

Rivisitazione degli strumenti informatici per il calcolo e la simulazione

L'informatica si è sviluppata come materia di studio durante gli anni cinquanta e sessanta, in un periodo di grande sviluppo tecnologico. Gran parte dell'interfaccia "informatica", altre scienze e attività umane stavano cambiando rapidamente: il computer entrava sempre più nella vita di molta gente comune.
La crescita e lo sviluppo dell'informatica sono stati paralleli a quelli della Fisica Nucleare. **Dall'inizio della cosiddetta "era nucleare" il computer è stato in grado di fornire previsioni accurate e puntuali sull'intensità del fenomeno nucleare**. Dal progetto degli armamenti nucleari alla progettazione e al controllo dei più sofisticati reattori nucleari il computer è diventato indispensabile.
Oggi molti studiosi di computer si interessano di programmazione e soluzione di problemi.
Subito dopo lo sviluppo della fisica nucleare, negli anni sessanta, venne quello della scienza aerospaziale.
Molte applicazioni già realizzate in campo nucleare sono state prontamente tradotte in applicazioni aerospaziali.
Ne fanno parte ad esempio i calcoli per la guida, il controllo e l'analisi degli effetti radioattivi. Molti problemi di affidabilità nella costruzione di reattori nucleari si sono presentati anche nell'ingegneria aerospaziale.
I problemi scientifici e ingegneristici "reali" non hanno mai soluzione in forma chiusa (formale): vanno quindi risolti col calcolo numerico. Dato che da essi spesso dipende la vita delle persone vanno risolti con un altissimo grado d'accuratezza (precisione).
Tali problemi "numerici" (computazionali) abbracciano praticamente la maggior parte dei problemi della fisica (nucleare e non), dell'ingegneria e di altre scienza applicate. Per ottenere risultati accurati nei calcoli si deve tralasciare il meno possibile e occorre procedere a piccoli, piccolissimi passi per trovare la convergenza con la precisione richiesta.
Per questo motivo è necessaria tanta, anzi tantissima, potenza di calcolo.
Mentre una volta la tecnologia informatica era appannaggio soltanto delle più grosse organizzazioni, oggi siamo al culmine della richiesta di scienza e tecnologia e **praticamente ogni ufficio tecnico effettua regolarmente calcoli con l'analisi agli elementi finiti** o per la verifica di impianti secondo i più svariati algoritmi numerici, **eseguiamo normalmente la simulazione di fenomeni complessi prima di realizzare costosi prototipi**, *facciamo calcoli per determinare quale sia il punto migliore dove scavare un pozzo petrolifero, o quali siano le mosse migliori per conseguire o massimizzare un vantaggio o contenere uno svantaggio* (con le teorie dei giochi), *progettiamo sistemi complessi di vario tipo*. Oggi usiamo il computer persino per divertirci comprimendo video a risoluzioni sempre più alte e vogliamo poterlo fare in tempi accettabili.
L'informatica è poi "dentro" alle macchine, che sempre più hanno comportamenti "intelligenti": sanno raccogliere dati dal mondo esterno per elaborarli in tempo reale, informandoci se qualcosa non va e prendendo provvedimenti d'emergenza (pensiamo alla macchina che frena da sola, o ai sistemi guidati dal satellite). Insomma, non ne possiamo più fare a meno.
Sono nate persino parole nuove come "***meccatronica***".
Le crescenti richieste di precisione e velocità in queste applicazioni hanno dato impulso a quello che oggi è l'inarrestabile progresso tecnologico del computer.
Esistono anche altri campi dell'informatica, non soltanto quelli delle applicazioni numeriche, ma più in generale, esistono programmi di simulazione, emulazione e intelligenza artificiale, oltre alle più svariate applicazioni non numeriche.
Giorno dopo giorno si moltiplicano i nostri bisogni e, nascono e poi si sviluppano soluzioni sempre nuove, sia hardware (macchine sempre più potenti) che software (applicazioni che sfruttano tutta la potenza del nuovo hardware).



Hardware

Quando ero studente, ahimè più di venti anni fa, il panorama hardware e software era profondamente diverso: l'hardware si divideva in super computer per applicazioni scientifiche/militari, mainframe per grandi organizzazioni civili e home computer in grado di far girare a malapena qualche videogioco in bassa risoluzione (320x200), la famosa rubrica del telefono, un editor di testo, un minimo di contabilità personale. In un epoca senza telefoni cellulari, era già un buon metodo per stupire amici e vicini di casa.
Avevo il mitico Commodore 64, che insieme all'Apple II condivideva la gloriosa CPU ***MOS Technology 6502***. Il primo aveva una versione speciale (*6510*) con alcuni registri hardware aggiuntivi per il registratore a cassette e, per la configurazione di memoria.
Le istruzioni più semplici prendevano 2 cicli di clock, quelle più complicate 7, in pratica: **Commodore 64 e Apple II avevano una potenza di calcolo di una piccola frazione di MIPS**[1] (circa 0.25).
Questo era l'hardware che ci si poteva comprare, pagandolo non poco se pensiamo che all'epoca un C64 costava come uno stipendio. *Soltanto adesso mi rendo conto dei sacrifici che fecero i miei per regalarmelo*...
Nel 1982, a parte Commodore e Apple, **il neonato PC aveva una CPU Intel 8086/8088**. Nonostante i suoi 4.77 Mhz di frequenza di clock, le prestazioni (**0.33 MIPS**), non erano molto superiori a quelle delle CPU considerate "della vecchia guardia" a 8 bit. Erano sì migliori rispetto al 6502 e allo Z-80 (altro concorrente dell'epoca) ma quest'ultimi non sfiguravano sulle macchine che equipaggiavano. Il PC, **anche nella versione a 8 Mhz spuntava al massimo 0.75 MIPS**.
Il nuovo e agguerrito contendente che equipaggiava la seconda generazione di home computer, **ATARI ST** e **Commodore AMIGA**, il **Motorola 68000** (*e successivi, fino al 68040*) aveva caratteristiche innovative che non avevano rivali per le CPU in commercio dell'epoca. Grazie inoltre alla genialità dei progettisti di quelle macchine, mi piace ricordare il compianto Jay Miner, le già potenti CPU Motorola erano affiancate a una superstruttura unica per i chipset di quei tempi: su Amiga, il controller della RAM Fat Agnus capace di muovere blocchi di memoria di grandi dimensioni per quei tempi (512, 1024 e 2048 kb) a grande velocità , Denise era un chip per la grafica fino a 640x512 con 4096 colori (incredibile per quei tempi, parliamo del 1985), con supporto agli sprites, sub-pixel scrolling e double buffering gestiva anche Mouse e altri dispositivi di puntamento (Joystick usato nei videogiochi), Paula era il chip audio con 4 canali indipendenti PCM 8 bit capace di generare frequenze da 20Hz a 29kHz, gestiva anche il floppy disk e la porta seriale. Sia la CPU che queste unità, erano dotate di Direct Memory Access (DMA), con Agnus come DMA Controller.
Era il massimo della tecnologia che si poteva avere in casa in quegli anni: un hardware raffinatissimo, almeno 10 anni avanti rispetto ai primi PC, anzi, quest'ultimi le presero di santa ragione fino all'avvento del microprocessore 80386 o il nuovo 80486 su tante applicazioni (grafica, applicazioni per musicisti, montaggio video, e ovviamente i videogiochi).
Nessuno comprava un PC per i videogiochi: la CPU non era potente abbastanza per muovere tutto in modo fluido. Le schede grafiche accelerate di oggi non esistevano per i PC. Il microprocessore 68000 ci abituo, grazie soprattutto alle successive versioni, e alla competizione agguerrita di quegli anni contro Intel a vedere il numero di MIPS moltiplicarsi nettamente, tuttavia la necessità di velocità era ancora ben al di sopra di quanto le CPU dell'epoca potessero fornire.
Arriviamo così agli anni '90 e la situazione hardware, per noi utenti, è "desolante" tant'è che negli uffici tecnici regnano sovrane le Workstation grafiche (macchine dedicate, dotate di hardware speciale e molto costose) mentre i PC vanno bene soltanto per le applicazioni da "ufficio" come il Word-Processing, Spreadsheet & Database client-side (il server dell'epoca non è un PC né ha un sistema operativo Windows).
Nel frattempo, il 486 aveva guadagnato, incremento di clock dopo l'altro una velocità intorno ai 50 MIPS nelle ultime versioni (partiva da 20; su questa CPU Intel fece un ottimo lavoro) tuttavia, tale velocità non era ancora lontanamente paragonabile alla velocità dei computer "per il calcolo", quelli cioè che

[1] MIPS = Million Instructions Per Second (milioni di istruzioni al secondo). Misura la velocità di un microprocessore in numero di istruzioni (assembly) del processore in esame al secondo. Il numero di MIPS misura quindi la velocità della CPU e non dell'esecuzione dei programmi dato che un programma svolge milioni, anzi, parecchi miliardi di istruzioni prima di terminare, tuttavia... Una CPU con più MIPS sarà comunque in grado, in generale, di far girare il codice più velocemente.

trasformavano noi appassionati in piccoli bimbi appoggiati col naso, fuori dalla vetrina della pasticceria. Tanto per fare qualche esempio, vediamo qualche supercomputer dell'epoca.

Cray Computers



Il **Cray-1**: nel **1976** (anno in cui fu realizzato) costava otto milioni di dollari. Installato nei laboratori di Los Alamos (*foto a lato*), le prestazioni erano fuori da ogni parametro per i tempi: **160 MFlops**[2] e ben **8 MByte di memoria centrale** (il primo PC aveva 32KByte, nel 1981...).
Il **Cray X-MP** (**1982**). **Primo modello multiprocessore**, seguito dal **Cray-2** (con **1.9 GFlops** di picco era oltre 10 volte più potente del Cray-1, con **2 GByte di memoria**) dopo 3 anni; nel 1982 avevamo 32/64 Kbytes di memoria centrale.
Nel 1993 usciva il **Cray-T3D** con **4096 processori Alpha a 150 Mhz, 615000 MIPS e 615 GFlops**.

E gli altri?

Intel aveva un mostro da **300 GFlops**: l'**Intel Paragon**, costituito da **4000 CPU I860** (*siamo però già nel 1992*).
C'erano poi altre soluzioni, più "abbordabili" ma erano comunque costosissimi sistemi per grandi organizzazioni, ecco alcuni esempi: **Alliant FX (8 CPU 68020 da 94 MFlops, 1985)**, **Bull DPS/90 (108 MFlops, 1986)**, **Convex C240 (200 MFlops, 1987)**, **IBM 3090 (800 MFlops, 1988)**, **Dec Alpha (200 MFlops, 1993)**.
Di fascia inferiore e ancora più economici (al costo di non meno di $100.000) c'erano:
IBM RS/6000 (100 MFlops, 1992), **IRIS di Silicon Graphics SGI (100 MIPS nel 1990)** e poi, **INDIGO** (SGI) e i computer della **famiglia HP-Unix** (con **CPU PA**) e **HP-Apollo** (con **CPU Motorola 68020, 68030 e 68040**, la stessa dei Macintosh e Amiga fino alla prima metà degli anni '90, prima di passare al PowerPC e poi ai processori Intel) oppure lo **Sparc** ("soltanto" 60 MFlops, ma $40000 nel 1993).
Mi piace ribadire il concetto che nel 1990 i più fortunati di noi avevamo una CPU Intel 80486 (1 MFlops?), 150 volte meno potente del Cray-1 e, 2000 volte meno rispetto al Cray-2 del 1982.

Il processore Pentium

Il 1993 ha segnato una svolta con l'uscita delle **CPU x86 di quinta generazione**: il processore **Intel Pentium**. *E' stata la prima CPU per i desktop a superare i 60 MFlops (e a raggiungene addirittura 200 MFlops nella sua ultima versione, a 200 Mhz).* ***Ha quindi un grande valore anche dal punto di vista storiografico poiché ha eguagliato e superato le prestazioni del Cray-1***.
Il processore Pentium ha dato quindi inizio a una nuova era per i computer desktop.
Per la prima volta si poteva acquistare un calcolatore "veloce" e realmente economico (un Pentium 133 Mhz si poteva acquistare con pochi milioni di lire nel 1995) con una potenza paragonabile a quella di un supercomputer (purtroppo di venti anni prima ma questo non importava):il PC si "emancipava", schiodandosi da rubriche telefoniche e altri programmi più divulgativi che realmente utili.
Finalmente si parlava di CAD (anche 3D), sostituendo così le costose Workstation; si potevano già sviluppare applicazioni di calcolo numerico "interessanti" e di simulazione e intelligenza artificiale..
Non si tratta più soltanto di "applicazioni per scienziati pazzi" (quelli dei reattori nucleari) ma diventavano applicazioni "per tutti" (per molti, almeno).
Da lì in poi è cambiata anche l'idea che con il computer ci si possa divertire, non soltanto con i videogiochi, ma sviluppando tanti dei nostri interessi: **con l'IBM PC XT sarebbe stato impossibile ad esempio**

[2] MFLOPS: Million FLOating-Point per Second, un milione di operazioni in virgola mobile al secondo. È un'unità di misura per la CPU avente meno condizioni del MIPS per il confronto dei calcolatori perché si basa sulle operazioni anziché sul numero di istruzioni. E' utile per valutare i calcolatori che devono far girare programmi che sfruttano operazioni floating-point (FP) come le applicazioni scientifiche / calcolo numerico.

comprimere un video in DivX, persino una musica con MP3; anche con i 486, i video che circolavano, erano su finestrelle minuscole che ci voleva la lente di ingrandimento e non erano certo in formato Full HD come oggi!

I sistemi operativi poi, erano "cose da guru" con interfacce "a caratteri", oggi con finestre e mouse tutti li usano!

I programmi di scacchi battevano soltanto i "polli", adesso invece il 99.99% della popolazione (rimangono fuori Gary Kasparov e altri due o tre fortunati) non ha speranza contro i migliori programmi per PC!

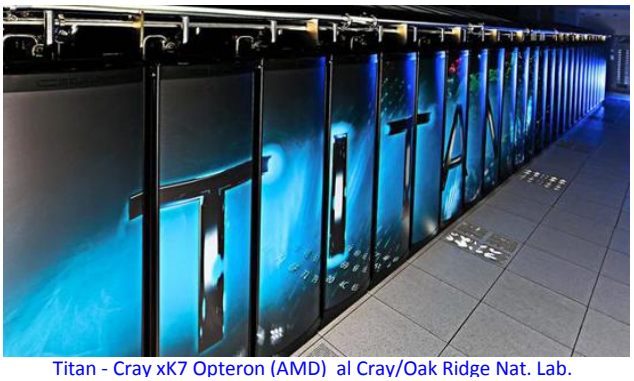
Il primo Pentium ha quindi aperto una breccia sul muro. *Generazione dopo generazione le CPU hanno raggiunto una potenza per cui CPU da 3 GHz multicore sono addirittura "sprecate" per usare Office*.

Così come il Pentium raggiunse il Cray-1, **il Pentium III e il corrispondente AMD *Thunderbird* da 1 Ghz di clock raggiunsero la velocità di un Cray-2** (vent'anni dopo).

In estrema sintesi, una tabella con alcuni sistemi presi come riferimento tecnologico di un epoca:

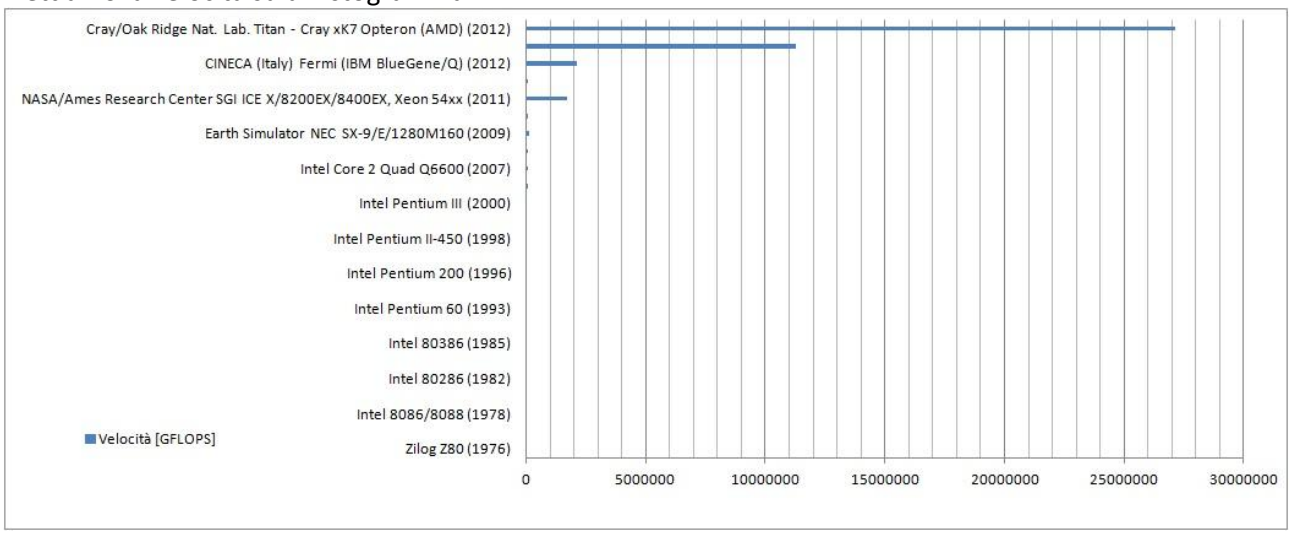
Produttore	System/CPU	Anno	Clock [Mhz]	MIPS	GFLOPS	Ambito
Zilog	Z80	1976	2.5	-	0	Home Computer/Embedded
Cray Computers	Cray-1A	1977	80	-	0.16	super-computer
Intel	8086/8088	1978	4	0.33 MIPS	0	PC
Motorola	68000	1980	8	1 MIPS	0	Workstation
Intel	80286	1982	12.5	2.6 MIPS	0	PC
Cray Computers	Cray X-MP	1983	-	-	0.8	super-computer
Intel	80386	1985	16	5 MIPS	0	PC
Intel	80486	1989	25	15 MIPS	0.001	PC
Motorola	68040	1990	25	20 MIPS	0.035	Workstation
Intel	80486DX	1992	50	-	0.03	PC
Intel	Pentium 60	1993	60	-	0.06	PC
Intel	Pentium 90	1994	90	-	0.09	PC
Intel	Pentium 200	1996	200	-	0.2	PC
Intel	Pentium II-233	1997	233	-	0.23	PC
Intel	Pentium II-450	1998	450	-	0.45	PC
Intel	Pentium III	1999	500	-	1	PC
Intel	Pentium III	2000	1000	-	2	PC
Intel	Pentium 4-520 Prescott	2004	2800	-	5.6	PC
Intel	Core 2 Quad Q6600	2007	2400	-	38.4	PC
Intel	Core i7-950	2009	3066	-	49	PC
Earth Simulator	NEC SX-9/E/1280M160	2009			131072	super-computer
Intel	Core i7-980	2011	3300	-	80	PC
NASA Research Center	SGI ICE X/8200EX/8400EX	2011			1731836	super-computer
Intel	Core i7-3770	2012	3400	-	109	PC
CINECA (Italy)	**Fermi (IBM BlueGene/Q)**	**2012**			**2097152**	**super-computer**
RIKEN AICS (Japan)	Sparc64 VIIIfx	2012			11280384	super-computer
Cray/Oak Ridge Nat. Lab.	Titan - Cray xK7 Opteron (AMD)	2012			27112550	super-computer

Mi piace rimarcare l'ottimo risultato del sistema "**Fermi**" che abbiamo qui al **CINECA** (Casalecchio di Reno, Bologna, sito web: http://www.cineca.it). CINECA compariva già nella lista dei ***top 500***, nelle prime posizioni, anche prima di questo importante aggiornamento tecnologico.

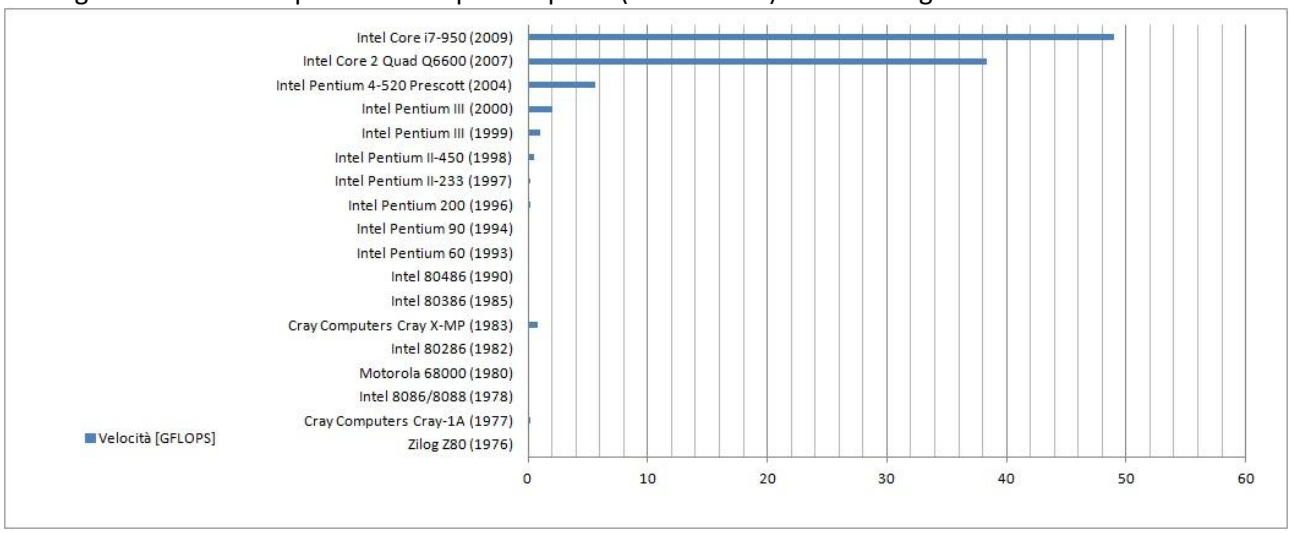


Titan - Cray xK7 Opteron (AMD) al Cray/Oak Ridge Nat. Lab.

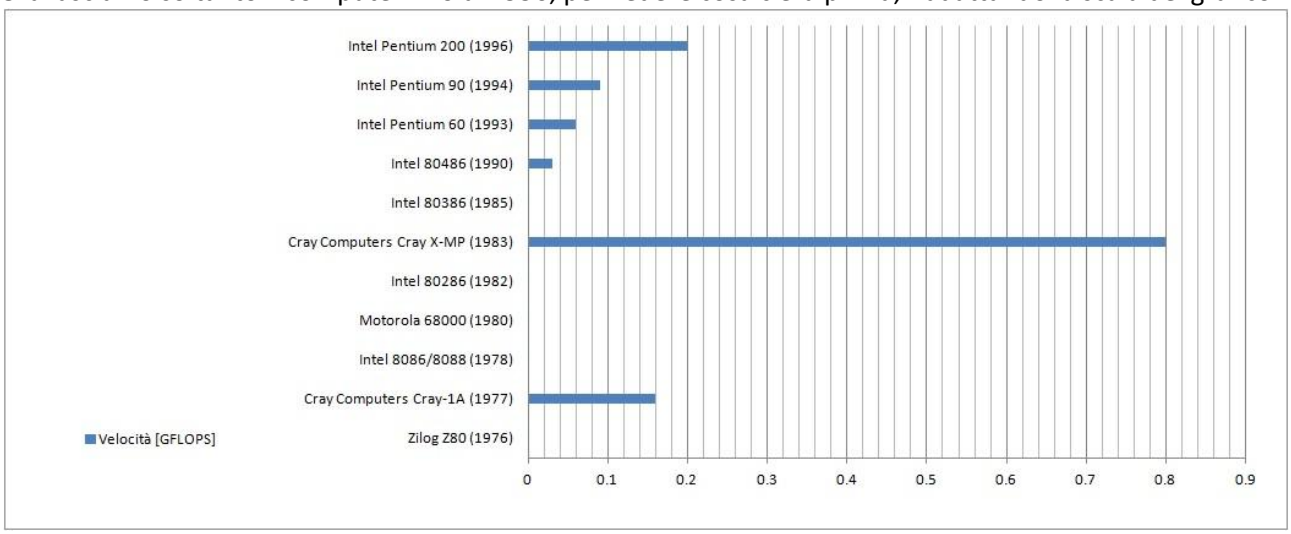
Mettiamo la velocità su un istogramma.



Ora togliamo dalla lista i più recenti supercomputer (fino al 2009). Altrimenti gli altri "restano invisibili".



Ora lasciamo soltanto il computer fino al 1996, per vedere cosa c'era prima, riadattando la scala del grafico.



Dai tre grafici emerge che:

1. per ogni periodo considerato, supercomputer e PC contemporanei non sono paragonabili;
2. **il paragone si può fare tra un PC e un super computer di almeno 20 anni prima**;
3. la potenza di calcolo delle CPU cresce circa in modo esponenziale;
4. **la potenza di calcolo del Cray-1A è stata raggiunta tra il 1995 e il 1996**.

E adesso?



Bene. Mi conforta sapere che il mio **Intel Core i7** è capace di macinare numeri all'incredibile velocità di **50 GFlops** (*davvero notevole per un computer piazzato sulla mia una scrivania, visto che **Earth Simulator occupa da solo l'intero piano di un grattacielo**, come vediamo nella foto a destra*), ma... come facciamo a sfruttarne la potenza di calcolo?

Occorre precisare che nessuna CPU sarebbe, ancora oggi, capace di una simile velocità di calcolo e i progettisti dei supercomputer, da subito, hanno capito che dovevano ricorrere al calcolo "parallelo" attraverso l'attivazione contemporanea di più unità di calcolo.

Ciò implica un software appositamente programmato per suddividere un problema in quante più parti elaborabili "in parallelo" sul maggior numero di unità di calcolo disponibili del sistema. Tale approccio si chiama appunto: **calcolo parallelo** vale a dire **l'esecuzione simultanea del codice su diverse unità di calcolo allo scopo di aumentare le prestazioni del sistema**..

Questo non è un articolo sul calcolo parallelo anche se **la scelta di uno strumento rispetto a un altro può facilitare il ricorso a questa tecnica**.

In questo articolo farò alcune considerazioni su alcuni strumenti di sviluppo (di soluzioni, programmi di calcolo) che ho utilizzato e utilizzo tutt'ora quando ho necessità computazionali: ci sarà anche il tempo di eseguire un piccolo benchmark.

Se gli algoritmi fossero scritti per far ricorso al calcolo parallelo, il guadagno prestazionale non dipenderebbe tanto dal fatto di usare questo o quello strumento ma, semplicemente, i tempi di elaborazione sarebbero divisi (circa) per un uguale fattore, a parità di unità di calcolo attivate.

Possiamo quindi escludere dalle nostre considerazioni il pur auspicabile ricorso, in tutti casi, del calcolo parallelo.

Essenzialmente le soluzioni per scrivere programmi che fanno calcoli potrebbero essere:

1. ricorso a **programmi commerciali** (simulazione/FEM, applicazioni specifiche, con le dovute limitazioni comprendiamo anche i fogli elettronici);
2. **linguaggi di programmazione** (BASIC, Pascal, Lisp, Fortran, C/C++, ecc.) e relativi strumenti di sviluppo come **compilatori** o **interpreti**;
3. a metà strada tra le prime due troviamo certi **ambienti/applicativi di calcolo/simulazione programmabili**[3] .

Se la soluzione 1) è praticabile, significa che qualcuno ha già sviluppato un software per noi, e ciò sarebbe ovviamente auspicabile.

Per questo motivo non analizziamo questa opzione.

Nella maggior parte dei casi un programma non è disponibile oppure esiste ma non siamo disposti ad acquistarlo.

Restano quindi gli altri due casi.

[3] Sono programmi che oltre a mettere a disposizioni funzioni specifiche, prevedono linguaggi proprietari, ad esempio Matlab, o ancora certi fogli elettronici che incorporano interpreti o compilatori. Molti programmi CAD inoltre offrono questa possibilità oppure mettono a disposizione librerie richiamabili da linguaggi di programmazione.



Linguaggi di programmazione

In riferimento ai linguaggi di programmazione si può dire che questi, dagli anni '50 ad oggi, hanno seguito la straordinaria evoluzione dell'hardware disponibile nel corso degli anni.
Questo ha determinato la fortuna di alcuni e il disuso di altri.
I linguaggi di programmazione si suddividono in: linguaggi di **basso livello** e di **alto livello**.

Linguaggio Macchina (e Assembly)

Costituisce il livello più basso di linguaggio. E' l'unico che la CPU può comprendere.
Vantaggi: legato alla struttura fisica del calcolatore (CPU), il più veloce ed efficiente perché il programmatore comunica direttamente con il microprocessore.
Svantaggi: programmi lunghi, difficili da codificare (scarsa efficacia nel descrivere gli algoritmi), difficile messa a punto e *debugging* dei programmi, manutenzione del codice assai complicata.
La velocità è il nodo più critico di molte applicazioni ancora oggi ma lo era a maggior ragione con le modeste CPU di tanti anni fa: in passato si era praticamente obbligati al ricorso all'Assembly in tantissime situazioni al punto che un programmatore non poteva non conoscerlo. Oggi fortunatamente l'uso dell'Assembly è limitato a pochissime situazioni.

Linguaggi

Il linguaggio di programmazione **è un linguaggio intermedio fra il linguaggio macchina e un linguaggio naturale attraverso il quale descrivere algoritmi**: li descrive con una ricchezza espressiva comparabile a quella dei linguaggi naturali senza però essere ambiguo, deve essere quindi assolutamente "rigoroso".
L'uso dei linguaggi di programmazione è il seguente:

1. si parte da un problema;
2. attraverso un'analisi si determina un algoritmo;
3. si scrive un programma (con un linguaggio di programmazione) che implementa l'algoritmo.

Il linguaggio di programmazione, non essendo linguaggio macchina, **necessita di una traduzione per diventare una sequenza di istruzioni che siano comprensibili e quindi eseguibili dal sistema**.
Compare quindi la figura del "***traduttore***" da linguaggio di programmazione (*la lingua in cui è espresso il programma sorgente, cioè la descrizione del nostro algoritmo scritta nel linguaggio di programmazione*) al linguaggio macchina (che si concretizza nel programma "oggetto").
Il file "sorgente" è generalmente costituito da più file, tranne che nel caso di algoritmi elementari dove può essere costituito da un file unico.
Se il linguaggio è di "alto livello", significa che è particolarmente orientato all'uomo, più descrittivo, vicino al linguaggio naturale e quindi più comprensibile al programmatore, svincolato dall'architettura della macchina e quindi adatto per CPU diverse.
Oppure di "basso livello", particolarmente orientato alla macchina e per architetture/CPU specifici.
Esempi di linguaggi di alto livello sono il Pascal, il BASIC, il Logo... Esempi di linguaggi di basso livello sono l'Assembler (o Assembly). **Caratteristiche dei linguaggi di basso livello è la corrispondenza 1:1 (o prossima a questo rapporto) tra istruzioni del linguaggio e istruzioni in linguaggio macchina**.
Da questo punto di vista il linguaggio C è singolare perché rappresenta un linguaggio a metà strada: molte delle sue istruzioni hanno una corrispondenza 1:1 con le istruzioni "macchina" tuttavia, pur non essendo particolarmente descrittivo, è in grado di esprimere algoritmi con grande efficacia pur rimanendo veloce ed efficiente praticamente come un programma scritto con l'Assembler.
Storicamente **il traduttore è un Compilatore o un Interprete per i linguaggi di alto livello** oppure un **Assembler per il linguaggio di basso livello** (impropriamente si confonde infatti l'assembly con il linguaggio macchina tuttavia oramai ci siamo rassegnati).
Compilatore, ***Interprete*** o ***Assembler*** si occupano di rendere disponibile un programma oggetto (o una sequenza di istruzioni in linguaggio macchina nel caso dell'interprete) che corrispondono all'algoritmo descritto nel codice sorgente attraverso il linguaggio di programmazione: lo fanno però in modi diversi.



Compilatore

Il compilatore è un programma che riceve un intero programma sorgente in Input e restituisce in Output il programma oggetto.
Alla compilazione segue il **linking** che collega più moduli oggetto prodotti dal compilatore in un unico programma eseguibile. Il linker (il programma che si occupa del linking) è anche necessario per risolvere i riferimenti esterni a ogni modulo non risolti dal compilatore (quindi include le librerie necessarie).
Ad esempio un codice sorgente in C espresso dai file di programma sorgente mod1.c, mod2.c, ... modN.c, compilato diventa programma oggetto mod1.obj, mod2.obj,...,modN.obj.
Grazie al linker diventa un file direttamente eseguibile (es. mod.exe), in linguaggio macchina.
I primi compilatori generavano programmi oggetto in linguaggio macchina di scarsa qualità rispetto al codice che un buon programmatore avrebbe potuto scrivere con l'assembler.
Oggi i compilatori sono anche in grado di ottimizzare il codice: si ottengono programmi più veloci da eseguire in primo luogo perché sono convertiti in linguaggio macchina, inoltre, è migliorata notevolmente la qualità del programma oggetto prodotto (il risultato è un programma in assembly di qualità elevata).
Sono esempi di linguaggi compilati: COBOL, Pascal, Fortran, C/C++.

Interprete

L'interprete legge ogni singola "frase" dal programma sorgente, nell'ordine del flusso del programma, la trasforma in una sequenza di istruzioni macchina ed esegue: traduzione ed esecuzione sono contestuali, quindi non esiste un programma oggetto ma è l'interprete che s'interfaccia alla CPU.
I programmi interpretati sono più lenti (*leggono una riga di codice, la traducono e la eseguono, poi passano all'altra, la traducono e la eseguono... dall'inizio alla fine del programma*).
Il costo della traduzione rende l'applicazione più lenta. In compenso la messa a punto del programma solitamente è migliore perché è più facile individuare la riga di programma dove si verifica un errore (*anche se le cose sono molto migliorate con l'introduzione di efficaci strumenti di debug*).
Esempi di linguaggi interpretati sono: il BASIC, Prolog, R.
Per molti linguaggi interpretati sono stati creati compilatori per migliorare le prestazioni.
In questo modo possono diventare anche loro programmi oggetto e quindi eseguibili, oppure, un compilatore può genera uno pseudo-codice (P-CODE).
Il P-Code non è codice nativo (*linguaggio macchina*) ma piuttosto dati che costituiscono un linguaggio intermedio (di basso livello) che viene eseguito da un programma in linguaggio macchina: veloce ma comunque meno performante del codice nativo.
Se è possibile scegliere tra un compilatore in codice nativo e un P-Code è bene scegliere il primo. Quest'ultimo sarà comunque preferibile a un interprete.
Il QuickBASIC, VisualBASIC (fino alla versione 6.0) sono esempi di linguaggi interpretati per cui si sono messi a disposizione compilatori (P-Code).
Python è un linguaggio **pseudocompilato**: un interprete si occupa di analizzare il codice sorgente e, se sintatticamente corretto, di eseguirlo. In Python, non esiste una fase di compilazione separata (come avviene in C, per esempio) che generi un file eseguibile partendo dal sorgente.



Le Macchine Virtuali

Esiste un modello "ibrido" in cui il programma sorgente viene compilato in un linguaggio intermedio (es: bytecode) e sarà mandato in esecuzione da **un software che s'interpone tra il programma nel metalinguaggio e il sistema operativo**: una **macchina virtuale**.
Se da un punto di vista pratico è assimilabile al P-Code, da un punto di vista filosofico è profondamente diverso: il P-Code sono istruzioni che fanno realizzare a un eseguibile il nostro algoritmo, nella macchina virtuale invece il bytecode è il vero e proprio linguaggio di programmazione in cui il nostro sorgente viene convertito.
Esempi:

1. Java viene compilato in un linguaggio chiamato **bytecode**. E' un linguaggio di livello molto basso, poco più alto del linguaggio macchina. E' indipendente dall'architettura ed è il linguaggio comprensibile dalla Java Virtual Machine (JVM);
2. .NET - il **Common Intermediate Language** (**CIL**) è un linguaggio "assembly" direttamente eseguibile dal Common Language Run-Time (**CLR**), la macchina virtuale di .NET di Microsoft. VisualBASIC.NET, Visual C++.NET e C# compilano il sorgente in CIL, indipendentemente dall'architettura.

La portabilità è garantita dalla disponibilità della VM per quella specifica CPU e sistema operativo.
Gli estimatori delle macchine virtuali pensano che sia il modo per compilare un programma su tutte le macchine e per tutti i sistemi operativi; i detrattori di queste, pensano che sia invece il modo di rendere tutto "interpretato".
Il dubbio è in effetti lecito: le prime JVM erano infatti molto lente.
Già dagli anni '80 però le VM sono dotate di un compilatore detto **JIT**[4].
In pratica una volta che il programma è stato messo a punto e corretto da errori viene:

1. compilato in bytecode/CIL, diventando "codice oggetto" platform-independent;
2. contestualmente all'esecuzione, bytecode (o CIL) attraverso il compilatore JIT diventano linguaggio macchina per una piattaforma specifica.

La compilazione JIT risulta veloce, praticamente istantanea, poiché il linguaggio "origine" è già di basso livello.
Questa compilazione in due fasi assicura portabilità del codice (punto 1) e prestazioni, perché **il programma non viene "emulato" dalla macchina virtuale, ma attraverso la compilazione Just In Time, diviene linguaggio macchina al momento dell'esecuzione**.
Il tempo (molto breve) della compilazione JIT è l'unico prezzo da pagare per questo tipo di soluzioni.
Personalmente non ho mai programmato in Java ma uso .NET e devo dire che la velocità che si ottiene è convincente rispetto ad altri programmi compilati in codice nativo.
*Rispetto a un programma compilato, se il compilatore non è ottimizzato (**è un fatto che la maggior parte del codice è compilato ancora per 80x386, e quando va bene per architettura i686**) .NET ottiene prestazioni anche superiori*.
Principalmente i vantaggi che spingono verso queste soluzioni, sono la portabilità del codice dovuta alla virtualizzazione, che consentono quindi di scrivere codice che dovrebbe in teoria essere "più longevo" e "sicuro" dato che la macchina virtuale si occupa anche della gestione in memoria degli oggetti.

[4] Un compilatore **J**ust-**I**n-**T**ime o **JIT** permette un tipo di compilazione, conosciuta anche come traduzione dinamica, con la quale è possibile aumentare le performance dei sistemi di programmazione che utilizzano il bytecode, traducendo il bytecode nel codice macchina nativo in fase di run-time.



Il confronto tra alcune soluzioni

Quale linguaggio scegliere?

Non esiste la soluzione migliore da tutti i punti di vista: abbiamo visto ad esempio che l'ideale per velocità è il linguaggio macchina, ma nel corso degli anni i programmatori hanno cercato di evitarlo come la peste perché troppo complicato.
Entrano pertanto in gioco altri fattori, eccone alcuni.

1. **Requisiti prestazionali**: per molte applicazioni la scelta è limitata dal fatto che un fattore critico di successo sono le prestazioni oppure il fatto che tali prestazioni si debbano garantire anche in condizioni "avverse" (poca memoria disponibile, hardware obsoleto, ecc.); un sistema operativo tanto per essere chiari non può essere scritto con un linguaggio interpretato (tra l'altro l'interprete è un programma che necessita di un S.O. per "girare").
2. **Disponibilità legate alla piattaforma o alla tecnologia che si vorrebbe impiegare**: capita spesso che una certa tecnologia o un certo compilatore non esistano per una piattaforma specifica; pensiamo ad esempio ai sistemi *embedded*: spesso le uniche alternative sono l'assembly e se va bene un compilatore per il linguaggio C (tipico nei microcontrollori industriali).
3. Aspetti legati alla **produzione del software** (***codifica***, ***sviluppo***, ***messa a punto***/***debugging*** e ***manutenibilità***): si richiedono ai progettisti tempi "risicati"; il mercato richiede spesso aggiornamenti che si realizzano con l'uscita di nuove versioni. Per questi motivi certe scelte divengono vincolanti. Non c'è quasi mai il tempo di riscrivere da zero il codice, magari in un altro linguaggio. Il codice deve quindi risultare riutilizzabile/riadattabile/manutenibile con semplicità.
4. **Gusti personali**: è ovvio che a parità di condizioni entrino in gioco anche le nostre preferenze e il fatto che un linguaggio di programmazione o ambiente di sviluppo ci sia più congeniale di altri.

Mentre i primi due punti sono prettamente "tecnici", misurabili, i rimanenti rimandano ad altre considerazioni che non possiamo certo fare qui, ma tanto per farne una: ***il fatto che esistano centinaia e centinaia di librerie di software già collaudate negli anni*** (minor tempo speso in codifica e *debugging*) ***ma che però ci vincolano sulla scelta del linguaggio*** (*per esempio potrebbero essere già scritte in C e potremmo non aver tempo di riscriverle per cui decidiamo di adottare il C per il nostro progetto*).
Vi sono poi anche **implicazioni etiche e/o commerciali** poiché i programmi possono essere di varia natura (open source, proprietari, commerciali, didattici, destinati alla distribuzione o ad uso interno di pochi membri di un'organizzazione): a seconda dei casi può far comodo un certo tipo di linguaggio piuttosto che un altro, ad esempio, è difficile che un programma commerciale sia distribuito con il codice sorgente e quindi difficilmente vengono usati per lo scopo linguaggi interpretati poiché il sorgente è necessario in fase d'esecuzione del programma; **normalmente i programmi commerciali sono tutti "compilati"** in modo che il codice sorgente non ci sia proprio, anzi, a volte anche il codice eseguibile è addirittura criptato in modo da risultare incomprensibile ai "curiosi".
Detto questo vi sono poi considerazioni tipiche per le varie applicazioni.
I **programmi per Windows** sono principalmente scritti in Visual BASIC, C/C++ (vari compilatori come Visual C++, Borland C++ Builder, gcc, etc.), Delphi/Pascal e Java.
Per i **programmi Unix o Linux** la scelta cade spesso sul C/C++ anche per motivi storici e culturali (Unix e Linux sono scritti in larga misura in C, i Windows Manager principalmente in C o C++ come le principali librerie) e quindi è una scelta consolidata, praticamente "naturale".
I **programmi MultiPiattaforma**: ad oggi, in pratica, sono scritti in Java oppure in C/C++ o C#, ricorrendo a Framework appositamente progettati per la portabilità es. wx-Widget (OpenSource), Borland CLX, Microsoft.NET grazie al progetto Mono. Da questo punto di vista, anche il C# merita d'essere preso in considerazione come validissima alternativa.
Programmi che girano "su Internet": sono scritti normalmente in PHP, ASP.NET,Java (attraverso Java Server Page).



Nelle **applicazioni numeriche/calcolo/simulazione, ricerca/ottimizzazione** il linguaggio più utilizzato è, per ragioni storiche, il **Fortran**. Per questo linguaggio compilato, si è scritto ogni genere di funzione e quindi, sebbene Fortran si di "vecchia generazione", è ancora una scelta "sensata".
Il **C/C++** ha ormai raggiunto e superato il Fortran e quindi è sempre una scelta più che valida, a maggior ragione in questo ambito, dato che risulta al contempo sia molto efficace che molto veloce ed efficiente: non fa certo rimpiangere il Fortran nel "macinare numeri".
Oggi però esistono molti applicativi (***MathCAD***, ***Matlab***, ***Octave*** per dirne alcuni) che incorporano linguaggi di programmazione che possono presentarsi, in certi ambiti, come valide alternative.
Infatti i linguaggi che mettono a disposizione, integrandosi con le numerose funzioni del programma stesso, garantiscono tempi di sviluppo e codifica molto più rapidi.
Altro punto a favore è che tali linguaggi normalmente si pongono a un livello ancor più alto dei linguaggi di cosiddetto "alto livello", ad esempio, Matlab descrive gli algoritmi con un linguaggio molto vicino a quello della "matematica pura" e gestisce un gran numero di entità matematiche (tipi di dati come ad esempio le matrici, operatori e funzioni) che altrimenti richiederebbero ore ed ore di programmazione, e quindi può essere "programmato" anche da chi non ha familiarità con altri linguaggi di programmazione.
Pensiamo a un ingegnere che deve fare una verifica numerica ma non ha tempo e voglia di scriversi un programma, compilarlo, cercare gli inevitabili errori... con Matlab risolve i problemi di calcolo in poco tempo.
Occorre rimanere mentalmente "aperti", capire pro e contro di una soluzione rispetto a un'altra.
Personalmente ho scritto algoritmi di calcolo in: Assembly (accadeva tanti anni fa per fortuna), Basic, Pascal, C e C++. Ad oggi **VisualBASIC.NET** e **C++** sono le scelte che prediligo per sviluppare applicazioni "da zero". Per Windows uso entrambi. Per Linux soltanto il secondo (anche se esiste Gambas che è simile a VB per Linux).
Anche Matlab oppure Octave rappresentano in molti casi essere ottime scelte. Grazie a Visual BASIC for Application anche lo stesso Excel può offrire soluzioni valide.
In pratica in questo articolo metterò a confronto:

- **Octave (o Matlab)**
- **C++ e managed C++**
- **Visual BASIC.NET**
- **Excel (con Visual Basic for Application)**

Cercherò di spiegare quelli che considero i punti di forza e i punti deboli di ciascuna soluzione, in base alla mia esperienza.



C++

Il C++ è il re dei linguaggi compilati. Molto potente, si programma al livello che si desidera in quanto racchiude in sé tutta la potenza di un linguaggio OOP moderno di alto livello ma al tempo stesso ha l'efficienza del C, a patto di conoscerlo bene, infatti:

1. il C++ può essere programmato come fosse C qualora si dovessero presentare problemi "prestazionali", non a caso i compilatori C++ compilano generalmente anche codice "C" puro.
2. un programma scritto "correttamente" in C++ non è più lento di un programma in C se non a livello infinitesimo (può capitare se il programma è scritto "male");
3. spesso quello che rallenta un programma in C++ è l'uso indiscriminato di determinati framework (strumenti utilissimi che fanno tanto lavoro per noi ma è bene farne un uso "consapevole" dato che ad essi deleghiamo funzioni importanti del nostro software).

Il C/C++ proprio come il Fortran ha accumulato negli anni, migliaia e migliaia di righe di codice in centinaia di librerie utilizzate nel calcolo numerico e non (cito ad esempio ***Numerical Recipes in C*** o ***Numerical Recipes in C++***) per semplificare la stesura degli algoritmi.
A differenza di tutte le altre soluzioni presentate, il C++ apre le porte alla scrittura di qualunque tipo di applicazione, per la sua potenza e versatilità.
Il C++ è facilmente accessibile nel senso che per programmare è sufficiente dotarsi di un compilatore: il linguaggio è uno standard ISO, non è quindi un linguaggio proprietario.
Molti ottimi compilatori C/C++ sono Open Source (o comunque distribuiti gratuitamente).
Il C++ è una scelta che non delude mai quando è richiesta potenza di calcolo, velocità. Per contro i tempi di sviluppo possono essere "importanti" e vanno ben considerati.
Diverso è il discorso del managed C++ (.NET è l'esempio più notevole).
A mio avviso l'uso del managed C++ è giustificato soltanto dal vantaggio che si può trarre dal ricorso a tecnologie presenti nel framework .NET altrimenti, tanto vale utilizzare lo standard C++.

Visual BASIC.NET

E' il linguaggio BASIC della piattaforma .NET di Microsoft per programmare applicazioni Windows 32 e 64 bit con uno strumento RAD. **Visual BASIC.NET è oggi un vero e proprio linguaggio OOP**.
Ora è possibile programmare e utilizzare vere classi (è diventato realmente Object Oriented), stilisticamente molto bello, pulito, veramente leggibile e facile da manutenere.
Per contro a differenza del vecchio BASIC (che ho sempre ripudiato), ora è un linguaggio veramente difficile da imparare e comprendere "a pieno", non meno del C++.
La piattaforma .NET si propone come anello di congiunzione tra vecchie tecnologie come COM e ActiveX, sostituiti da classi di oggetti "gestiti" (come nel managed C++ o nel C#).
La gestione "mediata" da una macchina virtuale assicura un garbage collector ed aumenta la sicurezza perché evita che il programma possa dialogare "direttamente" con il S.O. per l'allocazione della memoria.
Con VB.NET oggi si scrivono tantissime applicazioni per Windows, nei gestionali addirittura la fa da padrone per il facile accesso a tecnologie come ADO.NET, LINQ, classi per XML; per non parlare di Microsoft Presentation Foundation che consente di utilizzare tecnologie avanzate come DirectX.
Proprio come per il C++, stiamo parlando di un vero e proprio linguaggio di programmazione.
Questo assicura uno strumento che si pone al top della scelta ma che conserva tutti i problemi connessi allo sviluppo del software che ben conoscono i programmatori di computer.



Excel

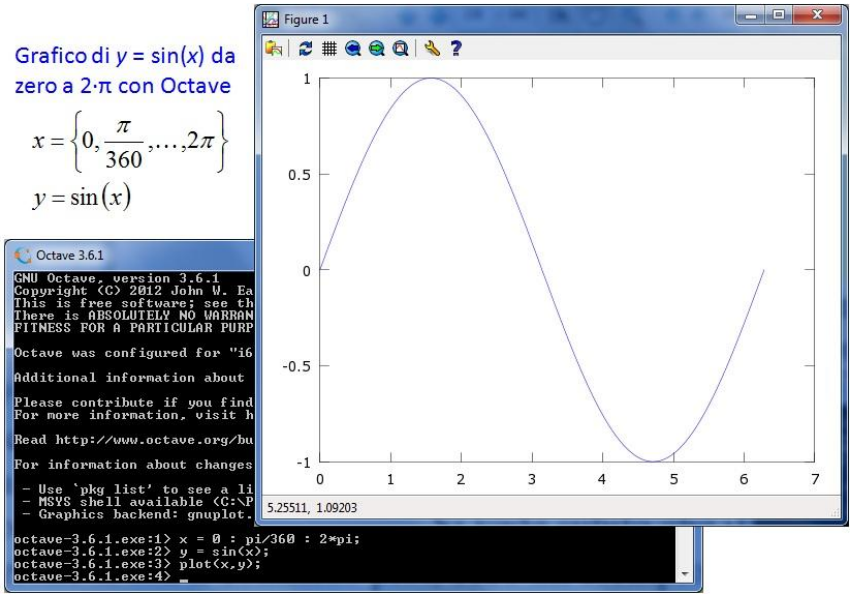
Quando dobbiamo fare un conto al computer ci viene subito in mente di utilizzare un foglio elettronico come Microsoft Excel, tanto per menzionare il più diffuso.
Il foglio elettronico è costituito da "un mare" di celle sulle quali è possibile compiere le più svariate operazioni (aritmetiche, logiche, di ricerca, informative e altro ancora). **Dispone di buone capacità grafiche e diverse caratteristiche di "reportistica" evoluta**: la ricerca obiettivo che consente di modificare l'input per ottenere un output specifico (di fatto risolve numericamente equazioni algebriche).
In pochi minuti si predispongono fogli di calcolo capaci di risolvere diversi problemi, per i quali poi basterà richiamare il file, modificare i dati del problema e vedere subito la risposta.
E' sicuramente adatto per tantissime situazioni ma difetta per quanto riguarda il calcolo e l'analisi numerica, di cui molti calcoli per applicazioni scientifiche necessitano: ad esempio non ha funzioni per derivare funzioni, calcolare i limiti, integrare funzioni o equazioni differenziali, non conosce i numeri complessi.
Excel tuttavia, entra di diritto nella schiera delle applicazioni "programmabili" infatti, **ricorrendo a *Visual BASIC for Application*, è possibile integrare il programma aggiungendo in pratica qualunque algoritmo**.
VBA è un subset di Visual Basic 6.0 (e non VisualBASIC.NET) ma che all'interno di Microsoft Excel ne moltiplica le potenzialità di un fattore elevato.
Diventa un'ottima scelta qualora il nostro problema può essere risolto in larga misura dai tanti strumenti che il Excel già mette a disposizione, per programmare soltanto quello che manca.
VBA viene compilato al momento dell'esecuzione in P-Code quindi il codice è eseguito con una buona velocità rispetto a un Basic "interpretato".
Va detto che **mentre C++ e Visual BASIC.NET (o altri linguaggi di programmazione "compilati") consentono di scrivere programmi eseguibili e indipendenti dal sistema che li ha generati, questa soluzione è invece una estensione di Microsoft Excel e rimane legata alla presenza del foglio elettronico**.
Per questo motivo va bene a scrivere procedure utili a livello personale o a livello di una organizzazione, ma se intendiamo scrivere un programma da distribuire, la dipendenza da Excel può essere un vincolo insormontabile.

Octave

Octave è un'applicazione specifica per l'**analisi** e il **calcolo numerico** (anche **numeri complessi**), l'**algebra lineare**, **calcolo matriciale**, **grafici di funzioni**, **equazioni algebriche lineari e non lineari**, **equazioni differenziali ordinarie lineari e non lineari**, **integrazione numerica**, **problemi di ottimizzazione**, **statistica**, **insiemi**, **geometria**, **interpolazione**, **teoria dei segnali** in larga misura compatibile con Matlab.
A differenza di Matlab non sovolge calcolo simbolico e non ha l'ambiente grafico Simulink.
Il linguaggio di programmazione (interpretato) è compatibile al 90% con Matlab: le poche differenze si possono adattare, per scrivere programmi che possono girare per entrambi Octave e Matlab.
Octave è rilasciato sotto licenza GPL e quindi può essere liberamente copiato e usato (**non costa un centesimo**). Il programma gira su Unix, Linux, Windows e MAC OS X. E' quindi una **soluzione *multiplatform***.
Mentre con un linguaggio di programmazione si deve costruire oltre all'algoritmo principale tutte le funzioni che servono, ad esempio se bisogna integrare una equazione differenziale, occorre una procedura che lo sappia fare, se occorre graficare funzioni occorre creare una interfaccia grafica, eccetera, Octave e Matlab oltre a conoscere praticamente tutti i più noti algoritmi di calcolo, includono avanzatissime capacità grafiche e non ci si deve curare dell'interfaccia, inoltre, **il linguaggio è di altissimo livello e consente di descrivere le operazioni da fare in modo molto vicino a come si farebbe in matematica**. A lato vediamo quanto è facile graficare una funzione.

Grafico di $y = \sin(x)$ da zero a $2\cdot\pi$ con Octave

$$x = \left\{0, \frac{\pi}{360}, \ldots, 2\pi\right\}$$
$$y = \sin(x)$$



Nessun linguaggio di programmazione "classico" consente di fare questo così rapidamente.
Questo è l'esempio più stupido che mi è venuto in mente... due istruzioni in più risolvono equazioni differenziali: provateci con il foglio elettronico o con qualsiasi linguaggio di programmazione!
MATLAB e Octave consentono di sviluppare gli algoritmi più velocemente rispetto ai tradizionali linguaggi di programmazione.
Forniscono già pronti, tutti gli strumenti necessari a trasformare le idee in algoritmi, tra cui:

1. Migliaia di funzioni matematiche, ingegneristiche e scientifiche
2. Algoritmi specifici di applicazione nei domini come elaborazione di segnali e immagini, progettazione di controlli, finanza computazionale e biologia computazionale
3. Sviluppo di strumenti per la modifica, il debug e l'ottimizzazione degli algoritmi.

Non vanno considerati come linguaggi di programmazione classici: l'interprete di Octave è molto lento. Lo stesso Matlab che invece dispone di un compilatore in P-Code rimane comunque meno performante di un programma in C/C++. Il loro obiettivo non è quello di conferire velocità al codice ma di agevolarne la creazione e lo sviluppo.

Mettiamoli alla prova

Consideriamo due problemi molto semplici.
Il primo è un algoritmo che coinvolge l'elaborazione di **numeri interi** e si pone come obiettivo la ricerca dei numeri primi (minori di un certo numero), il secondo invece è la soluzione di una equazione differenziale, che ovviamente fa largo uso dei **numeri in virgola mobile**. Gli algoritmi d'esempio saranno velocemente discussi nei prossimi paragrafi.

Crivello di Eratostene

Un numero intero maggiore di 1 è primo, se ammette soltanto due divisori: 1 e se stesso.
Un metodo per trovare i numeri primi fino a un fissato limite superiore è il cosiddetto **crivello** (*o setaccio*) **di Eratostene**[5], dal nome del grande matematico che lo ha inventato, prima del 200 a.C., di seguito illustrato.

Si comincia con l'elencare tutti i numeri naturali da 2 al limite superiore (es. N < 50).

		2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49

1) Consideriamo il primo della lista (**i = 2**). Cancelliamo tutti i suoi multipli (*per definizione non primi*) fino a N.

		2	3	~~4~~	5	~~6~~	7	~~8~~	9
~~10~~	11	~~12~~	13	~~14~~	15	~~16~~	17	~~18~~	19
~~20~~	21	~~22~~	23	~~24~~	25	~~26~~	27	~~28~~	29
~~30~~	31	~~32~~	33	~~34~~	35	~~36~~	37	~~38~~	39
~~40~~	41	~~42~~	43	~~44~~	45	~~46~~	47	~~48~~	49

2) Consideriamo il prossimo numero tra i non cancellati (**i = 3**). Cancelliamo tutti i suoi multipli fino a N.

		2	**3**	~~4~~	5	~~6~~	7	~~8~~	~~9~~
~~10~~	11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19
~~20~~	~~21~~	~~22~~	23	~~24~~	25	~~26~~	~~27~~	~~28~~	29
~~30~~	31	~~32~~	~~33~~	~~34~~	35	~~36~~	37	~~38~~	~~39~~
~~40~~	41	~~42~~	43	~~44~~	~~45~~	~~46~~	47	~~48~~	49

3) Se esistono altri numeri non cancellati, ripeto il punto 2 (*i = 5, poi 7, poi 11...*). Alla fine ottengo:

		2	**3**	~~4~~	**5**	~~6~~	**7**	~~8~~	~~9~~
~~10~~	**11**	~~12~~	**13**	~~14~~	~~15~~	~~16~~	**17**	~~18~~	**19**
~~20~~	~~21~~	~~22~~	**23**	~~24~~	~~25~~	~~26~~	~~27~~	~~28~~	**29**
~~30~~	**31**	~~32~~	~~33~~	~~34~~	~~35~~	~~36~~	**37**	~~38~~	~~39~~
~~40~~	**41**	~~42~~	**43**	~~44~~	~~45~~	~~46~~	**47**	~~48~~	~~49~~

Quindi **i numeri rimasti non cancellati sono primi**, nell'esempio, ci sono 15 numeri primi minori di 50:
M = { ***2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47*** };

Il crivello è ancora utilizzato come algoritmo di calcolo dei numeri primi da molti programmi per computer. Pur non essendo un algoritmo straordinariamente efficiente, è in compenso, semplice da tradurre in un qualsiasi linguaggio di programmazione e **rappresenta un buon banco prova per testare la velocità di codice eseguibile per computer e CPU stesse nell'eseguire operazioni sui numeri interi** (*con N particolarmente elevati si calcola il tempo necessario per trovare i numeri primi*).
Calcoleremo i numeri primi fino a N = 100000000 (cento milioni) e vedremo quanto tempo s'impiega.

[5] Eratostente (Cirene, circa 275 a.C., Alessandria d'Egitto 195 circa a.C.). Intellettuale dai molteplici interessi è stato un matematico, astronomo, geografo , poeta, storico e filosofo greco.
Fu uno dei principali studiosi del suo tempo e con essi mantenne sempre i contatti. Diresse ad Alessandria la più grande biblioteca del mondo antico e fu precettore di Tolomeo IV Filopatore. Oltre alla scoperta del metodo per trovare i numeri primi, gli sono stati attribuiti diversi risultati, tra cui una delle più accurate misurazioni della circonferenza terrestre di tutta l'antichità (rispetto alle misure con gli strumenti attuali l'errore fu di appena il 1.5%, una misura straordinariamente accurata per i mezzi dell'epoca a testimonianza del suo grande genio), fondata sull'osservazione delle differenti ombre proiettate dal sole in luoghi posti a diverse latitudini! (Carl B. Boyer, A history of Mathematics, John Wiley & Sons, 1968).

Octave

Vediamo l'implementazione con Octave del Crivello di Eratostene:

```
% Sieve.m
% Written by Paolo Ferraresi (C) 2012 with GNU Octave
1;
clear all
N = 100000000; % cento milioni
t0 = clock(); % legge il tempo
v = int8(ones(1,N)); % crea un vettore v di N numeri tutti al valore 1
L = fix(sqrt(N)+1);
for i=uint32(2:L)
  if v(i)  % se v(i) diverso da zero, è un numero primo
    for j=uint32(i*i:i:N) % elimina tutti i multipli di i con molteplicità maggiori o uguali a i fino a N
      v(j)=0; % mettendo il numero nella lista a zero
    end
  end
end
% conta e salva in M i numeri primi trovati (quelli diversi da zero, quelli rimasti nel vettore)
j = 0;
for i=2:N
  if v(i)
    j = j + 1;
    M(j) = i;
  end
end
printf("\nSecondi impiegati: %d\n",etime(clock(),t0));
printf("\nTrovati %d numeri primi.\n",columns(M));
```

L'istruzione **v = int8(ones(1,N))** crea una matrice di 1 riga e N colonne (i vettori sono particolari matrici con una dimensione unitaria): ciascun elemento è settato a 1 (per noi significherà che il numero è primo o presunto tale fino alla sua cancellazione, 0 è il numero cancellato dalla lista, e quindi non primo).

Il ciclo itera *i* da 2 a $\lfloor\sqrt{N}+1\rfloor$ (parte intera, ottenuta con la funzione **fix**).

Non c'è bisogno di iterare fino a N perché, con riferimento all'esempio della pagina precedente (N < 50):

1. Prendiamo il primo (i = 2), rimuove i numeri 4,6,...48.
2. Prendiamo il secondo (i = 3), rimuove i numeri 9,12,...48.

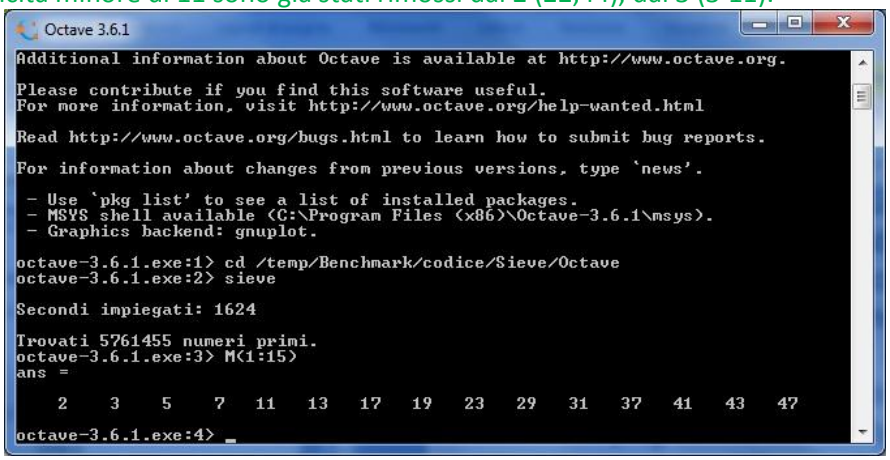
Notiamo che per i = 3 non è necessario partire a cancellare da 2i perché il 6 è già stato rimosso in quanto multiplo di 2, quindi a cancellare si parte da i^2 fino a N a salti di i, per questo motivo ci si può fermare alla radice di N+1, infatti:

3. Prendiamo i = 7, eliminando da i^2, rimuove il numero 49.
4. Prendiamo i = 11. Non elimina nessun numero dalla lista perché $11^2 = 121$.

Infatti, i multipli di 11 con molteplicità minore di 11 sono già stati rimossi dal 2 (22,44), dal 3 (3·11).

Questo è valido per qualunque numero primo tale per cui i^2>N.

Infine i numeri primi vengono contati (in *j*) e salvati nel vettore **M**, quindi il programma scrive il tempo impiegato e il numero di numeri primi trovati.



Il programma ha impiegato quasi 30 minuti per trovare 5761455 numeri primi.

Per vederne alcuni (o anche tutti) bisogna chiederlo, una volta che il programma ha terminato, inviando un comando ulteriore a Octave, ad esempio:

```
M(1:15)
```

mostra sullo schermo i primi quindi numeri.

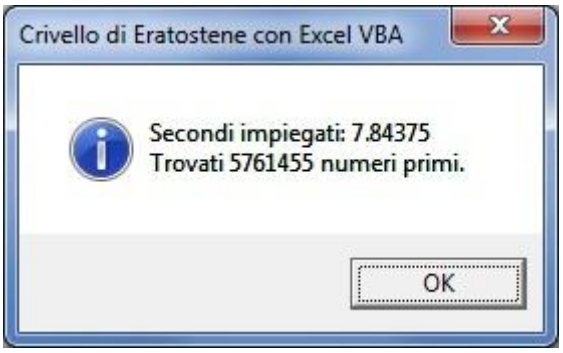
Nota: il programma non prevedeva questo comando, ma per il fatto che è un linguaggio interpretato gli oggetti definiti nel Workspace rimangono allocati e qualunque comando non fa differenza che sia impartito durante l'esecuzione del programma (Run-Time) o direttamente dalla linea di comando.

Excel (con Visual BASIC for Application)

Ecco il codice per Excel e VBA.

```
Option Explicit
Public Const NP As Long = 100000000 ' cento milioni
Function Sieve(ByVal N As Long) As Long
    Dim v() As Boolean
    ReDim v(N)
    Dim M As New Collection
    Dim L As Long, i As Long, j As Long
    ' Crivello di Eratostene
    L = Int(Sqr(N) + 1)
    For i = 2 To L
        If v(i) = False Then
            For j = i * i To N Step i
                v(j) = True
            Next
        End If
    Next
    ' salva i numeri primi in M e ne restituisce il numero
    For i = 2 To N
        If v(i) = False Then
            M.Add (i)
        End If
    Next
    Sieve = M.Count
    Set M = Nothing
End Function
Sub Test()
    Dim j As Long, S As String, t0 As Single
    t0 = Timer
    j = Sieve(NP)
    S = "Secondi impiegati: " & CSng(Timer - t0) & vbNewLine & _
        "Trovati " & CLng(j) & " numeri primi."
    MsgBox S, vbInformation, "Crivello di Eratostene con Excel VBA"
End Sub
```

Il codice è abbastanza simile al precedente.
In questo caso si utilizza un vettore **v** per la lista con tutti i numeri. Poiché dopo la dichiarazione il vettore v è pieno di valori settati a False, conviene impostare a True il numero quando non è primo, piuttosto che inserire un ciclo che mette tutti a True e li riporta a False quando il numero non è primo.



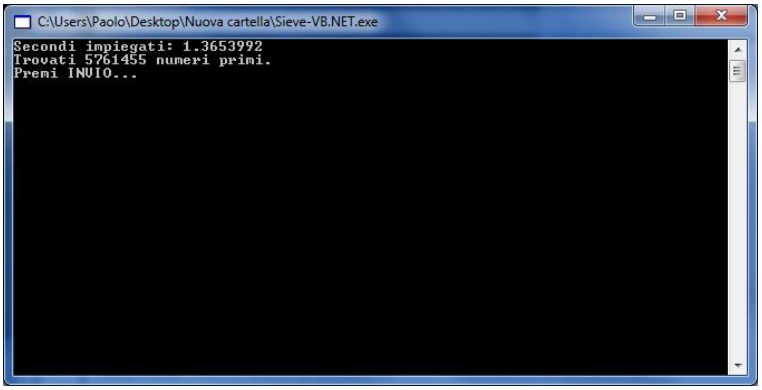
Lanciando la procedura Test, **nel giro di pochi secondi il programma calcola i numeri primi**.
Nota: il BASIC di Excel è sia interpretato che compilato quindi prevede una finestra dove dare dei comandi ma non è flessibile come quella di Octave per cui ora, saremmo in grossa difficoltà se volessimo vedere i numeri primi trovati.

Visual BASIC.NET

Ed ora ecco la versione in Visual BASIC.NET.

```
' Sieve.vb
' Written by Paolo Ferraresi (C) 2012 with Visual BAISC.NET 2010 console application
Module Module1
    Public Const NP As UInteger = 100000000 ' cento milioni
    Function Sieve(ByVal N As UInteger) As UInteger
        Dim v() As Boolean              ' alloca un vettore v di Boolean
        ReDim v(N)                      ' lo dimensiona a N valori impostati a false
        Dim M As New List(Of UInteger)  ' Vettore dinamico (List è più veloce di ArrayList)
        ' Crivello di Eratostene
        Dim L As UInteger = Int(Math.Sqrt(N) + 1)
        For i As UInteger = 2 To L
            If v(i) = False Then
                For j As UInteger = i * i To N Step i
                    v(j) = True
                Next
            End If
        Next
        ' Conta i numeri primi
        For i As UInteger = 2 To N
            If v(i) = False Then
                M.Add(i)
            End If
        Next
        Return M.Count
    End Function
    Sub Main()
        Dim T As New Stopwatch
        T.Start()
        Dim L As UInteger = Sieve(NP)
        Console.WriteLine("Secondi impiegati: {0}", T.Elapsed.TotalMilliseconds * 0.001)
        Console.WriteLine("Trovati {0} numeri primi.", L)
        Console.Write("Premi INVIO...")
        Console.ReadKey()
    End Sub
End Module
```

Il codice è ovviamente simile ai precedenti, in particolare al VBA usato dentro Excel.
Anche in questo caso per il vettore **v** si usa un array "classico" definito con **Dim** e (ri)dimensionato con **ReDim**, questo comando è lo stesso di VisualBASIC 6 (e quindi del VBA di Excel).
Per **M** (i numeri primi trovati), invece, si possono usare i "generics", in particolare **List(Of T)** è quello che fa al caso mio: consente di allocare in modo del tutto automatico una lista di elementi del tipo T (in questo caso Interi senza segno).



Proprio come per il codice scritto in VBA, l'array esce dal ReDim già inizializzato a False e quindi sarebbe tempo perso riempirlo di valori True, meglio aggirare l'ostacolo ritenendo False il presunto numero primo e settarlo a True per indicare che è cancellato.
Il tempo viene preso con la classe **Stopwatch**.
Il programma impiega 1.36 s per essere eseguito.
In questo caso, la lista M è proprio creata per nulla: nel senso che al termine dell'esecuzione, il programma la distrugge e non è possibile impartire comandi extra. Il programma compilato non ci accorda infatti la possibilità d'impartire altri comandi come fanno invece i programmi scritti con linguaggi interpretati e quindi se siamo interessati a fare come abbiamo fatto con Octave, cioè volessimo visualizzare alcuni dei numeri primi trovati (o anche tutti), dovremmo ***modificare il programma, ricompilarlo e rimandarlo in esecuzione, e fare questo ogni volta che si modifica qualunque cosa***. Detto questo, attualmente è il programma che ha impiegato meno dei precedenti per risolvere il problema.

C++

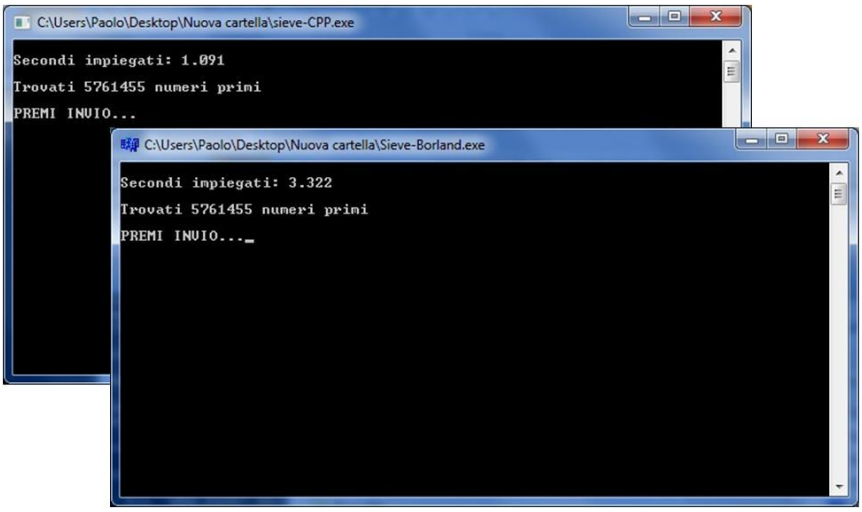
Il seguente codice **standard C++** è stato compilato con Visual C++ 2010 ma è praticamente portabile su qualunque sistema rimuovendo la prima direttiva (**#include "stdafx.h"**) tipica del compilatore Microsoft e utile per gli *header precompilati* (accelera la compilazione), simile al **#pragma hdrstop** per il Borland.

```
// Sieve.cpp
// Written by Paolo Ferraresi (C) 2012 Standard C++ console application
#include "stdafx.h"
#include <iostream>
#include <vector>
#include <ctime>
const unsigned int NP = 100000000; // cento milioni

unsigned int Sieve(const unsigned int N)
{
	std::vector<bool> v(N+1,true); // crea la lista dei numeri (con un vettore di booleani)
	std::vector<unsigned int> M; // e' il vettore dove verranno memorizzati i num. primi
	// Crivello di Eratostene
	for (unsigned int i=2;i*i<=N;++i)
		if (v[i]) for (unsigned int j = i*i;j<=N;j+=i)
			v[j]=false;
	// trova i numeri primi (quelli rimasti in v) e li aggiunge in fondo (in fila) a M
	for (unsigned int i=2;i<=N;++i)
		if (v[i])
			M.push_back(i);
	return M.size(); // ritorna il numero di elementi dentro M (conta i num. primi)
}

int main(int argc, char* argv[])
{
	std::clock_t t0;
	t0 = std::clock(); // legge il tempo
	unsigned int j = Sieve(NP); // lancia la funzione che calcola i num primi.
	std::cout	<< "\nSecondi impiegati: " << static_cast<float>((std::clock()-t0))/CLK_TCK << "\n"
			<< "\nTrovati " << j << " numeri primi\n" << "\nPREMI INVIO...";
	// attende la pressione del tasto INVIO
	std::cin.clear(); // svuota il buffer per evitare che venga preso un tasto premuto prima
	std::cin.get(); // legge un carattere da console in.
	return 0;
}
```

Nel caso del C++ Standard ho scelto di utilizzare la **STL** per poter utilizzare **std::vector**. La ***Standard Template Library*** è comoda, veloce, stabile e affidabile sia in fase di sviluppo che in fase d'esecuzione. Un uso ben fatto delle librerie STL non penalizza la velocità d'esecuzione.



L'algoritmo è quindi lo stesso di prima e la differenza è giusto il fatto che la classe container **vector** utilizza i booleani (true, false) con la possibilità di settarli già al valore che preferiamo (in questo caso **true**) attraverso il costruttore.
Durante la conta dei numeri primi non devo aggiornare manualmente un indice ma provvede a tutto il metodo **push_back()** del vettore che, aggiunge un elemento in coda.
La sua numerosità è restituita dal metodo **size()**. In questo non ci sono differenza con il List(Of T) di Visual BASIC.NET.
Al termine dell'esecuzione, la memoria è rilasciata, senza possibilità di effettuare altre operazioni.
Il C++ impiega (*sul mio Intel Core i7*) **1.1 secondi con Visual C++ 2010 e 3.3 secondi con il Borland C++ Builder 6**.

Managed C++

Adattiamo il programma precedente per funzionare con le classi dalla piattaforma .NET.
La STL è sostituita dai Generics: in pratica List<T> è la stessa usata nel programma in Visual BASIC.NET.

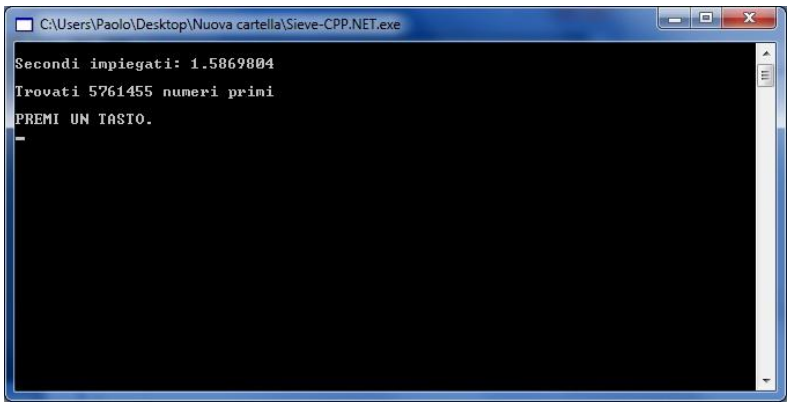
```
// Sieve.cpp
// Written by Paolo Ferraresi (C) 2012 with Visual C++ CLR console application
#include "stdafx.h"
using namespace System;

const unsigned int NP = 100000000; // cento milioni

unsigned int Sieve(const unsigned int N)
{
	array<bool> ^v = gcnew array<bool>(N+1); // array in managed C++.NET std::vector funziona ancora!
	for(unsigned int i=0;i<=N;++i) v[i]=true; // purtroppo il costruttore non setta a true!
	Collections::Generic::List<unsigned int>^ M = gcnew Collections::Generic::List<unsigned int>();

	// Crivello di Eratostene
	for (unsigned int i=2;i*i<=N;++i) // i va da 2 fino a quando i*i<=N
		if (v[i]) for (unsigned int j = i*i;j<=N;j+=i) // se trova un numero primo rimuove da
			v[j]=false;                                // v tutti i multipli di molteplicità >=i .
	// trova i numeri primi (quelli rimasti in v) e...
	for (unsigned int i=2;i<=N;++i)
		if (v[i])
			M->Add(i); // li aggiunge in fondo al vettore M.
	return M->Count; // ritorna il numero di elementi dentro M (conta quindi i numeri primi)
}

int main(array<System::String ^> ^args)
{
	Diagnostics::Stopwatch^ T = gcnew Diagnostics::Stopwatch;
	T->Start();
	unsigned int j = Sieve(NP); // lancia la funzione che calcola i numeri primi.
	Console::WriteLine("\nSecondi impiegati: " + (T->Elapsed.TotalMilliseconds*0.001).ToString());
	String ^S = gcnew String(static_cast<unsigned int>(j).ToString());
	Console::WriteLine("\nTrovati " + S + " numeri primi\n\nPREMI UN TASTO.");
	Console::ReadKey(); // legge un carattere da Console.
	return 0;
}
```



Vediamo che impiega maggior tempo nell'esecuzione rispetto al programma di prima, compilato con Visual C++ in codice nativo.
Questo è il tempo in più necessario al compilatore JIT.
Fortunatamente questo è un tempo "fisso" che si perde soltanto all'avvio del programma, poi le prestazioni dovrebbero essere "identiche", assicura Microsoft.
Personalmente non sono un estimatore del managed C++.
Il C++ è un linguaggio "perfetto" che non aveva la necessità di sostituire le classi della libreria standard.
Le mie rimostranze riguardano il fatto che la libreria STL è molto più leggera e flessibile delle classi Generics di .NET, ad esempio.
Gli iteratori STL hanno un livello di astrazione maggiore rispetto all'interfaccia IEnumerator di .NET.
Ho molto apprezzato le classi di contenitori generici .NET, programmando con Visual BASIC, perché vanno a colmare un vuoto che indubbiamente c'era, ma il C++ già disponeva della STL.
Però, questi sono gusti personali.
Vero anche che .NET può diventare molto comodo quando si scrivono applicazioni per Windows con tanto di controlli, DirectX e altre tecnologie messe a disposizione dal sistema operativo. Ma se non è questo il motivo, meglio non lasciare il vecchio e caro standard C++ compilato.

Integrazione di ODE

Come esempio di calcolo in virgola mobile supponiamo di avere il seguente problema di Cauchy:

$$\frac{d^2y}{dx^2} = -\frac{1}{2} \text{ ai seguenti valori iniziali } \begin{cases} y_0 = 0 \\ \left.\frac{dy}{dx}\right|_{x=0} = \frac{5}{2} \end{cases}$$

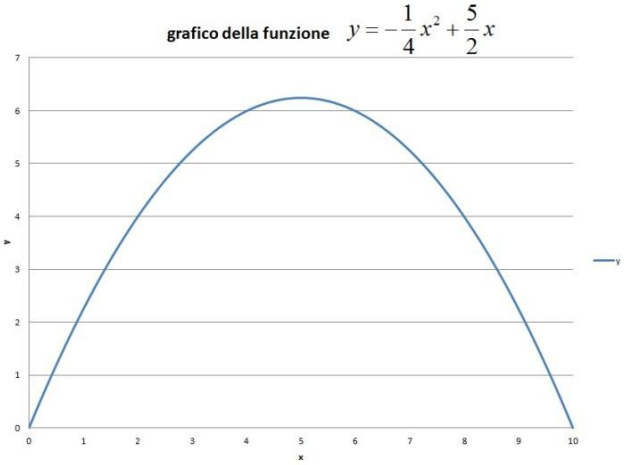
Vorremmo calcolare il valore della funzione *y*(x) tra 0 ≤ x ≤ 10.
Per risolvere l'equazione chiamiamo:

$z = \frac{dy}{dx}$, e ovviamente si ha: $\frac{dz}{dx} = \frac{d^2y}{dx^2}$. Integrando con le condizioni iniziali, due volte, otteniamo:

$$\frac{dz}{dx} = -\frac{1}{2} \rightarrow \int_{5/2}^{z} dz = -\frac{1}{2}\int_0^x dx \rightarrow z = -\frac{1}{2}x + \frac{5}{2}$$

e

$$\frac{dy}{dx} = -\frac{1}{2}x + \frac{5}{2} \rightarrow \int_0^y dy = -\frac{1}{2}\int_0^x xdx + \frac{5}{2}x \rightarrow y = -\frac{1}{4}x^2 + \frac{5}{2}x$$



A destra, possiamo vedere anche un grafico della funzione *y*(*x*).
Come ulteriore verifica, ai programmi, faremo calcolare l'integrale definito della funzione appena calcolata da 0 a 10, che risulta:

$$Q = -\frac{1}{4}\int_0^{10} x^2 dx + \frac{5}{2}\int_0^{10} xdx = -\frac{1}{12}10^3 + \frac{5}{4}10^2 = \frac{125}{3}$$

Chiederemo quindi ai programmi di calcolare l'integrale e verificare che risulti circa 41.667.

Per l'equazione d'esempio non c'è bisogno di un programma al computer dato che è facilmente risolvibile per via analitica, tuttavia ci consente di fare test affidabili poiché i risultati sono noti.
Una volta implementato un algoritmo, il computer non farà distinzione tra equazioni lineari e non, difficili o semplici da risolvere, semplicemente i parametri dell'equazione diventeranno uno dei possibili input e la soluzione sarà l'output del nostro algoritmo.
E' per questo che in informatica si cerca di studiare algoritmi semplici: in essi è racchiusa l'essenza per risolvere problemi più complessi.
Algoritmi semplici combinati insieme risolvono problemi più complicati.

Il metodo di Eulero

L'analisi numerica dispone di numerosi metodi che permettono di risolvere questo problema. Il più semplice è il **metodo di Eulero** che, può essere applicato alle **equazioni differenziali ordinarie (ODE) del primo ordine scritte in forma normale** e, per estensione, al grado *n* applicando *n* volte il metodo: fornisce risultati approssimati, così come ogni metodo numerico, di semplice comprensione, facilmente programmabile, ha però una convergenza lenta ed è spesso necessario fare molti calcoli per ottenere un accettabile approssimazione del valore cercato. Vediamo come funziona:

- la EDO, $\frac{dy}{dx} = f(x,y)$, genera un campo di direzioni nel dominio **D** in cui *f*(*x*,*y*) è definita;
- le linee integrali sono tangenti al campo di direzioni e ricoprono tutto il dominio **D**;
- essendo *f*(*x*,*y*) una derivata prima, è il coefficiente angolare della retta tangente in P(x,y) alla curva integrale passante per P(x,y);
- la retta di coefficiente angolare *m*, passante per il punto $P(x_0,y_0)$ ha equazione $y = m\cdot(x-x_0)+y_0$.

Poiché $y(x_0) = y_0$ è un dato del problema di Cauchy, ricaviamo un'approssimazione dei valori successivi:

$$y_{n+1} \approx y_n + h\cdot f(x_n, y_n) \qquad [1]$$

in cui *h* è definito come il "passo" ($x_{n+1} = x_n + h$).

La soluzione "avanza" di un passo *h* alla volta da un noto *n* al nodo *n*+1 ma considera il valore della derivata soltanto all'inizio del nodo e tale valore, purtroppo, viene assunto "costante" fino al nodo *n*+1 causando errori di approssimazione importanti. ***Il metodo può essere migliorato, considerando il valore della derivata intermedio tra i punti n e n+1***.

L'implementazione del metodo in questo articolo si rifà proprio a questo concetto, calcolando la derivata prima con il metodo di Eulero e riapplicandolo in un punto intermedio per calcolare il valore della funzione y.

x	y''	y' (Eulero)	y' = f'(x)	~~y (Eulero)~~	y (Eulero *)	y = f(x)	y' (Eulero *)
0	-0.5	2.5	2.5	~~0~~	0	0	2.5
0.5	-0.5	2.25	2.25	~~1.25~~	1.1875	1.1875	2.375
1	-0.5	2	2	~~3.375~~	2.25	2.25	2.125
1.5	-0.5	1.75	1.75	~~3.375~~	3.1875	3.1875	1.875
2	-0.5	1.5	1.5	~~4.25~~	4	4	1.625
2.5	-0.5	1.25	1.25	~~5~~	4.6875	4.6875	1.375
3	-0.5	1	1	~~5.625~~	5.25	5.25	1.125
3.5	-0.5	0.75	0.75	~~6.125~~	5.6875	5.6875	0.875
4	-0.5	0.5	0.5	~~6.5~~	6	6	0.625
4.5	-0.5	0.25	0.25	~~6.75~~	6.1875	6.1875	0.375
5	-0.5	0	0	~~6.875~~	6.25	6.25	0.125
5.5	-0.5	-0.25	-0.25	~~6.875~~	6.1875	6.1875	-0.125
6	-0.5	-0.5	-0.5	~~6.75~~	6	6	-0.375
6.5	-0.5	-0.75	-0.75	~~6.5~~	5.6875	5.6875	-0.625
7	-0.5	-1	-1	~~6.125~~	5.25	5.25	-0.875
7.5	-0.5	-1.25	-1.25	~~5.625~~	4.6875	4.6875	-1.125
8	-0.5	-1.5	-1.5	~~5~~	4	4	-1.375
8.5	-0.5	-1.75	-1.75	~~4.25~~	3.1875	3.1875	-1.625
9	-0.5	-2	-2	~~3.375~~	2.25	2.25	-1.875
9.5	-0.5	-2.25	-2.25	~~2.375~~	1.1875	1.1875	-2.125
10	-0.5	-2.5	-2.5	~~1.25~~	0	0	-2.375

Con riferimento alla tabella a destra e al problema di Cauchy considerato,

$$\frac{d^2y}{dx^2} = -\frac{1}{2} \text{ ai seguenti valori iniziali } \begin{cases} y_0 = 0 \\ \left.\frac{dy}{dx}\right|_{x=0} = \frac{5}{2} \end{cases}$$

y' calcolato con eq. 1 dà: $y'(0.5) = y'(0) + h\cdot y''(0) = 2.5 - 0.25 = 2.25$

Riapplicando il metodo di Eulero, avremmo per y:

$y(0.5) = y(0) + h\cdot y'(0) = 0 + 0.5\cdot 2.5 = 1.25$ (***sbagliato!***)

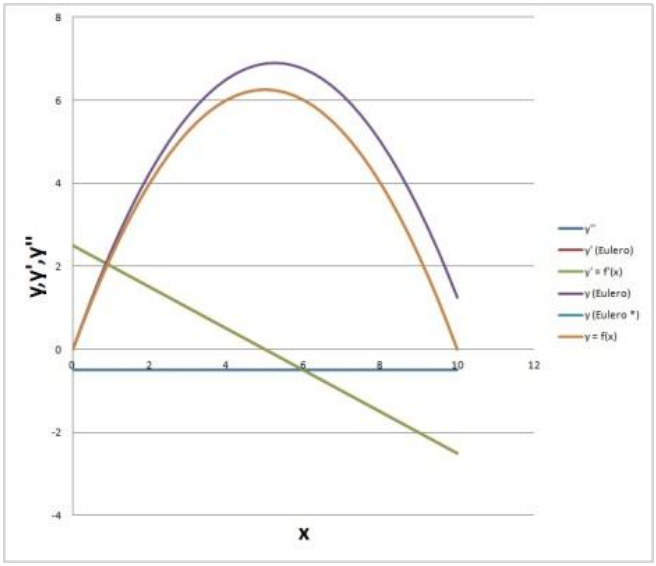
La derivata prima non rimane costante tra 0 e 0.5 e quindi otteniamo un errore d'approssimazione troppo grande!

La colonna *y'* (*Eulero* *) è il metodo migliorato, che prende i valori delle derivate nei punti intermedi, infatti dopo il valore corretto a x=0, tutti gli altri sono calcolati sul mezzo passo: y'(0.25)=2.375, y'(0.75)=2.125...

E' quindi possibile calcolare un valore più accurato per y:

$y(0.5) = y(0) + h\cdot y'(0.25) = 0 + 0.5\cdot 2.375 = 1.1875$

Reiterando i calcoli per tutti i punti si ha un calcolo numerico approssimato con l'integrazione dell'EDO di secondo grado.



Nel grafico si vedono le curve della derivata seconda, della derivata prima, della soluzione col metodo non corretto (in viola, vistosamente sbagliata): **la soluzione con il metodo corretto coincide perfettamente con la soluzione analitica**.

Al pari del il Crivello di Eratostene per trovare numeri primi e come test per l'aritmetica intera, il metodo di Eulero sopravvive nel tempo ed è ancora largamente utilizzato per la sua straordinaria semplicità.

Chiederemo ai programmi d'integrare questa equazione tra 0 e 10 con 10000000 (dieci milioni) di punti e di verificare il risultato integrando la funzione trovata e riscontrando che fornisca lo stesso risultato del calcolo formale (cioè l'integrale definito tra 0 e 10 vale circa 41.667).

Octave

Vediamo come se la cava Octave nell'implementazione del metodo di Eulero.

```
% risolve l'equazione differenziale y'' = -1/2 con y'(0)=5/2 e y(0)=y0=0.
% con il metodo di Eulero
1;
clear all
N = 10000000; % dieci milioni di punti
t0 = clock(); % legge il tempo

x = linspace(0,10,N+1); % var. indipendente
y2 = -0.5*ones(1,N+1); % derivata seconda = -0.5
y1 = zeros(1,N+1);
y = zeros(1,N+1);
Passo = x(2)-x(1);

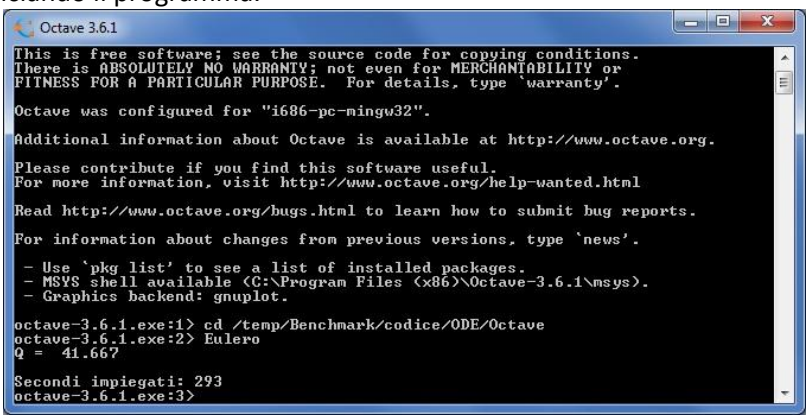
% imposta le C.I.
y1(1) = 2.5;
y(1) = 0.0;

% integra l'equazione
yp0 = y1(1)+0.5*Passo*y2(1) ; % anticipa di mezzo passo y'
for i = 2:N+1
  Passoy2 = Passo * y2(i);
  y(i) = y(i - 1) + Passo * yp0; % integra y
  y1(i) = y1(i - 1) + Passoy2; % integra y'
  yp0 += Passoy2;
end
Q = trapz(x,y) % calcola l'integrale definito di y(x) tra x(0) e x(N+1)
printf("\nSecondi impiegati: %d\n",etime(clock(),t0));
```

Il programma calcola *y*(*x*) in dieci milioni di punti tra 0 e 10 quindi integra la funzione sempre nel medesimo intervallo.

Notare che non ho dovuto scrivere nessuna funzione per calcolare l'integrale definito della funzione y(x) perché Octave ha già la funzione **trapz**(x,y) per fare questo.

Ecco cosa appare lanciando il programma.

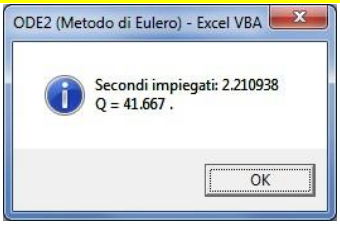


Il tempo necessario per compiere l'operazione è di 293 secondi, come mostra la finestra del programma, in pratica impiega 5 minuti.

Excel (con Visual BASIC for Application)

```
Option Explicit
Const NP = 10000000 ' dieci milioni di punti
Function linspace(Inizio As Double, Fine As Double, NumeroPunti As Long) As Double()
    Dim x() As Double, Passo As Double, NMenoUno As Long, i As Long
    NMenoUno = NumeroPunti - 1
    ReDim x(NMenoUno)
    Passo = (Fine - Inizio) / NMenoUno
    For i = 0 To NMenoUno
        x(i) = Inizio
        Inizio = Inizio + Passo
    Next
    linspace = x
End Function
' calcola l'integrale definito di y da x(0) a x(N).
Function trapz(ByRef x() As Double, ByRef y() As Double) As Double
Dim N As Long, Passo As Double, i As Long, Q As Double, h As Double
N = UBound(x)
Passo = x(1) - x(0)
h = 0.5 * Passo
Q = 0#
For i = 0 To N - 1
    Q = Q + (y(i + 1) + y(i)) * h
Next
trapz = Q
End Function
' integra l'equazione differenziale di secondo grado y'' = a, con il metodo di Eulero
' x(0...L) = vettore con la variabile indipendente
' y2(0...L) = vettore con la derivata seconda (input)
' y0 = y(0), condizione iniziale (input)
' yp0 = y'(0), condizione iniziale (input)
' y1(0...L) = vettore con la derivata prima (output)
' y(0...L) = vettore con la funzione (output)
Sub Eulero2g(ByRef x() As Double, ByRef y2() As Double, _
               ByVal y0 As Double, ByVal yp0 As Double, _
               ByRef y1() As Double, ByRef y() As Double)
    Dim L As Long, i As Long, Passo As Double, Passoy2 As Double
    L = UBound(x) ' L = indice superiore (supposti di uguali dimensioni N = L+1)
    Passo = x(1) - x(0)
    y(0) = y0    ' i primi elementi dei vettori y e yp
    y1(0) = yp0 ' sono le condizioni iniziali!
    yp0 = yp0 + 0.5 * Passo * (y2(0))
    For i = 1 To L
        Passoy2 = Passo * y2(i)
        y(i) = y(i - 1) + Passo * yp0
        y1(i) = y1(i - 1) + Passoy2
        yp0 = yp0 + Passoy2
    Next
End Sub
Sub test()
    Dim S As String, t0 As Single, Q As Double, i As Long
    t0 = Timer
    Dim x() As Double
    ReDim y2(NP) As Double, yp(NP) As Double, y(NP) As Double
    x = linspace(0, 10, NP + 1)
    For i = 0 To NP
        y2(i) = -0.5
    Next
    Eulero2g x, y2, 0, 2.5, yp, y ' risolve l'equazione differenziale
    Q = trapz(x, y) ' calcola l'integrale per verificare il risultato (41.667)
    S = "Secondi impiegati: " & CSng(Timer - t0) & vbNewLine & _
        "Q = " & CDbl(Round(Q, 3)) & " ."
    MsgBox S, vbInformation, "ODE2 (Metodo di Eulero) - Excel VBA"
End Sub
```

Il programma risulta più lungo e complesso rispetto a Octave, il BASIC non conosce le funzioni linspace e trapz: le dobbiamo programmare noi!



Un linguaggio di programmazione (sia Basic o C++) mi obbliga a non tralasciare alcun dettaglio e a scrivere molto più codice (tempi di sviluppo/codifica più lunghi, più tempo di messa a punto/*debugging*).
Nonostante tutto, Excel è in grado di far girare il calcolo in poco più di due secondi, un tempo molto buono per un linguaggio di supporto a un foglio elettronico.



Visual BASIC.NET

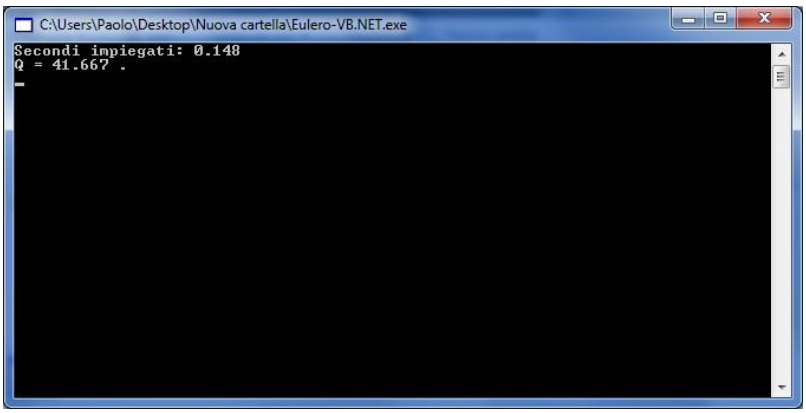
Ecco il codice in Visual BASIC.NET

```
' Eulero.VB - risolve l'equazione differenziale y'' = -1/2 con y'(0)=yp=5/2 e y(0)=y0=0.
Module Module1
    Const NP As UInteger = 10000000 ' dieci milioni di punti
    Function Linspace(ByVal Inizio As Double, ByVal Fine As Double,
                      ByVal NumeroPunti As UInteger) As Double()
        Dim x(), Passo As Double, NMenoUno As UInteger = NumeroPunti - 1
        ReDim x(NMenoUno)
        Passo = (Fine - Inizio) / NMenoUno
        For i As UInteger = 0 To NMenoUno
            x(i) = Inizio
            Inizio += Passo
        Next
        Return x
    End Function
    Function Trapz(ByRef x() As Double, ByRef y() As Double) As Double
        Dim N As UInteger = UBound(x), Passo As Double = x(1) - x(0)
        Dim h As Double = 0.5 * Passo, Q As Double = 0.0
        For i As UInteger = 0 To N - 1 : Q += (y(i + 1) + y(i)) * h : Next
        Return Q
    End Function
    Sub Eulero2g(ByRef x() As Double, ByRef y2() As Double,
                 ByVal y0 As Double, ByVal yp0 As Double,
                 ByRef y1() As Double, ByRef y() As Double)
        Dim L As UInteger = UBound(x)
        Dim Passo As Double = x(1) - x(0)
        y(0) = y0 : y1(0) = yp0
        yp0 += 0.5 * Passo * y2(0)
        For i As UInteger = 1 To L
            Dim Passoy2 As Double = Passo * y2(i)
            y(i) = y(i - 1) + Passo * yp0
            y1(i) = y1(i - 1) + Passoy2
            yp0 += Passoy2
        Next
    End Sub
    Sub Main()
        Dim T As New Stopwatch
        T.Start()
        Dim x() As Double, y2() As Double, yp() As Double, y() As Double
        ReDim y2(NP), yp(NP), y(NP)
        x = Linspace(0.0, 10.0, NP + 1) ' crea uno spazio lineare da 0 a 10
        For i As UInteger = 0 To NP : y2(i) = -0.5 : Next
        Eulero2g(x, y2, 0.0, 2.5, yp, y)
        Dim Q As Double = Trapz(x, y)
        Console.WriteLine("Secondi impiegati: {0}", T.ElapsedMilliseconds * 0.001)
        Console.WriteLine("Q = {0}.", Math.Round(Q, 3))
        Console.ReadKey()
    End Sub
End Module
```

Non meno che con il BASIC di Excel, anche in questo caso non possiamo contare sul fatto che siano note le funzioni linspace e trapz e pertanto dobbiamo preoccuparci noi di scriverne il codice.

Per il resto il codice è simile al precedente (VBA per Excel).

I tempi di esecuzione invece sono molto diversi in quando Excel compila il suo VBA in P-Code mentre il compilatore di Visual BASIC produce il CIL che poi il JIT trasforma in linguaggio macchina nativo.

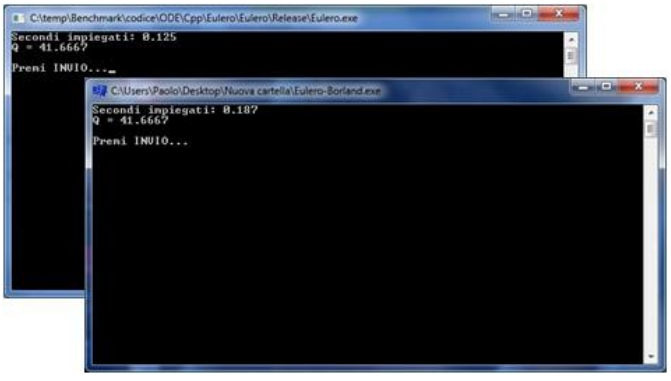


C++

Di seguito il codice in Standard C++, compilabile su qualunque piattaforma dotata di compilatore standard.

```
// Eulero.cpp - Written by Paolo Ferraresi (C) 2012 in Stadard C++
#include <iostream>
#include <iterator>
#include <ctime>
const unsigned int NP = 10000000; // dieci milioni di punti
template <class ForwardIterator, class T>
void linspace(ForwardIterator first, ForwardIterator last, T base,const T limit)
{
        const std::iterator_traits<ForwardIterator>::difference_type N = std::distance(first,last);
        const T passo = (limit-base)/(N-1);
        for(ForwardIterator it = first;it!=last;++it)
        {
                *it = base;
                base+=passo;
        }
}
template <class ForwardIterator,class T>
const T trapz(ForwardIterator xstart,ForwardIterator xlast,ForwardIterator ystart, T Q)
{
        const T h(0.5*(xstart[1]-xstart[0]));
        while(++xstart!=xlast) Q+=(*ystart + *++ystart)*h;
        return Q;
}
void Eulero2g(double* x, double *y2, double y0, double yp0, double* y1, double* y, unsigned int N)
{
        const double passo(x[1]-x[0]);
        *y = y0;
        *y1 = yp0;
        yp0+=0.5*passo*(*y2);
        for(unsigned int i = 1;i<=static_cast<unsigned int>(N-1);++i)
        {
                const double passoy2(passo*y2[i]);
                y[i]=y[i-1]+passo*yp0;
                y1[i]=y1[i-1]+passoy2;
                yp0+=passoy2;
        }
}
int main(int argc, char* argv[])
{
        const std::clock_t start = std::clock();
        double* x = new double[NP+1];                   // alloca l'array x (NP+1 punti)
        double* y2 = new double[NP+1];                  // alloca l'array y2 (per la y'')
        double* y1 = new double[NP+1];                  // alloca l'array y1 (per y')
        double* y = new double[NP+1];                   // alloca l'array y
        linspace(x,x+NP+1,0.0,10.0);                    // inizializza lo spazio lineare da 0 a 10
        for(unsigned int i=0;i<NP+1;++i) y2[i]=-0.5;    // riempie y2 con la derivata seconda
        Eulero2g(x,y2,0.0,2.5,y1,y,NP+1);               // integra l'equazione differenziale
        double Q = trapz(x,x+NP+1,y,0.0);               // calcola Integrate(y,x)
        std::cout << "Secondi impiegati: " << static_cast<float>(std::clock()-start)/CLK_TCK << "\n";
        std::cout << "Q = " << Q << "\n\nPremi INVIO...";
        delete [] x; delete [] y2; delete [] y1; delete [] y; // rilascia la memoria
        std::cin.clear(); // svuota il buffer per evitare che venga preso un tasto premuto prima
        std::cin.get(); // legge un carattere da console input
        return 0;
}
```

Per non fare torto agli altri codici non sono stati utilizzati container STL (come ad esempio std::vector<double>) ma soltanto i tipi primitivi, quindi, array di double.
In questo modo i codici risulteranno ancora più "simili" a come potrebbero essere scritti con l'assembler e quindi: molto diretti.
Il Visual C++ produce un codice che impiega 0.125 s, il Borland C++ 0.187 s.



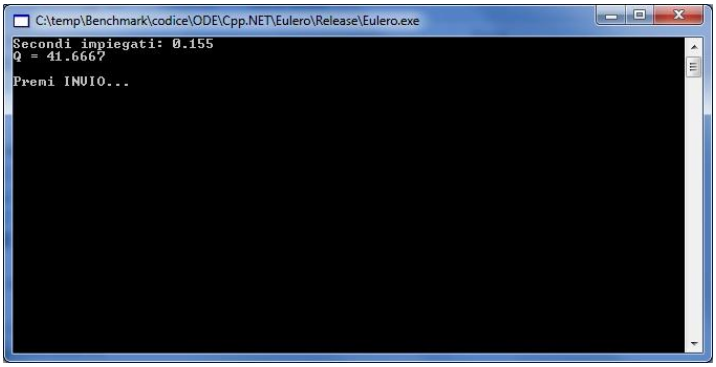


Managed C++

Il codice è identico a prima tranne il fatto che si è dovuta aggiungere in testa la seguente direttiva:

```
// Eulero.cpp per Visual C++ + CLR
#include "stdafx.h" // aggiungere questa riga
...
```

Il resto del codice è identico, da #include <iostream> fino alla fine.
Compilando il programma e lanciandolo in esecuzione si ottiene il tempo di 0.155 s.



Risultati

Mettendo a confronto le prestazioni delle varie soluzioni otteniamo la seguente tabella (tempo in secondi):

Soluzione	Sieve	ODE
C++ Visual Studio 2010	1.09	0.125
C++ Borland C++ Builder 6	3.32	0.187
Managed C++ (V.S. 2010)	1.59	0.155
VB.NET (Visual Studio 2010)	1.36	0.148
Excel 2007 + VBA	7.84	2.21
Octave	1624	295

Come vediamo, purtroppo, **Octave** ottiene risultati "imbarazzanti" rispetto a tutte le altre soluzioni. **Fortunatamente la verità sta a metà strada, ovvero, in parte le cose stanno proprio così: il codice eseguito da Octave è effettivamente molto più lento rispetto a qualunque altra soluzione "compilata" (anche in P-Code)**, tuttavia, è arrivato il momento di recuperare una delle mie citazioni preferite : -

"**Ognuno di noi è un genio, ma se si giudica un pesce dalla sua abilità di arrampicarsi sugli alberi, lui passerà tutta la sua vita a credersi stupido**." (***Albert Einstein***).

Semplicemente è sbagliato utilizzare Octave come qualunque altro linguaggio di programmazione.
Ad esempio, a un certo punto del codice che ho scritto per Octave (Crivello di Eratostene) si trova:

```
    for j=uint32(i*i:i:N) % elimina tutti i multipli di i con molteplicità maggiori o uguali a i fino a N
      v(j)=0; % mettendo il numero nella lista a zero
    end
```

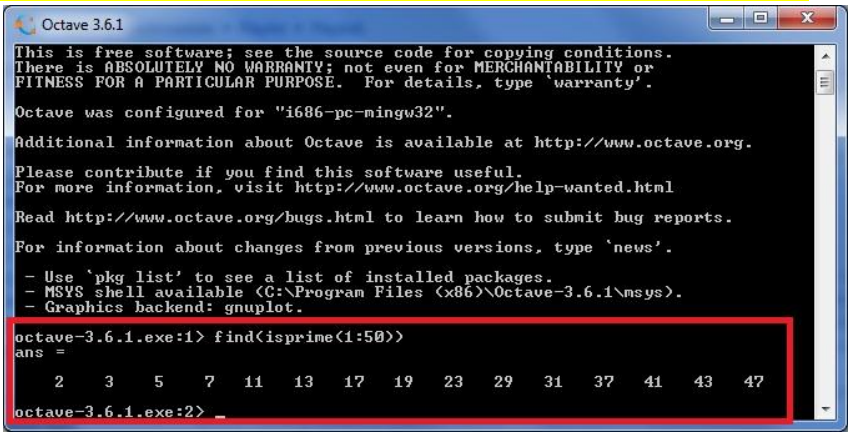
questo è quello che si farebbe con un normale linguaggio di programmazione, Octave invece supporta le operazioni "vettoriali" per cui è sufficiente scrivere qualcosa del tipo:

$$v_{i^2,i^2+i,\ldots,N} = 0$$

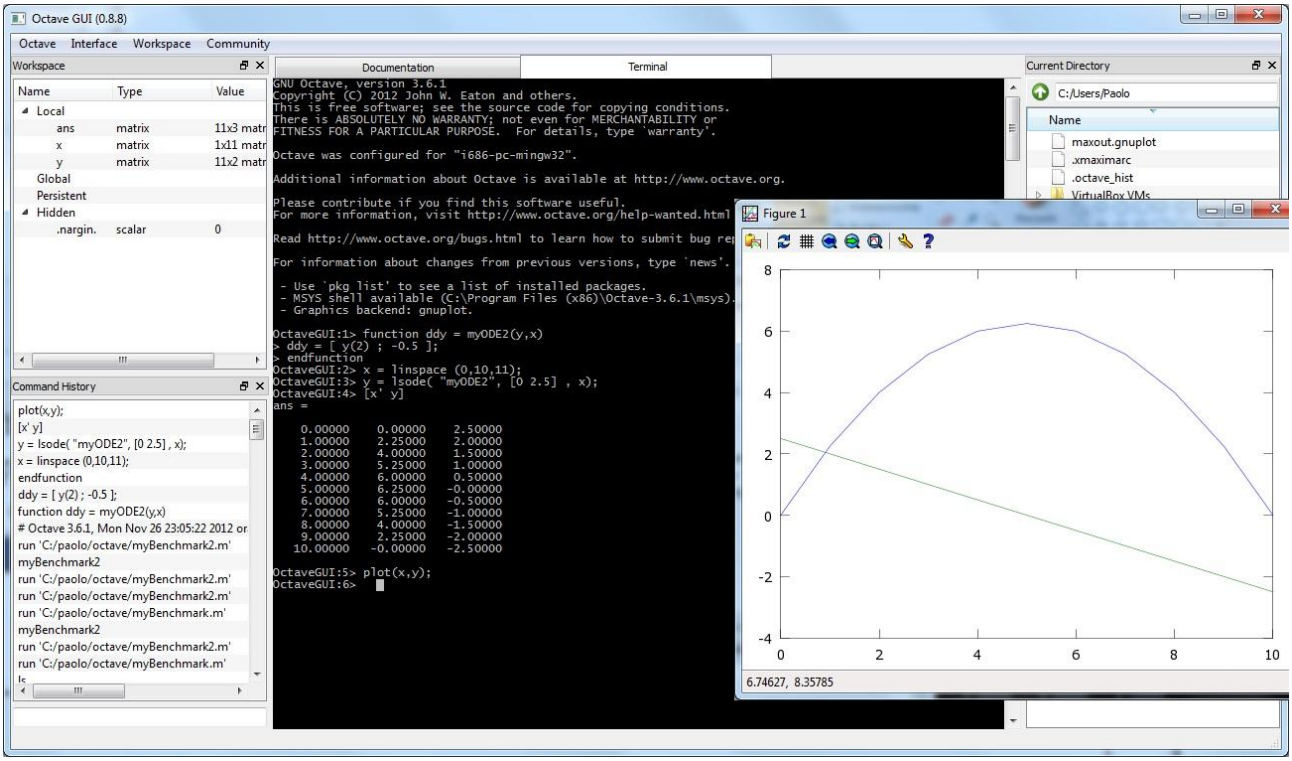
Per azzerare tutti gli indici richiesti senza utilizzare un ciclo for/next, infatti scrivendo:

```
    v(i*i:i:N) = 0;
```

si riducono i tempi al 70%.
Octave inoltre conosce tutti gli algoritmi numerici, quindi, esiste già un test di primalità (**isprime**). E' semplicissimo ottenere un vettore di numeri primi. Ecco un esempio che mostra i numeri primi presenti nell'intervallo da 1 a 50, in **una riga di codice!** La velocità? Le funzioni **isprime**, **find** e tutte le altre native sono scritte in linguaggio macchina ottimizzato per SSE2 o AMD64 e quindi va molto veloce.



Per l'equazioni differenziali, va ancora meglio. Octave con 5 righe di codice ci regala anche il grafico!!!



Il codice "corretto" per risolvere la nostra equazione, è questo:
Ora Octave impiega 6 secondi.
Può sembrare ancora tanto ma...consideriamo che la funzione **lsode** utilizza metodi ben più sofisticati[6] di Eulero e può quindi integrare anche problemi "***stiff***" altrimenti inavvicinabili, con grande accuratezza. Vale la pena di attendere qualche secondo in più.

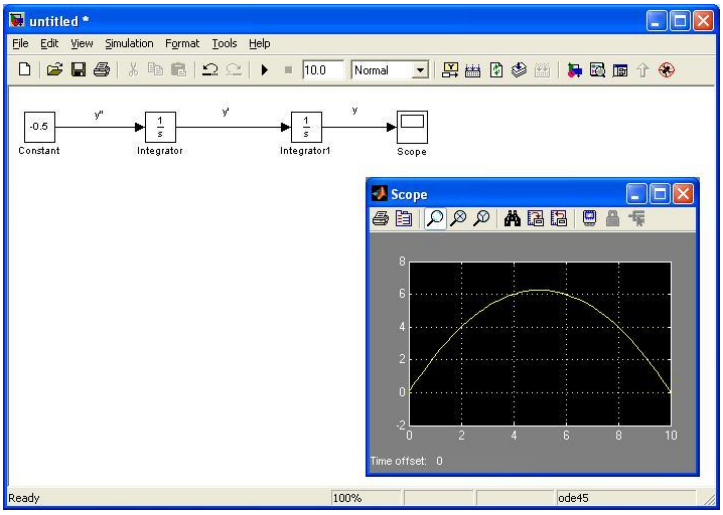
```
% risolve l'equazione differenziale y'' = -1/2 con y'(0)=yp=5/2 e y(0)=y0=0.
1;

N = 10000000; % dieci milioni di punti

function ddy = myODE(y,x)
        ddy = [y(2) ; -0.5];
endfunction

t0 = clock();
x = linspace(0,10,N+1); % alloca uno spazio lineare di N+1 punti da 0 a 10
y = lsode("myODE",[0 2.5],x); % risolve l'equazione differenziale myODE

Q = trapz(x',y(:,1)) % calcola l'integrale y(x) da x = 0 a x = 10
printf("\nSecondi impiegati: %d\n",etime(clock(),t0));
```



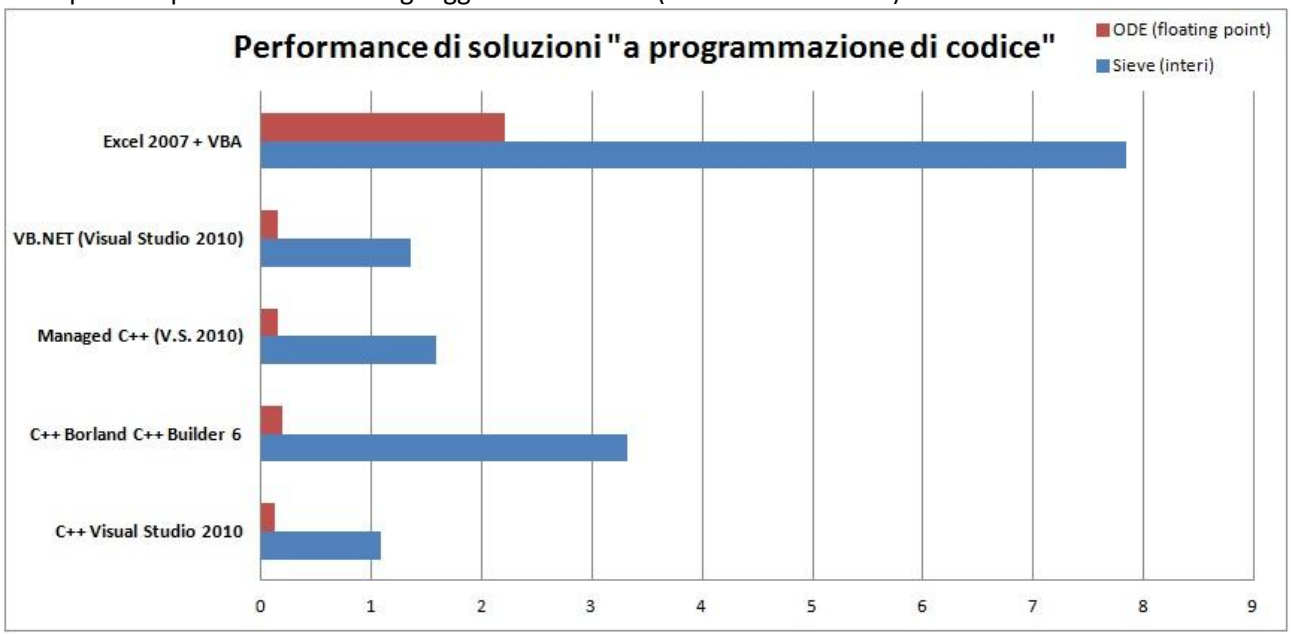
Se poi usiamo "Matlab", grazie a Simulink (puoi trovare un mio articolo qui, se sei interessato) otteniamo il risultato di destra, in modo completamente "Visual", cioè, senza dover scrivere una sola riga di codice.

[6] Alan C. Hindmarsh, ODEPACK, *A Systematized Collection of ODE Solvers, in Scientific Computing*, R. S. Stepleman, editor, (1983).

Quindi Octave è una miniera d'oro.
Anche se un programma in C++ impiega un secondo contro un'ora di Octave, con un linguaggio di programmazione come minimo ci lascio diverse ore soltanto per scrivere il codice quando invece con il primo impiego pochi minuti.
Per questi motivi non renderemmo giustizia ad Octave (né a Matlab) se li confrontassimo con le altre soluzioni, per cui, mi sento di toglierle dal confronto.
Quand'è che invece vale la pena di adoperare il C++ o Visual BASIC? quando un algoritmo deve essere lanciato molto frequentemente! Più volte al giorno. Allora sì che i tempi d'elaborazione "super rapidi" ripagano le ore perse a sviluppare il codice. Per Excel le cose invece "stanno" a metà strada e cioè, lo sviluppo sarà tanto più rapido quanto riusciremo ad usare le funzioni già disponibili nel foglio elettronico (in pratica conviene se il grosso del lavoro è in grado di farlo Excel e *la parte in VBA è soltanto una integrazione*).

Conclusioni

Ecco quindi le performace del "linguaggi" usati nel test (escludendo Octave).



Questi sono tutti "compilati" e come era logico attendersi:

1. Si conferma che il **P-Code non è veloce come il codice nativo**.

Tuttavia:

2. Excel 2007 + VBA rimangono una scelta eccellente per tanti lavori in cui servono molte delle funzionalità messe a disposizione da uno ***spreadsheet*** (es. tabelle, formule e grafici)!

Venendo invece alle soluzioni che prevedono la scrittura di un programma vero e proprio, osserviamo che al primo posto (top delle prestazioni) e al quarto abbiamo due compilati in codice nativo, uno abbastanza recente (Visual C++ 2010) e l'altro di qualche anno fa (Borland, versione del 2002).
Al secondo e terzo posto abbiamo invece la soluzione "Ibrida" con programmi che girano per la piattaforma Microsoft .NET, da cui possiamo trarre le seguenti conclusioni:

3. **Più o meno le prestazioni del managed C++ e di Visual BASIC.NET si equivalgono e sono molto buone se comparate al miglior codice nativo**.

Su questo Microsoft aveva dato assicurazione e direi che la promessa è stata mantenuta (la principale differenza sta nel tempo impiegato dalla compilazione JIT all'avvio del programma).
A conferma di quello che dicevamo:

4. **il codice nativo "classico" (*unmanaged*) se teoricamente è il più veloce, in pratica si deve guardare alla qualità e al grado d'ottimizzazione del codice compilato** poiché in certi ambiti è facile assistere a vere e poprie "sorprese".

Quindi il mio consiglio è quello di non lasciarsi influenzare dai pregiudizi ma tenere a mente che comunque **è la qualità del codice sorgente a rendere ottimo o pessimo un programma**. *Se ad esempio uso un Bubble Sort per ordinare un database, sarà lento a prescindere dal linguaggio o dal compilatore impiegato, lento persino se lo programmo con l'Assembler mentre un merge sort sarà "decente" anche se scritto in BASIC.*
Trovare sempre un equilibrio tra le necessità (tempi di sviluppo e prestazioni richieste): non serve perdere tempo per scrivere "il programma perfetto", vale sempre la regola che il 20% del codice è responsabile dell'80% dei tempi d'esecuzione, anzi, di solito anche meno!
Più che scrivere programma "stra-ottimizzati" **è meglio scrivere programmi affidabili** (che facciano quello per cui si propongono) e ben commentati per poter intervenire in un secondo momento, se necessario.
Se poi vi sono necessità "velocistiche" allora, un buon compilatore "moderno" migliora le cose, a patto che il nostro codice sia scritto bene!
Saluti e... buon codice a tutti!

Paolo Ferraresi.