Un orario scolastico parallelo

Enrico Sartorello

Anno Scolastico 2005/2006

Indice

Ringraziamenti

Per la stesura di questa tesina desidero ringraziare in primo luogo l'Istituto Tecnico Industriale Vito Volterra di San Donà di Piave, ed in particolare la Scuola di Calcolo Parallelo nella persona dell'insegnante Prof. Roberto Carrer per avermi stimolato e dato l'opportunità di approfondire le tematiche che sono alla base di questa tesina.

Ringrazio inoltre i miei colleghi studenti Baccega Andrea e Bucciol Antonio per la fondamentale collaborazione nello sviluppo del programma, senza i quali questo progetto non avrebbe potuto nascere, crescere e raggiungere risultati tangibili.

Infine, esprimo una sincera gratitudine nei confronti dei miei insegnanti, i quali si sono dimostrati a più riprese interessati agli sviluppi di questa "avventura" e disponibili ad eventuali discussioni in merito, ed al comitato organizzativo delle Olimpiadi Italiane di Informatica, grazie al quale ho scoperto che l'informatica può essere anche un momento di incontro e di sfida, uno stimolo ed una fonte di opportunità per la crescita professionale e personale.

1 Introduzione

1.1 Perché "parallelo"

Tradizionalmente, il software viene sviluppato per un'esecuzione **seriale**: l'algoritmo che risolverà il problema preso in esame viene sviluppato per essere eseguito in un'unico computer, dotato di un'unica CPU, e si compone di una serie di istruzioni che verranno eseguite una dopo l'altra.

Normalmente, la CPU si trova nell'impossibilità di eseguire più di un'operazione per istante di tempo: è proprio da questa limitazione che nasce la nobile arte della *programmazione parallela* che, nella sua accezione più semplice, si definisce come l'uso simultaneo di più risorse computazionali per risolvere un problema.

La programmazione parallela cerca di fornire un'alternativa ai rigorosi limiti imposti dalla programmazione seriale, legata in maniera inscindibile ai limiti di velocità dell'hardware e inadatta a grandi problemi che richiedono enormi risorse di calcolo, quali ad esempio:

- Problemi di simulazione o di modellamento che richiedono molta precisione;
- Problemi di gestione e manipolazione di grandi moli di dati, ad esempio i fenomeni sismici;
- Altri problemi di carattere scientifico e non, ad esempio gli studi sul genoma umano o sulla dispersione dell'inquinamento atmosferico;

A problemi di questa matrice il parallelismo, attraverso attenti studi, può fornire la possibilità di suddividere il programma in sotto-programmi più piccoli (attività o *task*), ed assegnarli a più risorse che operano contemporaneamente: queste risorse possono essere più processori integrati in un unico sistema, un numero arbitrario di computer collegati in rete, oppure una combinazione delle due possibilità.

Il concetto di fondo resta comunque molto semplice: velocizzare l'esecuzione di un programma facendo svolgere contemporaneamente più istruzioni indipendenti. Per ottenere un contesto simile è necessario che:

- a livello hardware, ci sia la disponibilità di risorse multiple collegate fra di loro (tramite bus o cavi di rete);
- a livello software, è necessario che l'algoritmo risolutivo venga implementato in maniera parallela, suddividendo in parti il problema, assegnandone ognuna ad una risorsa e cercando di bilanciare il carico di lavoro tra le varie risorse;

In questo studio si tralascierà quasi completamente l'aspetto relativo alle problematiche di natura hardware, per dedicarsi ai problemi algoritmici legati all'uso della programmazione parallela per risolvere un problema estremamente interessante: l'elaborazione di un valido orario scolastico per una scuola superiore.

Per un più approfondito studio sui pro e i contro delle tecniche di programmazione parallela e seriale, si rimanda a [1, 2].

1.2 Analisi del problema: un orario scolastico

Andiamo ora ad analizzare il problema vero e proprio.

Il nostro intento è quello di sviluppare un programma che generi un orario scolastico *valido* e *coerente*, a partire da tre insiemi di informazioni: le *cattedre*, ovvero l'assegnazione di un professore all'insegnamento di una data materia in una certa classe (con il relativo numero di ore settimanali), i *desiderata*, ovvero quell'insieme di richieste avanzate dal corpo docente (giorni liberi, ore preferite, etc) che cercano di adattare la settimana lavorativa alle esigenze dei singoli insegnanti, e i *laboratori*, aule predisposte all'insegnamento di particolari materie. Da notare che, per scelta, in fase di progettazione la palestra è stata anch'essa definita come laboratorio in cui però è contemplata la possibile presenza contemporanea di due classi.

Per *valido*, intendiamo un orario la cui distribuzione di ore sia tale per cui per insegnanti e studenti risulti essere il meno pesante possibile durante tutta la settimana. Si pensi, ad esempio, che risulta impossibile applicare un orario che accumula quattro ore di matematica in una classe tutte lo stesso giorno, in quanto questo risulterà controproducente sia per gli studenti, che dovranno subire in termini didattici una lezione molto pesante, sia per il docente, che molto probabilmente si troverà impossibilitato nel distribuire i carichi di studio e di esercizio per quella classe durante la settimana.

Per orario *coerente* o legale, intendiamo invece una distribuzione oraria tale per cui non si verifichino situazioni "impossibili" o comunque non corrette. Ad esempio, un docente non potrà mai trovarsi la stessa ora su due classi differenti.

Queste congetture porteranno a stilare una lista di *vincoli* che verrà ampiamente discussa nel paragrafo 2.1, e che servirà, attraverso un'opportuna funzione, a quantificare la *bontà* di un orario, ovvero ad ottenerne una valutazione numerica che descriverà quanto questo orario è "buono" o si discosta dalla perfezione. Nella documentazione scientifica, questo valore viene spesso indicato come *fitness* o *cost*.

1.3 L'ambiente di lavoro

Per cercare di ottenere un risultato soddisfacente è necessario predisporre un ambiente di sviluppo che si adatti il più possibile alle necessità imposte dal problema, e ci fornisca la massima trasparenza ed elasticità in ogni fase di lavoro.

Sin dall'inizio, è parsa logica la scelta di sviluppare il programma sotto ambiente GNU\Linux, in quanto questo sistema operativo si presta in maniera naturale ad accogliere progetti di questa natura, disponendo di una nutrita quantità di librerie, compilatori e debugger *free*, e di strumenti di controllo delle risorse del sistema (utilizzo della CPU, gestione dei processi, informazioni sulle connessioni, etc) fondamentali per quantificare le prestazioni complessive

del programma o eventuali problemi aleatori ricorrenti nella programmazione parallela.

Inoltre il programma è stato scritto in C, a nostro avviso il linguaggio di programmazione più malleabile e più adatto ai nostri scopi e alle nostre esigenze relative alla gestione dinamica della memoria, alla ricorsione e alla gestione di determinati servizi a basso livello (ad esempio i segnali), e questo linguaggio è storicamente legato al sistema del pinguino sin dalla nascita.

Non va dimenticato che il programma è reso parallelo grazie all'ausilio delle librerie MPI (Message-Passing Interfaces), sviluppate nel 1993-1994 da un gruppo di ricercatori: esse rappresentano uno dei primi standard per la programmazione parallela, nonché il primo in assoluto basato sullo scambio di messaggi, e possono essere integrate nei linguaggi C, C++ e Fortran ma solo in ambiente Linux. A onor del vero esiste anche un porting funzionante sotto Windows, ma questa alternativa non è neanche stata presa in considerazione.

Gli strumenti di lavoro utilizzati sono quindi:

- gcc 3.4.4 (compilatore C)
- gdb 6.3 (debugger)
- mpich2 1.0.3 (librerie MPI)
- KEdit, GEdit, Kate (editor di testo)
- gnuplot 4.0 patchlevel 0 (programma per il plotting)

Il tutto è stato utilizzato in una installazione di Linux Gentoo 2005.1 [4].

2 Nel vivo del programma

2.1 Regole e dritte per poter realizzare un buon orario: i vincoli

Prima di addentrarsi nelle scelte algoritmiche del programma, è stato necessario definire a grandi linee di quali fasi e caratteristiche sia composto il problema di creazione di un orario da applicare ad una scuola, una sorta di primo sguardo alla realtà che si è inteso esplorare.

In questa fase, fondamentale è stato l'apporto dei due "oraristi" della nostra scuola, i professori Rosengart e Romano, che ci hanno aiutato a delineare gli aspetti fondamentali che portano alla stesura di una tabella oraria, evidenziando le problematiche annesse e le procedure "umane" che tentano di fornire una soluzione.

Attraverso il confronto con questi due insegnanti, è stato possibile definire una serie di condizioni o per meglio dire *vincoli* a cui il nostro programma avrebbe dovuto certamente tener conto nei processi di generazione e valutazione di un orario, e a suddividere tali entità in due categorie differenti:

- vincoli obbligatori, che devono essere assolutamente rispettati, a cui appartengono le seguenti condizioni:
  - un professore deve insegnare almeno 5 giorni alla settimana;
  - i laboratori devono essere assegnati per primi; una volta decisa la loro scansione oraria essa non verrà più toccata nelle successive elaborazioni;
  - nel generare un orario, bisogna tener conto di insegnanti in condizioni particolari, ad esempio di insegnanti a completamento d'orario in altri istituti, per i quali può essere necessario riservare obbligatoriamente (previo accordo con l'altro o gli altri istituti in cui il docente esercita) determinati giorni liberi;
- vincoli non obbligatori, ma fondamentali ai fini della valutazione di un orario, in quanto ne determinano la qualità didattica; ad ognuno di essi può essere dato un peso differente dagli altri in base a quanto, secondo una valutazione soggettiva, l'aspetto preso in considerazione influenza la buona riuscita di un orario:
  - bisogna tener conto dei *desiderata*, richieste quali ad esempio il giorno libero, o la quasi impossibilità di insegnare determinate ore della settimana;
  - le ore di laboratorio devono essere distribuite in maniera quanto più uniforme nell'arco della settimana;
  - se la palestra vede la compresenza di due classi, esse non devono avere uno scarto di età superiore ad un anno: ad esempio, non può succedere che ad una stessa ora la palestra venga divisa tra una prima e una quarta;

- un insegnante di materia teorica non deve fare più di 4 ore di insegnamento al giorno;
- le materie vanno equamente suddivise durante la settimana: ad esempio le 3 ore di inglese di una classe non dovrebbero venire assegnate in successione il lunedì, il martedì ed il mercoledì;

A livello logico-concettuale, è parso utile porre come area di lavoro due tabelle orarie, una con le materie assegnate durante la settimana a ciascuna classe (dimensione: numero di classi * numero di ore settimanali) ed un'altra con le classi (oppure un eventuale ora libera) assegnate a ciascun insegnante durante le ore della settimana (dimensione: numero di insegnanti * numero di ore settimanali). E' palese che queste due tabelle sono strettamente legate l'una all'altra, in particolare da una è possibile ricavare l'altra e viceversa: nel codice questo è possibile richiamando le funzioni `prof2classi()` e `classi2prof()`.

2.2 Schema di funzionamento generale

Vediamo ora, senza entrare troppo nei dettagli, il funzionamento del programma da noi sviluppato.

E.N.R.I.C.O. (*ENhanced RInged Cluster Oracle*, così è stato chiamato il programma) per prima cosa si preoccupa di leggere da un file di testo tutto l'insieme di informazioni di cui ha bisogno per poter comporre un orario. Queste informazioni, racchiuse in un file da noi chiamato *Orario.ESP*, comprendono tutti i dati di cui ha bisogno il nostro programma per poter stilare un orario ex-novo: la lista di tutte le materie, la lista di tutti gli insegnanti, la lista di tutte le classi, la lista di tutte le cattedre, ove una cattedra è rappresentata dalla quaterna (materia, insegnante, classe, numero di ore).

Va notato che, per scelta progettuale, le ore di ricevimento e le ore in cui i docenti sono a disposizione per eventuali supplenze sono state definite come particolari materie, applicando lo stesso concetto per cui la palestra è stata definita come un particolare laboratorio.

Una volta che il programma ha concluso l'acquisizione e la successiva memorizzazione dei dati, svolta dalla funzione `leggiorario()`, esso si preoccupa di creare un orario coerente tramite la funzione `creaorario()`, che mediante una euristica con code con priorità ci garantirà che l'orario verrà generato assegnando gradualmente le ore classe per classe, e non riempiendo una classe per volta. La coerenza dell'orario in fase di generazione è garantita da continui controlli effettuati mediante chiamate alla funzione `correggi()`, che viene utilizzata anche per segnalare eventuali problemi critici.

Per poter eseguire l'attività di generazione con un livello di complessità non eccessivamente penalizzante, in questa fase è necessario utilizzare strutture dati particolari adeguate alle necessità: esse, oltre ad essere per certi versi ostiche e poco duttili, contengono informazioni che nei task successivi non verranno utilizzate. Per questo motivo, a generazione conclusa viene effettuato un *porting*, ovvero una fase di trasferimento dei dati da un set di strutture dati (quello utilizzato nella fase preliminare) ad un altro set di strutture dati (quello che

verrà utilizzato d'ora in poi). In questa fase, le informazioni non necessarie vengono tralasciate, mentre quelle interessanti per le elaborazioni successive vengono riorganizzate in nuove strutture di memoria. La funzione che si occupa di questo compito si chiama, guarda caso, `porting()`.

Arrivati a questo punto abbiamo tutto ciò di cui necessitiamo: un orario coerente da cui partire ed un insieme di strutture dati che ci consente di modificarlo a piacimento e di valutarne la configurazione. Possiamo dunque partire con la fetta più pesante di elaborazione per ottenere un buon orario: si entra nel territorio di *Monte Carlo*, con l'algoritmo *Simulated Annealing*.

L'algoritmo è ampiamente discusso nei paragrafi successivi: al termine della sua esecuzione, viene restituito l'orario migliore (definito dalla funzione `qualita()`) ottenuto dalle varie generazioni. Esso può essere in seguito stampato in formato HTML, rappresentato nelle due tabelle *classi/cattedre* e *professori/classi*.

2.3 I file di input

Come anticipato in precedenza, il primo passo che ci porta a realizzare un orario scolastico è quello dell'acquisizione dei dati di input: in E.N.R.I.C.O, i dati di input si suddividono in due categorie:

- dati relativi alle materie, agli insegnanti, alle cattedre, contenuti nel file *Orario.ESP*;
- dati relativi ai *desiderata* degli insegnanti, contenuti in un insieme di file con estensione *bsb*, utilizzando il formato *[nomedocente].bsb*

Vediamo ora il formato di ognuna delle due categorie di file di input.

2.3.1 Formato del file Orario.ESP

Orario.ESP è un file plain text[1]che contiene nella prima riga un numero naturale N, rappresentante il numero di materie che verranno inserite nell'orario.

Le successive N righe contengono stringhe di caratteri che descrivono il nome di ogni materia. E' da notare che una disciplina svolta sia con ore di lezione in classe (lezione teorica) sia con ore di lezione in laboratorio (lezione pratica) viene conteggiata come due materie distinte. Ad esempio "Informatica" e "Laboratorio di Informatica" vengono considerate due materie differenti.

La riga N+2 esima contiene un numero naturale M, il numero di insegnanti: le successive M righe contengono stringhe che descrivono il nome di ogni insegnante. Lo stesso avviene per le classi (dove figurano anche l'ora di ricevimento e l'ora "a disposizione") e per i laboratori (dove figurano anche le palestre).

Arrivati a questo punto, il file enumera tutte le cattedre nel formato:

[numero ore] [materia] [insegnante] [classe] [laboratorio]

[1]I file plain text sono file di testo codificati in ASCII privi di formattazione. Un file di questo tipo non può contenere ad esempio caratteri colorati o in grassetto, ed ha il vantaggio di essere estremamente portabile. E.N.R.I.C.O. utilizza per ogni attività di Input/Output file plain text.

Nel caso in cui la cattedra di insegnamento sia teorica, il campo laboratorio rimane vuoto.

Inoltre, è doveroso sottolineare che il processo di memorizzazione e di successiva creazione di un orario da zero nel nostro programma può andare a buon fine se e solo se il file *Orario.ESP* contiene un set di dati coerenti tra loro; per tal motivo, esso fallirà se ad esempio nella definizione delle cattedre verrà specificata una materia il cui nome non è stato introdotto nella prima parte dello stesso file.

2.3.2 Formato dei file .bsb

Come è già stato anticipato, i dati relativi alle richieste personali degli insegnanti vengono salvati in file plain text con estensione *bsb*: ognuno di questi file conterrà le informazioni relative ad un singolo insegnante, le quali possono essere inserite con comodità mediante un interfaccia utente scritta in Perl[5].

Attraverso questa interfaccia testuale è possibile inserire moltissime richieste quali il giorno libero, le ore preferite, le ore non preferite, le ore impossibili, i laboratori in cui il docente insegna, che serviranno in seguito a valutare un orario, e sono rappresentate da numeri interi memorizzati consecutivamente all'interno del file secondo un ordine ben preciso.

La scelta di utilizzare un file per ogni insegnante, contrapposta all'utilizzo di un unico file contenente tutti gli insegnanti, viene incontro alle necessità che si possono presentare in atto di modifica dei dati: in questo modo è sicuramente più facile il reperimento di quelli relativi ad un singolo insegnante, ed il suo conseguente aggiornamento.

2.4 Analisi delle strutture dati utilizzate

Analizzando le strutture dati implementate in E.N.R.I.C.O., dobbiamo da subito effettuare una distinzione tra strutture dati utilizzate nella fase preliminare di generazione di un orario scolastico da zero e strutture dati utilizzate nei processi successivi, derivate dalle precedenti mediante il *porting*.

Iniziamo descrivendo le prime, senza fermarci sulla definizione delle singole strutture basilari (a mio modo di vedere di scarso interesse), per soffermarci invece sulla base portante dell'intera fase iniziale che coincide con la struttura dati *heap*: comprenderne il funzionamento e le peculiarità sarà necessario e sufficiente per capire il funzionamento della parte preliminare del programma.

2.4.1 Le code con priorità

Spesso in informatica si creano delle situazioni in cui non è necessario mantenere una sequenza di *record*[2] rigorosamente ordinati per *chiave*[3], bensì è fondamentale solamente raccogliere un insieme di record e reperire celermente quello con

[2]Si definisce *record* un insieme di dati eterogenei. Gli elementi di un record sono detti anche campi, e sono identificati da un nome e da un tipo.

[3]La *chiave* di un record è un sottoinsieme dei campi dello stesso record che ne permette l'identificazione univoca.

chiave più grande (o più piccola, a seconda delle necessità); in seguito, può essere necessario aggiungere record all'insieme già raccolto, e di questa nuova *collezione* elaborare nuovamente l'elemento con chiave più grande.

Una struttura dati appropriata in situazioni simili deve mettere a disposizione operazioni di inserimento di un nuovo elemento nell'insieme e di cancellazione dell'elemento con chiave maggiore: tale tipo di struttura prende il nome di *coda con priorità.*

Gli ambiti di utilizzo delle code con priorità sono vastissimi: si va dai moduli del sistema operativo che gestiscono la ripartizione delle risorse computazionali tra più processi (lo *scheduling*), agli algoritmi di visitazione di alberi e grafi (ad esempio gli algoritmi di Prim e Dijkstra), al possibile impiego negli algoritmi di ordinamento (l'*Heapsort*).

Le code con priorità possono essere implementate in maniere differenti: a seconda dell'implementazione, le varie operazioni su di esse avranno prestazioni diverse e potranno essere adatte a determinate applicazioni, ma non ad altre. E' necessario quindi trovare una sorta di via di mezzo, una situazione di bilanciamento tra le varie operazioni in termini di complessità: un efficiente compromesso è quello che riusciamo ad ottenere, eccezion fatta per l'operazione di unione tra code, utilizzando una struttura dati nota come *heap*.

2.4.2 La struttura dati Heap

L'*heap* è dunque una struttura dati classica che permette operazioni efficienti su una coda con priorità.

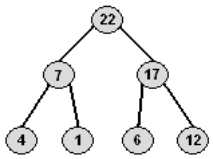
Un heap è in sostanza un albero binario[4] con due proprietà:

1. è **heap-ordinato**, ovvero ogni nodo ha chiave maggiore o uguale alle chiavi di tutti i figli di quel nodo (purchè i figli esistano). Equivalentemente, la chiave di ogni nodo è minore o uguale alla chiave del padre di quel nodo (purchè il padre esista).

2. è memorizzato in un array.

La memorizzazione su array si rende conveniente in quanto è più compatta e non necessita di legami espliciti tra nodi: è infatti sufficiente definire che, dato un nodo di posizione i, il nodo padre si trova nella posizione $\lfloor i/2 \rfloor$, mentre i due figli si trovano nelle posizioni $2i$ e $2i + 1$.

La rappresentazione mediante link espliciti può essere comunque utilizzata in determinate situazioni, per poter implementare una struttura senza doverne conoscere a priori la dimensione massima.



Rappresentazione grafica di un albero binario heap-ordinato.
Studi algoritmici hanno dimostrato che è particolarmente conveniente utilizzare alberi binari completi.

[4] Gli *alberi* in termini informatici sono un'astrazione matematica importantissima per l'analisi di algoritmi dinamici e per l'impiego in strutture dati. Sostanzialmente, un albero è un grafo, ovvero un insieme di vertici (oggetti semplici che possono contenere informazioni) ed archi (connessione diretta fra due vertici) che soddisfano determinate proprietà.

In particolare un albero binario è, definendolo ricorsivamente, un nodo connesso a due alberi binari che vengono chiamati sottoalbero di sinistra e sottoalbero di destra. Questi sottoalberi possono eventualmente essere vuoti.

Algorithm 1 Ripristino dei vincoli dello heap dal basso verso l'alto

k: nodo che origina l'anomalia
p: padre di k

```
while ((k non è radice) && (p < k)) {
    scambia k con p;
}
```

2.4.3 Algoritmi su Heap

Un'importante osservazione che può essere fatta banalmente è questa: dato uno heap di N nodi, il cammino[5] massimo che può essere percorso è $\lg N$, in quanto nell'ultimo livello avremmo $N/2$ nodi circa, $N/4$ nel penultimo e così via.

Tutti gli algoritmi implementati negli heap operano in primo luogo una lieve modifica strutturale che potrebbe violare i vincoli dello heap stesso, per poi riordinare la struttura in modo che tali vincoli risultino ripristinati. In particolare, si distinguono due casi base:

1. inserimento di un nuovo nodo in coda allo heap o aumento della priorità di un nodo già esistente: in tal caso bisogna percorrere lo heap verso l'alto per ripristinare i vincoli. Di volta in volta infatti il figlio, che in maniera anomala risulta avere una chiave maggiore di quella del padre, verrà scambiato con esso, e l'operazione verrà ripetuta fino a quando si incontrerà un padre con chiave maggiore o si raggiungerà la radice. La pseudocodifica di questo processo è illustrata nell'Algoritmo 1. Per quanto precedentemente affermato, il numero di confronti che verranno eseguiti sarà al più $\lg N$.

2. cancellazione del nodo radice o diminuzione della priorità di un nodo esistente: in questo caso è necessario ripristinare i vincoli scorrendo lo heap dall'alto verso il basso, in quanto l'anomalia generata non affliggerà il padre del nodo modificato (la sua priorità resta maggiore di quella del figlio), bensì i due figli del nodo che potrebbero avere una priorità maggiore della sua. Dall'esame della pseudocodifica riportata nell'Algoritmo 2 si deduce che questa tecnica impiega tempo proporzionale a $2 \lg N$ nel caso peggiore, in quanto ad ogni iterazione bisogna effettuare due operazioni: trovare il figlio con chiave maggiore, e decidere se tale figlio deve essere scambiato con il padre.

Comprendere lo heap e le operazioni in esso implementate può risultare semplice, immaginandolo ad esempio la rappresentazione di una gerarchia aziendale, dove i figli di un nodo sono i suoi diretti subordinati, mentre il padre è il loro diretto superiore: la "scalata" di un nodo verso l'alto può rappresentare il progresso nella carriera di un dipendente promettente e

[5] Dato un grafo qualsiasi, il *cammino* è una sequenza di archi che parte da un nodo e conduce ad un altro.

Algorithm 2 Ripristino dei vincoli dello heap dall'alto verso il basso

k: nodo che origina l'anomalia
f1,f2: figli di k

```
while (k non è una foglia[6]) {
    fm = maggiore tra f1 ed f2;
    if (fm > k)
        scambia fm con k;
    else break;
}
```

qualificato, che si ferma solo quando trova un superiore più qualificato. La cancellazione del nodo con chiave massima può invece rappresentare la sostituzione del presidente d'azienda, il cui ruolo viene coperto dal più qualificato dei suoi due diretti subordinati.

Ricordiamo che la nostra coda con priorità è memorizzata in un array: l'inserimento di un nuovo elemento corrisponde quindi ad aggiungere quest'ultimo a fine array, e ad applicare successivamente il ripristino dei vincoli dal basso verso l'alto; allo stesso modo la cancellazione dell'elemento con chiave massima può essere ottenuta correttamente estraendo l'elemento in cima allo heap, spostando l'ultimo elemento in prima posizione, ed applicando infine il ripristino dei vincoli dall'alto verso il basso.

Da questa osservazione si deduce che il tempo massimo per la creazione da zero di uno heap (realizzabile mediante operazioni di inserimento successive) è proporzionale ad $N \lg N$, tempo che in media sarà *lineare* in quanto se consideriamo casuale la sequenza di numeri inseriti, essi tenderanno a dover salire solo di alcuni dei livelli dello heap.

2.4.4 Heap in E.N.R.I.C.O.

Conclusa la fase di acquisizione dei dati di input, abbiamo tutte le informazioni necessarie per poter costruire un orario: insegnanti, classi, laboratori, cattedre.

E' quindi necessario stabilire un criterio con il quale procedere per la creazione sequenziale. Il criterio da noi adottato si basa sul conteggio delle ore che rimangono da assegnare: di volta in volta, fino al completamento della tabella oraria, viene presa in considerazione la classe a cui mancano più ore da assegnare (heap `Frequenze_Globali`), e di essa viene presa in considerazione la *cattedra* con il maggior numero di ore rimanenti da assegnare, secondo la stessa logica (array di heap `Frequenze_Locali`). La materia corrispondente alla cattedra estratta dallo heap della classe viene assegnata alla prima ora disponibile per la classe stessa e per l'insegnante ad essa associato.

Utilizzando questo metodo, performante grazie all'impiego di heap, si garantisce una creazione graduale, che investe le classi in un avanzamento omogeneo; inoltre l'orario generato, privo di lunghe catene di ore con la stessa cattedra,

ottiene già un grado qualitativo discreto e si presenta più malleabile in vista di successive modifiche.

2.4.5 Altre strutture dati

Facciamo ora un rapido escursus sulle strutture dati utilizzate per il Simulated Annealing, costituite essenzialmente da array numerici, strutture ed array di strutture che non necessitano di particolari analisi algoritmiche.

```
struct classi_st {
     int lab[ORE];
     int cat0[ORE];
     int cat1[ORE];
};
```

Contiene le cattedre assegnate per ogni ora ad una classe. Nel caso in cui l'ora sia teorica, il campo `lab` conterrà `NO_LAB`; nel caso in cui l'ora non veda la compresenza di due insegnanti ma la presenza di un solo insegnante, il campo `cat1` rimarrà vuoto.

Un array di `classi_st` di dimensione N_{classi} costituisce la struttura `tab_classi`, ovvero la tabella oraria relativa alle classi.

```
struct prof_st {
     int lab[ORE];
     int cat[ORE];
};
```

Contiene le cattedre assegnate per ogni ora ad un insegnante. Nel caso in cui l'ora sia teorica, il campo `lab` conterrà `NO_LAB`.

Un array di `prof_st` di dimensione $N_{insegnanti}$ costituisce la struttura `tab_prof`, ovvero la tabella oraria relativa ai docenti.

```
struct catt_st {
     int prof;
     int classe;
     int materia;
     int max_ore_consec;
     int marked;
};
```

Contiene la descrizione di ogni cattedra. Il parametro `max_ore_consec` definisce quante ore consecutive di insegnamento può fare l'insegnante, mentre `marked` viene utilizzato esclusivamente come flag nella funzione `qualita()`.

```
int ** ore_pgiornaliere;
int *** ore_mgiornaliere;
```

Queste due strutture contengono, ad ogni istante, il numero di ore giornaliere di lavoro assegnate ad un determinato insegnante in un determinato giorno, ed il numero di ore giornaliere in cui una data materia è stata assegnata in una certa classe un determinato giorno. Anch'esse vengono utilizzate nella funzione `qualita()`.

3 Un orario "ottimo" e parallelo

In questa sezione si studierà il fulcro di E.N.R.I.C.O., inteso come la fetta di programma che richiede il maggior quantitativo di elaborazione.

3.1 Metodi di Monte Carlo

Il metodo di Monte Carlo è un metodo numerico che viene utilizzato per risolvere problemi matematici che, presentando un elevato numero di variabili o una complessità eccessiva, non possono essere risolti facilmente.

Generalizzando, possiamo definire metodo di Monte Carlo un qualsiasi metodo di simulazione statistica, dove la simulazione viene effettuata mediante l'impiego di numeri pseudocasuali. Questi metodi sono in contrasto con i tradizionali metodi numerici, in cui si applicano tipicamente equazioni di vario tipo (ad esempio differenziali parziali) per descrivere un sottostante sistema fisico o matematico: molte delle applicazioni di Monte Carlo simulano direttamente il processo fisico disinteressandosi delle equazioni differenziali che lo descrivono, operando solo su funzioni di densità di probabilità note per il fenomeno da simulare. Note le funzioni di densità, Monte Carlo procede esaminandole casualmente, trattando poi opportunamente i campionamenti per ottenere il risultato cercato. E' chiaro sin da ora che questo metodo necessita di modalità veloci di generazione di numeri casuali uniformemente distribuiti nell'intervallo $[0, 1]$.

Nel calcolo delle probabilità si definisce *funzione di ripartizione* di una variabile casuale X, la funzione $F(X)$ che associa ad ogni elemento x la probabilità dell'evento "la variabile casuale X assuma valori minori o uguali di x":

$$F(x) = P(X \leq x)$$

La funzione di densità di probabilità non è altro che la derivata di quest'ultima funzione:

$$f(x) = \frac{dF(x)}{dx}$$

Le origini di questa metodologia affondano le radici negli anni '40 presso il centro di ricerche nucleari Los Alamos (New Mexico), dove John Von Neumann e Stanislaw Marcin Ulam utilizzarono la simulazione con numeri pseudocasuali per generare i parametri delle equazioni che descrivevano la dinamica delle esplosioni nucleari. Il nome Monte Carlo venne coniato dai due scienziati in riferimento al celebre casinò della capitale del Principato di Monaco, dove roulette, slot machines e sale da gioco possono decretare la fortuna o la rovina di una persona in modo del tutto casuale, riducendo ogni evento all'uscita di questo o quel numero.

Il primo importante impiego di questo metodo si ebbe nella Seconda Guerra Mondiale, utilizzato nello sviluppo della bomba atomica. Oggi, il metodo di Monte Carlo rappresenta l'unica via pratica per risolvere integrali di funzioni a sei o più dimensioni; viene inoltre impiegato per risolvere equazioni differenziali parziali, per predire gli andamenti degli indici di borsa, per approssimare la soluzione di problemi NP-completi in tempo polinomiale.

3.1.1 Dalla pratica alla teoria

Per comprendere il metodo di Monte Carlo, può essere utile iniziare con una analogia legata alla vita quotidiana.

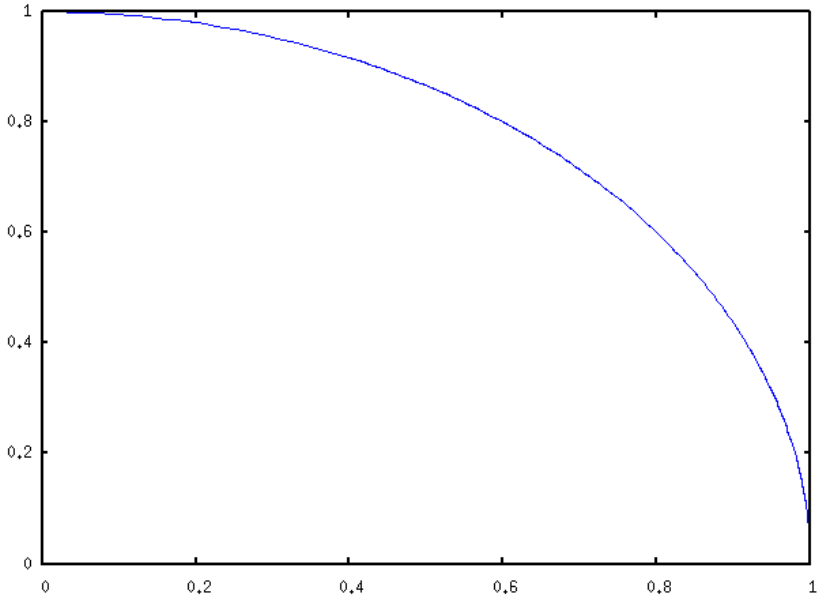
Supponiamo di voler conoscere il valore di π. Sappiamo che l'area di un cerchio di diametro D è $\pi D^2/4$, e che l'area di un quadrato di lato D è D^2. Immaginiamo di esporre alla pioggia una bacinella a forma di cerchio con diametro D, contenuta in un'altra bacinella di forma quadrata DxD. Dopo aver

atteso un paio d'ore, misuriamo la quantità di acqua piovana contenuta nelle due bacinelle: il rapporto tra la quantità contenuta nella bacinella circolare e la quantità totale dovrebbe essere circa:

$$PI = \frac{Q_{CIRCOLARE}}{Q_{TOTALE}} = \frac{\pi D^2/4}{D^2} = \pi/4$$

Moltiplicando per 4 il rapporto ottenuto si ottiene quindi un'*approssimazione* di π.

Possiamo ottenere una stima simile effettuando una *simulazione* utilizzando numeri casuali: consideriamo un piano cartesiano in cui sia tracciato un quarto di circonferenza di raggio 1, contenuto in un quadrato di lato 1:



Consideriamo ora una successione di coppie di numeri (x, y) entrambi distribuiti uniformemente tra 0 e 1. Ognuna di queste coppie rappresenta un punto contenuto nel quadrato; generiamo un numero n di coppie (x, y), conteggiando le m di esse che soddisfano la condizione di appartenere all'area sottesa dal quarto di circonferenza $x^2 + y^2 \leq 1$. Il rapporto m/n ottenuto sarà ancora una volta un'approssimazione di $\pi/4$.

Per dimostrare che questo algoritmo è corretto è sufficiente richiamare il teorema della media del calcolo integrale, il quale afferma che

> *Se una funzione $y = f(x)$ è continua in un intervallo chiuso e limitato $[a, b]$, allora esiste certamente nell'intervallo aperto (a, b), almeno un'ascissa c tale che $I = \int_a^b f(x)dx = (b - a)f(c)$*

Il metodo di Monte Carlo stima il valore I valutando $f(x)$ in n punti scelti in maniera casuale ed uniforme all'interno dell'intervallo $[a, b]$; il valore atteso per $\frac{1}{n}\sum_{i=0}^{n-1} f(x_i)$ è $f(c)$, quindi

$$I = \int_a^b f(x)dx = (b-a)f(c) \approx (b-a)\frac{1}{n}\sum_{i=0}^{n-1} f(x_i)$$

Nel nostro caso, volendo approssimare π, abbiamo posto $b = 1$ ed $a = 0$, e ponendo

$$f(x,y) = \begin{cases} 1 & se\ x^2 + y^2 \leq 1 \\ 0 & altrove \end{cases}$$

abbiamo calcolato $\frac{1}{n}\sum_{i=0}^{n-1} f(x_i, y_i)$.

3.1.2 Componenti fondamentali di un algoritmo di Monte Carlo

L'esempio fatto e le descrizioni sino ad ora date ci consentono di stilare una lista dei componenti fondamentali per la maggior parte delle applicazioni di Monte Carlo:

- *funzioni di densità di probabilità*: un set di funzioni che descrivono il fenomeno fisico (o matematico) preso in esame;
- *generatore di numeri casuali*: è necessaria una fonte di numeri casuali generati uniformemente in un intervallo;
- *regola di campionamento*: uno o più metodi per stabilire la densità di probabilità ottenuta con una certa sequenza di numeri casuali;
- *parallelizzazione*: algoritmi che velocizzino l'esecuzione e/o ne suddividano l'elaborazione su sistemi distribuiti.

3.1.3 Problemi di ricerca dell'ottimo

In realtà, l'esempio prima visto per il calcolo di π appartiene solo ad una classe di problemi risolvibili tramite Monte Carlo, ovvero i problemi di *integrazione*. E.N.R.I.C.O. invece appartiene ad un'altra classe di problemi, i problemi di *ottimizzazione*, che si distinguono in quanto operano su funzioni multi-dimensionali che devono essere minimizzate.

Questi problemi di enorme complessità algoritmica possono essere risolti soltanto mediante procedure stocastiche, che si basano sulle *random walks*, ovvero sull'esplorazione casuale della funzione multi-dimensionale cercando di convergere in zone in cui la funzione diminuisce.

Il più famoso problema di ottimizzazione è quello del commesso viaggiatore (noto nella comunità scientifica come *TSP*, *travelling salesman problem*): dato un numero di città, ed il costo per passare da una qualsiasi città ad ogni altra,

determinare il percorso più economico per visitare ogni città una sola volta e tornare alla città di partenza.

Come E.N.R.I.C.O., il TSP appartiene a quell'insieme di problemi classificati come *NP* ("Non-deterministic Polynomial time"), ovvero problemi decisionali risolvibili solo in tempo polinomiale, ed è un problema di *ricerca dell'ottimo*.

L'ottimo, nella fattispecie, è il percorso più economico che attraversi tutte le città una sola volta tornando alla partenza. Nel nostro programma si può considerare ottimo un orario che soddisfi rigorosamente i vincoli precedentemente posti. In realtà, è palese l'impossibilità di costruire immediatamente un orario ottimo: la numerosità delle variabili in gioco (insegnanti, laboratori, cattedre, vincoli obbligatori, etc) rendono proibitivo un approccio euristico e esageratamente dispendiosa una ricerca a forza bruta (brute force).

3.1.4 Simulated Annealing

In informatica, *brute force* è il nome dato ad un algoritmo generico di risoluzione dei problemi che si basa sull'enumerazione progressiva di tutte le possibili soluzioni per il dato problema, che vengono quindi trattate in funzione di ciò di cui si necessita.

Matematicamente parlando, applicare brute force equivale al trovare tutte le possibili permutazioni, cosa del tutto improponibile nel nostro programma (equivarrebbe al trovare tutti i possibili orari): è quindi necessario seguire una strategia diversa, che ci consenta di individuare l'ottimo indipendentemente dalla vastità dello spazio di ricerca.

Per fare questo, abbiamo realizzato ed adattato alla nostra problematica un algoritmo noto come *Simulated Annealing*. Il nome è ispirato al processo in metallurgia di annealing, che consiste nel portare a temperatura di fusione un materiale, per poi farlo raffreddare in maniera lenta e controllata. Così facendo, il materiale trattato vedrà aumentare le dimensioni dei propri cristalli, ottenendo una struttura regolare priva di difetti.

Il calore infatti induce gli atomi a muoversi rispetto alla posizione iniziale (un minimo locale dell'energia interna), facendoli vagare casualmente in stati di più alta energia, ove riescono più facilmente a muoversi; al calare della temperatura, le energie atomiche diminuiscono, rendendo agli atomi difficoltoso il movimento. Il raffreddamento lento aumenta la probabilità che essi riescano a raggiungere una configurazione con un livello di energia interna inferiore rispetto a quello iniziale, dando al materiale l'opportunità di raggiungere uno stato di minimo energetico, che corrisponde alla sua forma cristallina.

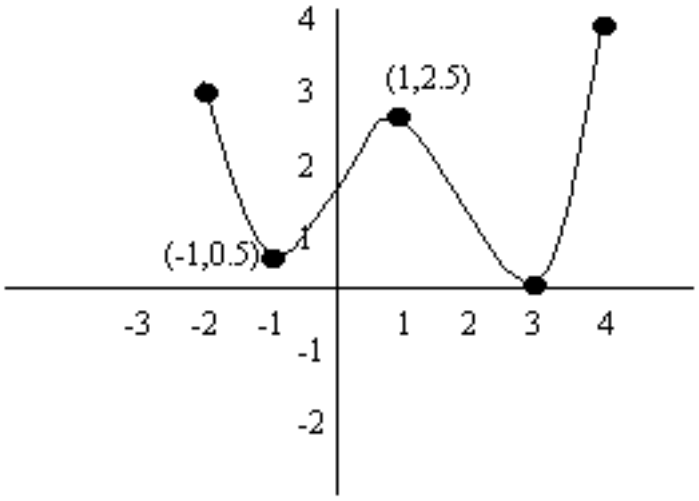
Analogamente al processo fisico, il simulated annealing (*SA*) considera l'ottimizzazione di un problema equivalente al raggiungimento di un determinato stato di un materiale; il valore di una funzione obiettivo per una particolare soluzione corrisponde all'energia associata ad un particolare stato energetico, e la soluzione ottima per il problema equivale al raggiungimento di uno stato di minimo energetico.

3.1.5 L'Algoritmo

Il SA è un algoritmo iterativo. Ad ogni iterazione, l'algoritmo sostituisce la soluzione corrente con una nuova soluzione "vicina" alla precedente e scelta casualmente. Se il valore della funzione obiettivo per questa nuova soluzione è inferiore rispetto alla precedente, quest'ultima viene accettata. In caso contrario, la nuova soluzione viene accettata con una probabilità definita da una funzione $P(s, s', T)$ dove s è la soluzione corrente, s' la soluzione nuova e T è la *temperatura* corrente.

Questa funzione ha dei requisiti fondamentali: non deve restituire zero nel caso in cui $s' > s$, in quanto è necessario poter accettare anche soluzioni peggiori (con uno stato energetico maggiore) delle precedenti. In questo modo, si evita di rimanere bloccati in un *minimo locale*, ovvero una configurazione energetica che non è il minimo globale (l'ottimo), ma è inferiore a quella di tutti i suoi "vicini".

In questo grafico di esempio, il minimo locale è $(-1, 0.5)$ mentre il minimo globale è $(3, 0)$.



Inoltre, al diminuire della temperatura T, $P(s, s', T)$ per $s' > s$ deve tendere a zero, riducendo la possibilità di "scalare dei picchi", ovvero accettare configurazioni energetiche superiori a quella corrente, sino a raggiungere un certo valore ε per il quale possa accettare solo configurazioni $s' < s$.

Nel nostro caso

$$P(s, s', T) = e^{\frac{-(s-s')}{T}}$$

Per implementare un algoritmo SA, dobbiamo definire 4 parametri che avranno un peso decisivo in termini di prestazioni e di funzionamento del programma:

- rappresentazione delle soluzioni: nel nostro caso una soluzione è una combinazione di ore assegnate formanti un orario scolastico legale;

- definizione di una funzione costo: ovvero di una funzione che, data una soluzione, stabilisca e quindi quantifichi l'energia ad essa associata. E' realizzata in E.N.R.I.C.O. mediante la funzione `qualita()`;

- spostamento in una soluzione *vicina*: dato un orario corrente O, definiamo un nuovo orario O' vicino ad esso se O' è ottenibile da O mediante uno scambio di due ore scelte casualmente. Lo scambio può in realtà coinvolgere più di due ore (si parla dunque di *catena di scambi*) in modo tale da essere condotto a buon fine nonostante particolari configurazioni che ne pregiudicherebbero la legalità;

- realizzazione di una funzione di raffreddamento: questo parametro può condizionare molto le performance dell'algoritmo. La temperatura iniziale deve essere sufficientemente alta per garantire che discese e salite della funzione abbiano pressochè la stessa probabilità. Inoltre, il raffreddamento non deve essere né troppo veloce (con il rischio di fermarci troppo presto in un minimo locale) né troppo lento. Questa fase presuppone dunque la conoscenza di qualche valore di stima della differenza $s' - s$ tra soluzioni vicine. La nostra scelta è stata questa: all'inizio di ogni ciclo esterno T viene inizializzato a
$$T_0 = \frac{1}{k^{0.18}}$$
con k numero dei cicli esterni; successivamente, ad ogni singola iterazione T assume la natura di una semplice funzione geometrica:
$$T_{i+1} = T_i * 0.999995$$

Lo pseudocodice di Simulated Annealing implementata in orario è riportato in Algoritmo 3.

3.2 La parallelizzazione

E' stato già accennato in precedenza al fatto che lo spazio entro cui deve operare il nostro programma rappresenta il vero limite alla buona riuscita di una sua esecuzione.

Esattamente in questo scenario entra in gioco la potenza del calcolo parallelo: l'immaginario dominio N-dimensionale da esplorare, ovvero l'insieme di tutti i possibili orari, viene suddiviso in n processi. Questi processi dunque, mediante SA, si occuperanno di esplorare una zona limitata del dominio alla ricerca di minimi.

In realtà, una suddivisione rigorosa tra processi non è possibile in quanto a priori non è nota la natura del dominio in termini di cardinalità; questa problematica è stata affrontata definendo una funzione di distanza alquanto banale

> *Due orari O ed O' hanno distanza d se essi hanno d ore assegnate in maniera differente.*

Algorithm 3 Simulated Annealing in E.N.R.I.C.O.

s, s': orari
t: temperatura
q, q': qualità di un orario
u: numero casuale $\in U[0,1]$

s = orario iniziale
for ($k = 1$; $k < N_{est}$; $k{+}{+}$) {
    $t = \frac{1}{k^{0.18}}$
    q = qualita(s)
    $i = 0$
    while ($i < N_{iter}$) {
        s' = scambio_casuale(s)
        if (qualita(s') < qualita(s) or $u \leq e^{\frac{s-s'}{t}}$) {
            $s = s'$
            $i = 0$
        } else $i = i + 1$
        $t = t * 0.999995$
    }
}

ed utilizzando un generatore di numeri pseudo-casuali parallelo, ovvero un generatore che garantisca ad ogni processo la possibilità di generare una sequenza di numeri pseudo-casuali con *periodo*[7] sufficientemente elevato, riproducibile e non correlata con quelle contemporaneamente generate negli altri processori. Nella fattispecie, è stato adottato un generatore Lagged Fibonacci da noi sviluppato.

In seguito, è stato necessario definire una disciplina secondo la quale regolare l'esecuzione del programma. Tipicamente gli algoritmi di Monte Carlo optano per due diverse soluzioni:

1. esecuzione del programma sino al raggiungimento di un certo valore, entro il quale la soluzione ottenuta può considerarsi ottima;

2. esecuzione del programma sino all'esaurimento del tempo macchina disponibile, al termine del quale viene presa in considerazione la miglior soluzione ottenuta;

Per E.N.R.I.C.O. abbiamo adottato una disciplina della seconda specie: il programma resta in esecuzione su ogni processo per un numero N fissato di cicli esterni. N può essere virtualmente infinito, ma l'esecuzione può essere arbitrariamente bloccata in ogni momento mediante un'interfaccia grafica in Java realizzata appositamente. Essa si preoccupa di segnalare al processo principale

[7] Il periodo di un generatore di numeri pseudocasuali rappresenta la quantità massima di numeri generabili senza ripetizioni.

(noto anche come processo 0) la volontà dell'utente di terminare il programma: questo processo si occuperà poi di avvisare gli altri $n - 1$ processi mediante un *messaggio broadcast*[8].

Il programma prevede inoltre una sorta di monitorizzazione grafica. Ad ogni ciclo esterno i processi si sincronizzano e comunicano al processo principale la loro miglior qualità: questa informazione sarà utile per tracciare un grafico temporale che metterà in luce il progressivo miglioramento di orario ottenuto nei vari processi.

In conclusione, l'orario migliore ottenuto può essere esportato in formato HTML tramite la funzione `stampastrutturadati()`, che viene invocata a fine esecuzione del programma. In qualsiasi momento dell'esecuzione è comunque possibile ottenere l'orario migliore in formato HTML cliccando sul pulsante "Stampa tabella orari" dell'interfaccia grafica precedentemente citata.

Esempi di esecuzione del programma nei suoi vari aspetti verranno presentati in sede di prova orale con l'ausilio di un proiettore.

[8]Il broadcast è una comunicazione che coinvolge tutte le entità di un particolare raggruppamento logico di processi. Nella fattispecie, il processo 0 invia un'informazione che tutti gli altri processi riceveranno.

Riferimenti bibliografici

[1] Introduction to Parallel Computing `http://www.llnl.gov/computing/tutorials/parallel_comp/`

[2] Introduction to Parallel Programming `http://www.mhpcc.edu/training/workshop/parallel_intro/MAIN.html`

[3] Message-Passing Interfaces Standard `http://www-unix.mcs.anl.gov/mpi/`

[4] Gentoo Linux Official Site `http://www.gentoo.org`

[5] Perl Official Site `http://www.perl.org`

[6] Algoritmi in C++ di Robert Sedgewick (Addison-Wesley, 2003)

[7] Lagged Fibonacci Generator da Wikipedia `http://en.wikipedia.org/wiki/Lagged_Fibonacci_generator`

[8] Scuola di Calcolo Parallelo `http://scp.volterraprojects.org`