Parallel Processing

di Luciano Macera

prima parte

A partire da questo numero inizia su MCmicrocomputer una nuova serie di articoli dedicati alla programmazione parallela. Di stampo principalmente didattico, gli articoli tratteranno in modo abbastanza esaustivo tutte le tematiche riguardanti questa tecnica di programmazione ormai possibile anche su molti personal computer in circolazione.
Parleremo, certo, anche del multitasking di Amiga (la prima macchina «personal» con un VERO sistema operativo multitask), ma le nostre discussioni resteranno, finché possibile, rigidamente ancorate a schemi del tutto generali, utilizzando linguaggi di programmazione parallela molto vari e tra loro diversificati. Faremo largo uso anche dell'OCCAM, il linguaggio di programmazione del transputer, col quale mostreremo «sul campo» alcune risoluzioni a problemi tipici della programmazione parallela

Parallelismo

I lettori ancora non «multitasking» magari si staranno già chiedendo perché tutto quest'affanno riguardo questa tecnica di programmazione a loro, ahinoi, poco nota. Del resto in C, in Pascal, in Basic (orrore!!!), si riesce a programmare di tutto. Volendo anche in linguaggio macchina magari senza neanche un assemblatore simbolico. È vero, come diceva adp «tutto quello che si può ben dire si può ben fare», e con qualsiasi mezzo di calcolo (anche una succosa Macchina di Turing), si può calcolare qualsiasi... algoritmo calcolabile. Ma non corriamo troppo. Anzi, non usciamo fuori tema: di calcolabilità MC se n'è già occupata svariati numeri fa nella riuscita rubrica «Appunti di Informatica» e dunque non è il caso di tornare nuovamente sull'argomento.

Dicevamo: perché programmazione parallela? Le risposte a quest'interrogativo sono molteplici, e vorremmo procedere per gradi. Innanzitutto la programmazione parallela permette di risolvere più facilmente problemi «intrinsecamente paralleli». Pensate ad esempio ad un sistema operativo che deve rivolgersi simultaneamente a più dispositivi per ricevere e dare informazioni di vario tipo: dati, segnali, sincronismi. Normalmente tutto è risolto «a colpi di interrupt», magari anche nidificati l'uno nell'altro fino ad ottenere un funzionamento più o meno affidabile dell'intero «accrocco». Ma quanto è più bello, più elegante, più efficiente, più produttivo, risolvere il problema scrivendo un processo per ogni funzione svolta dal nostro sistema operativo e lasciare che questi «vadano in parallelo», con le loro esecuzioni e interruzioni indeterministicamente realizzata dallo stesso kernel gestore del multitasking?

Così avremo un processo che attende caratteri dalla tastiera, un processo che asserve le richieste su disco, un altro processo che provvede alla stampa (eventualmente appoggiato da altri processi che implementano uno spooler intelligente), un altro ancora ha a che fare con gli output su video, ed altri ancora che implementano i rimanenti dispositivi utilizzabili dagli altri processi utente. E così la stampa magicamente diventa assolutamente indipendente dalle operazioni su disco o dagli eventuali ulteriori comandi impartiti da tastiera. Analogamente le elaborazioni video (dal semplice scrolling e spostamento di finestre a vere e proprie animazioni in tempo reale) non devono fare a cazzotti con le rimanenti attività della macchina quali quelle relative alle altre periferiche.

E poi in un sistema operativo multitask la potenza totale di calcolo può facilmente aumentare passando da architetture uniprocessor (in cui tutto il parallelismo di cui sopra è simulato attraverso meccanismi di time sharing e sospensione processi in attesa) ad architetture multiprocessor dove le varie attività svolte, e non solo quelle di sistema ma soprattutto quelle «utente», sono eseguite in parallelo su più processori contemporaneamente.

Ma se le architetture multiprocessor, almeno per quanto riguarda le utenze «personal», possono sembrare oggi ancora fantascienza, anche il semplice multitask implementato su singolo processore può dare risultati sorprendenti.

Immaginate ad esempio di dover eseguire una complessa ricerca su un vostro database che, probabilmente, terrebbe impegnata la macchina per alcuni minuti. Una ricerca di questo tipo implica un continuo accesso ai dischi per trovare le informazioni desiderate. Come si sa, in ogni sistema di calcolo le periferiche rappresentano *sempre* veri e propri colli di bottiglia: non s'è ancora vista, infatti, una unità a dischi

più veloce di un processore. In pratica analizzando quel che succede nel dominio del tempo in un calcolatore monotask che esegue continui accessi al disco, vedremmo che la CPU è minimamente impegnata a dare ordine al device e per lo più attende l'esito delle varie operazioni ad esso richieste. Tra la richiesta di una di queste operazioni (ad esempio: Lettura del blocco 4565...) e l'esito di tale richiesta la CPU potrebbe fare qualcos'altro. Se il sistema è multitasking, potrebbe eseguire altri calcoli in tutti i tempi in cui dovrebbe «inutilmente» aspettare la terminazione delle operazioni disco. È così che mentre il nostro HD fa «tractrac» cercando le informazioni richieste, nulla ci vieta di lanciare anche il nostro WP per cominciare a scrivere una relazione e magari altrettanto parallelamente lasciare la nostra scheda Fax pronta a ricevere messaggi dalla linea telefonica. Operazioni che in un sistema rigidamente monotask avremmo dovuto effettuare in tempi successivi impiegando complessivamente un tempo superiore. Certo che se invece di una situazione simile (del tutto normale, però) avremmo la necessità di calcolare un enorme spread sheet e contemporaneamente disegnare una porzione dell'insieme di Mandelbrot (situazione per nulla normale, però) su un sistema multitask-uniprocessor non ci guadagneremo nulla non riuscendo a sfruttare tempi morti come nel caso precedente. Anzi, per la verità, impiegheremo addirittura un tempo superiore dato che comunque il multitasking ha un suo costo in termini di tempo macchina utilizzato dal sistema per la sua stessa implementazione (il cosiddetto overhead).

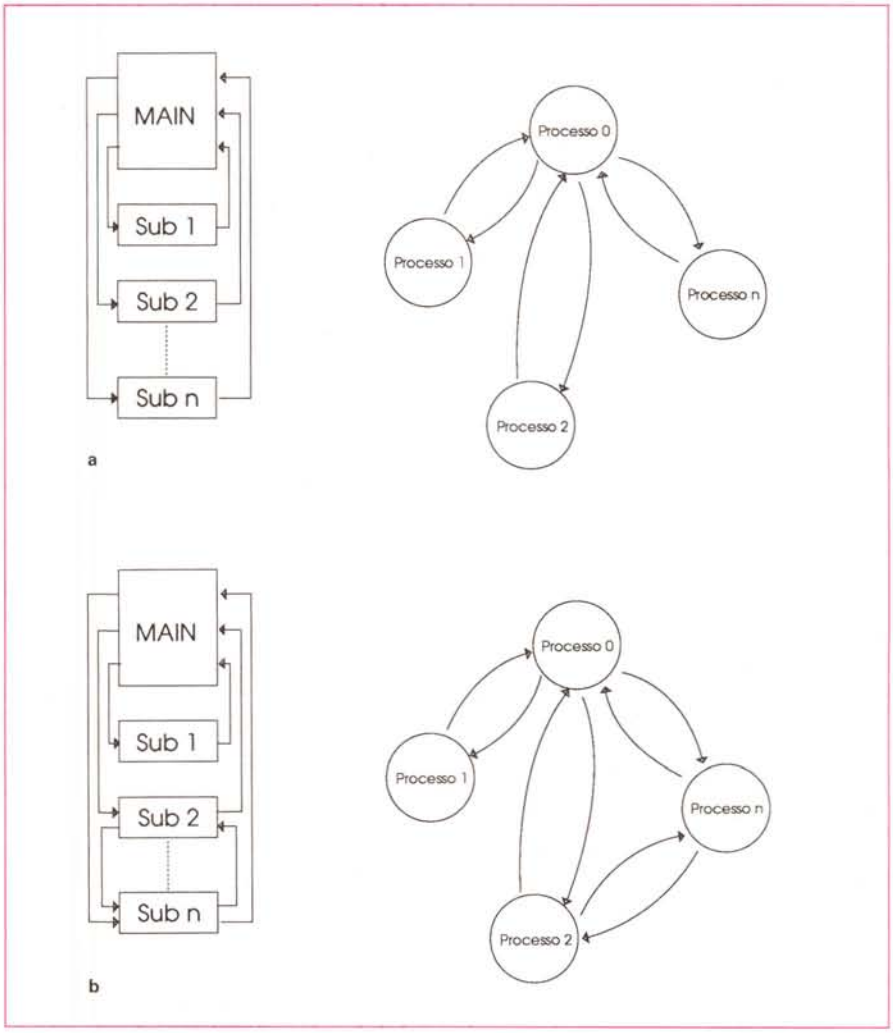

Figura 1 - A confronto un programma monotask con uno multitask. Le subroutine possono essere implementate da processi indipendenti.

Zoccoli e CPU

Aprendo un computer, la visione di un po' di zoccoli vuoti per integrati presenti sulla mother board ci richiama subito alla mente la possibilità di espandere la memoria del sistema o al più di aggiungere un coprocessore matematico alla CPU.

Sarebbe bello invece trovare un giorno all'interno dei personal computer alcuni zoccoli, diciamo una decina o anche più, pronti ad accogliere altri processori per espandere la potenza di calcolo del nostro sistema. Il tutto, magari, in maniera assolutamente trasparente: al momento del boot il sistema farebbe un bel check dei processori installati in quel momento (eliminando via software eventualmente i processori guasti) configurandosi opportunamente come macchina mono o multiprocessore. Inoltre tale «dinamicità» potrebbe permanere anche a computer funzionante eliminando anche nel bel mezzo di una elaborazione CPU difettose.

Ogni processore in funzione preleverebbe dalla lista dei processi in stato di pronto un processo per eseguirlo fino a successiva sospensione dovuta a richiesta di I/O o a «quanto di tempo scaduto». Così in ogni istante se «n» sono i processi funzionanti in quel mo-

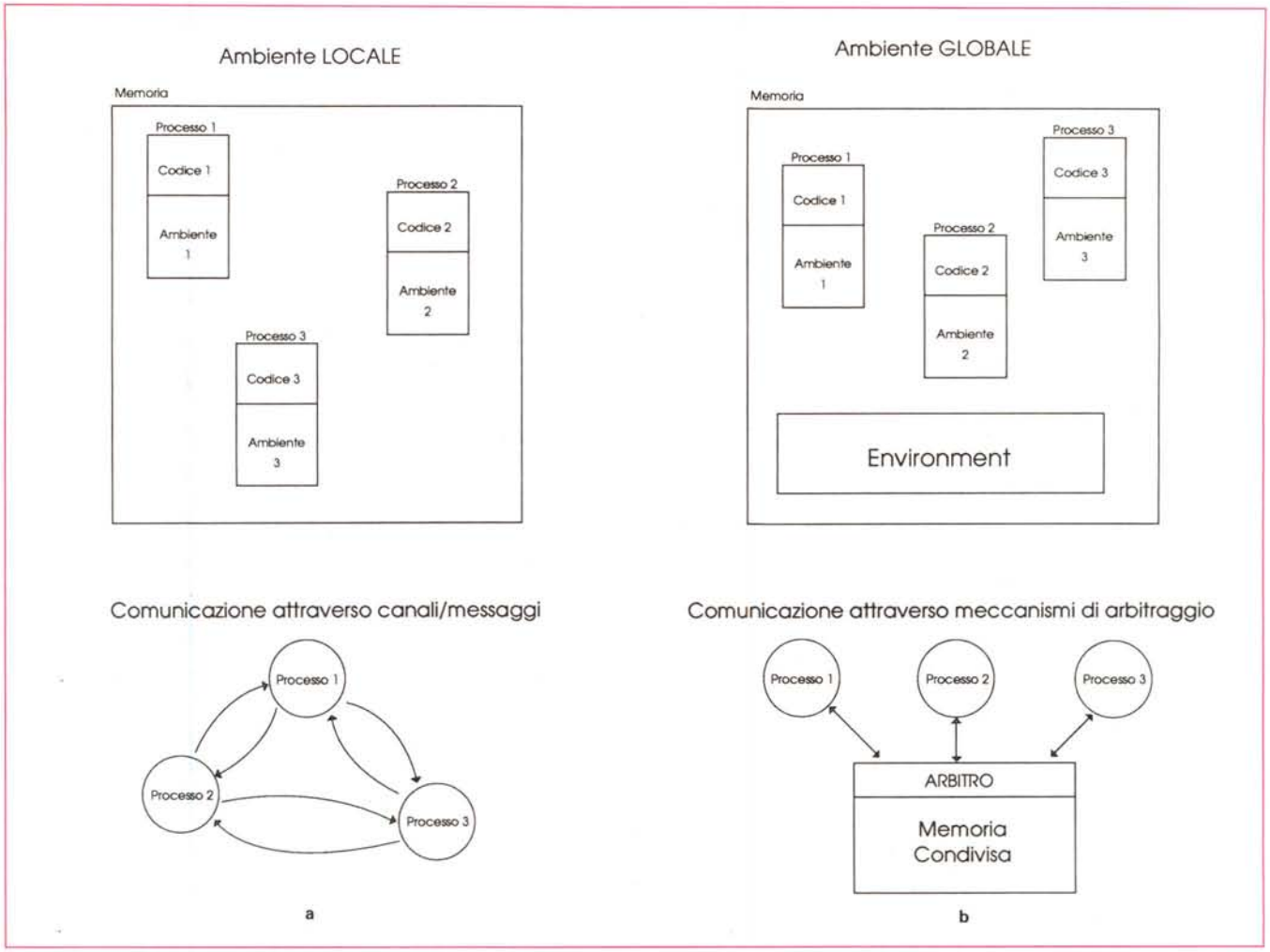

Figura 2 - Comunicazione ad ambiente locale e ad ambiente globale.

mento, fino ad «n» saranno i processi in esecuzione parallela. Inoltre ogni processore implementerebbe a sua volta un proprio multitask (in questo caso a parallelismo simulato) in modo che se «n» sono le CPU attive in quel momento ed «m» i processi da eseguire su ogni processore saranno eseguiti «m/n» processi contemporaneamente. In pratica potenza di calcolo REALMENTE configurabile secondo le effettive necessità.

Multitasking vs Monotasking

Grazie alla programmazione strutturata è ormai possibile scrivere i propri programmi non solo come insieme di linee di codice logicamente collegate da salti condizionati e non, ma più elegantemente come un insieme di moduli autonomi (le procedure) con le quali dialogare per mezzo dei meccanismi del passaggio dei parametri. Al punto che nei moderni linguaggi di programmazione appena un po' evoluti ogni procedura ha un proprio ambiente locale nel quale riconosce ed utilizza proprie variabili e strutture che nulla hanno a che spartire con analoghi riferimenti simbolici visibili in altre zone del programma. In più, nella programmazione appunto sequenziale, è di solito riconoscibile nella struttura di un programma il cosiddetto «main» che invoca varie procedure e funzioni e, per l'appunto, la definizione di quest'ultime.

Nel passaggio alla programmazione multitask, può essere conveniente (in alcuni casi necessario) vedere le singole procedure come veri e propri processi attivi, in attesa di ricevere sui propri canali i dati da elaborare e da rispedire a chi ha richiesto l'elaborazione. In pratica al posto di una procedura di sort possiamo immaginare un processo ordinatore che riceve in ingresso l'array da riordinare e restituisce in uscita (al committente) l'array ordinato.

Sparisce poi, in un certo senso, anche il concetto di «main» (o programma principale). Tutti i processi che formano il nostro programma multitask sono tra loro equivalenti: tutti attendono sulle loro porte di ingresso gli input per fornire degli output.

In figura 1A è mostrata tale analogia tra le due tecniche di programmazione mono e multitask. In figura 1B troviamo anche il caso in cui le procedure siano invocate anche l'un l'altra (oltre che dal «main»): nella soluzione multitask altrettanto è possibile che i processi cooperino tra loro prima, eventualmente, di restituire un risultato al processo committente.

I vantaggi di una soluzione simile sono praticamente ovvi: in una gestione

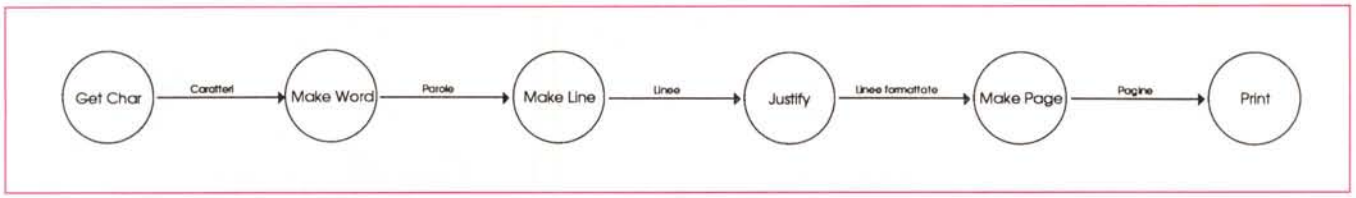

Figura 3 - Esempio di Spooler di stampa multitask con funzionamento pipeline.

«main-procedure» il chiamante ferma la sua elaborazione fino a quando la procedura non termina la sua, restituendo i risultati. Anche se tali risultati non servono immediatamente per una successiva elaborazione. Nella soluzione multitask, sempre se si vuole, è invece possibile demandare un determinato compito più o meno gravoso ad un processo concorrente e continuare la propria elaborazione invocando, ad esempio, altri servizi.

E se pensate che un meccanismo simile non porti poi benefici quando il parallelismo è solo simulato (nei computer uniprocessor) basta pensare alla soluzione ben più allettante dei computer multiprocessor in cui otterremo davvero un incremento di velocità potendo realmente eseguire più calcoli contemporaneamente. Banalmente è vero che un programma di per sé sequenziale non «corre di più» su un'architettura multiprocessor mentre il discorso è ben diverso per i programmi multitask.

Forme di comunicazione

Esistono fondamentalmente due tipi di comunicazione tra processi in esecuzione parallela. Ad ambiente locale e ad ambiente globale. Nel primo caso ogni processo ha soltanto una propria zona di memoria privata nella quale realizza il suo ambiente (costanti, variabili, array, strutture, liste), nel secondo caso oltre a questa esiste una ulteriore zona di memoria condivisa da tutti i processi in esecuzione.

In base a questa destinazione cambia, conseguentemente, il meccanismo di comunicazione tra processi. Meccanismo che, comunque, è strettamente necessario dal momento che a ben poco servirebbe un sistema multitask in cui i vari processi non fossero in grado di comunicare tra loro.

Vedremo maggiormente in dettaglio tutto questo nelle prossime puntate di questa rubrica. In questa sede anticiperemo solo la differenza fondamentale tra le due forme sopra indicate. In figura 2A è rappresentata schematicamente la memoria utente di un computer multitask in cui i processi cooperano ad ambiente locale. In pratica per ogni processo in esecuzione è riservata una zona di memoria contentente il codice da eseguire e un'ulteriore zona di memoria (sempre per ogni processo) in cui sono mantenute tutte le variabili (e le altre strutture) di quel processo. Nella cooperazione ad ambiente globale (figura 2B) oltre a questo troviamo anche un Environment comune a tutti i processi in esecuzione. Nel primo caso non disponendo di zone di memoria condivise la comunicazione interprocess avviene attraverso porte e canali di comunicazione utilizzando primitive di scambio messaggio tipo:

SEND (Porta, Messaggio)

che spedisce il messaggio indicato sulla porta indicata, e

RECEIVE (Porta, Messaggio)

che riceve dalla porta indicata il messaggio in arrivo depositandolo nella variabile (o struttura) indicata come secondo parametro. Nel caso dell'ambiente globale la comunicazione avviene, come detto, utilizzando zone di memoria condivisa. In questo caso è però necessario arbitrare in qualche modo l'accesso a queste zone di interscambio per evitare ad esempio che più processi la utilizzino per scrivere oppure che un processo destinatario inizi a leggere prima che un altro processo mittente abbia terminato di scrivere o viceversa. I meccanismi per implementare tale arbitraggio sono vari e si va dalla semplice sospensione del multitasking per tutta la durata dell'operazione di lettura/scrittura a più sofisticati meccanismi di locking, semafori o monitor per gestire, ad esempio, condivisione di device. Ma di tutto questo, come detto, ne riparleremo nelle prossime puntate.

Multitasking e pipeline

Per concludere questa breve introduzione alle tecniche multitask, vogliamo indicarvi un altro utilizzo di questo tipo di programmazione. I processi di cui è formato un programma possono anche avere un funzionamento pipeline (catena di montaggio) in cui ogni modulo è collegato ad un modulo successivo e ad un modulo precedente, eccezion fatta, ovviamente, per il modulo iniziale e finale della catena.

In figura 3 è mostrato uno spooler di stampa «evoluto» con questo tipo di funzionamento. Il primo processo preleva da un opportuno canale di input (ad esempio un file su disco o lo stesso flusso di uscita di qualsiasi altro programma) i caratteri da stampare e li spedisce, l'uno dopo l'altro, al processo successivo che forma le parole. In pratica raggruppa insiemi di caratteri separati da almeno uno spazio.

Le parole così formate vengono passate al processo MakeLine che, come dice il suo nome, forma una prima bozza della linea da stampare occupandosi di unire tra loro tante parole fino ad arrivare ad una lunghezza massima pari alla dimensione della linea di stampa.

La linea grezza così formata è passata al processo Justify che esegue la giustificazione rispetto ai margini e la centratura della linea rispetto alla pagina di stampa. Così le linee ormai giustificate possono essere passate all'ultimo processo «attore» che raccoglie le linee giustificate per formare le pagine di stampa complete, eventualmente, di numerazione, titolo di giro, ecc.

Per finire le pagine sono inviate al processo di stampa che semplicemente si occupa dell'interfacciamento con la stampante vera e propria.

Ovviamente mentre è in stampa il documento «x» gli altri processi possono essere occupati nel medesimo istante a «lavorare» il documento «x-1», «x-2», ecc. ecc. Proprio come si conviene ad una catena di montaggio.

Sul prossimo numero cominceremo a trattare un po' più da vicino alcune delle argomentazioni introdotte questo mese. Arrivederci...

MC