilazione del codice sono presenti tre makefile. Il primo, chiamato semplicemente "Makefile", occorre per compilare generalmente il codice per un pc linux. Un avvertenza: è necessario creare le directory "lib" ed "executables" nella cartella principale dell'algoritmo (es: parallel/executables). Queste cartelle conterranno i prodotti della compilazione, executables per gli eseguibili e lib per gli object file. Gli ulteriori makefile, Makefile.intel e Makefile.gnu, sono stati utilizzati per la compilazione del codice sul Galileo utilizzando il compilatore mpi di intel e gnu.

8.3 Esecuzione del programma parallelo

Il main del programma (che è da considerare puramente a scopo di debugging e non main di un applicazione definitiva) ha la seguente interfaccia a linea di comando:

```
$ mpirun [opzioni_mpi] executables/qs [n1] [n2] [
    dim_crivello_eratostene] [dim_intervallo] [dim_blocco]
```

`n1`, `n2` A scopo di debugging l'applicazione prende in input due numeri che moltiplica tra loro producendo il numero N da fattorizzare. Porre uno dei due numeri a 1 per inserire direttamente l'N da fattorizzare;

`dim_crivello_eratostene` Per calcolare la base di fattori si calcolano prima i numeri primi da 0 a `dim_crivello_eratostene` (nota: la base di fattori sarà molto più piccola di questo parametro);

`dim_intervallo` dimensione insieme di A per i queali ogni salve fattorizza i rispettivi $Q(A)$;

`dim_blocco` Dato un intervallo ad uno slave questo lo separa in altri sottointervalli che assegna ad ogni thread. Ogni thread calcola alla volta `dim_blocco` valori di A;

`fact_print` Parametro che indica ogni quante fattorizzazioni trovate, il programma produce un output su stdout per mostrare la percentuale di completamento.

```
$ mpirun executables/qs-intel 1 18567078082619935259 4000
   10000000 10 1000 50
```

3

9 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 */

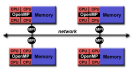

Figura 1: Architettura ibrida