UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Dipartimento di
Scienze Fisiche, Informatiche e Matematiche

Corso di Laurea in Fisica

Tesi di Laurea

Profiling e ottimizzazione del modulo biogeochimico del sistema previsionale Copernicus del Mar Mediterraneo

Relazione di tirocinio

Relatore
Prof. Mauro Ferrario

Correlatori:
Dott. Eric Pascolo
Dott. Paolo Lazzari

Laureando
Simone Vacondio

Tutore Aziendale
Dott. Fabio Affinito

Anno accademico 2015–2016

Composto con pdfLaTeX il 12 ottobre 2016

Sommario

Uno dei temi cruciali che coinvolge la popolazione mondiale è legato alle trasformazioni che stanno interessando il sistema Terra nella storia contemporanea. La comunità scientifica ha rilevato e documentato (IPCC reports) che sono in atto rapidi cambiamenti climatici, uno su tutti l'innalzamento della temperatura. L'ambiente in cui viviamo è un sistema vasto e complesso, e per stimare con precisione il suo stato, e le sue eventuali alterazioni, sono necessari sistemi osservativi in grado di misurare parametri fisici di interesse su scala globale. Oltre a grandi quantità di dati, sono necessari modelli numerici che possano integrare tutte le informazioni acquisite, ricostruire lo stato attuale del sistema e fornire scenari di cambiamento climatico per i decenni futuri. In tale contesto l'Unione Europea ha lanciato uno dei più avanzati programmi per il monitoraggio e la previsione ambientale: il sevizio Copernicus[1].

Uno degli ambienti più inesplorati ed importanti per la nostra vita è quello marino. Dal punto di vista economico le aree costiere sono strategiche perché qui si concentrano la maggior parte delle attività umane. La conoscenza e la previsione di tutti i processi fisici, chimici e biologici che avvengono nell'ecosistema marino sono essenziali per valutare lo stato del mare, evitare danni ambientali e creare programmi per uno sviluppo sostenibile. L' OGS (Istituto Nazionale di Oceanografia e Geofisica Sperimentale), all'interno del quale ho svolto il mio tirocinio, sviluppa e mantiene la catena operativa per il forecasting biogeochimico del Mar Mediterraneo nel quadro del servizio europeo Copernicus.

Il cuore della catena operativa per la previsione biogeochimica del Mar Mediterraneo è il reattore biogeochimico implementato attraverso il software BFM. Per fornire analisi sempre più accurate e in minor tempo è essenziale disporre di software altamente efficienti ed ottimizzati che possano essere utilizzati su cluster ad alte prestazioni, quindi ad ogni revisione della parte scientifica del software devono corrispondere un nuovo profiling ed una ottimizzazione del codice.

Il mio lavoro di tesi ha riguardato in una prima fase l'apprendimento delle nozioni principali relative al calcolo ad alte prestazioni (High Performance Computing - HPC). Questo conoscenze mi sono servite, nella seconda fase, a condurre lo studio della performance e la definizione di una strategia di ottimizzazione di BFM-V5, che verrà utilizzato per la futura versione del servizio Copernicus.

[1] http://www.copernicus.eu/

Per questo ho lavorato sulla medesima macchina utilizzata dal sistema previsionale OGS: il cluster HPC PICO in CINECA. Utilizzando il software Intel VTune Amplifier XE 2016, ho analizzato le subroutine maggiormente "time-consuming" e localizzato i nuovi bottleneck introdotti dalla revisione scientifica del codice; inoltre ho identificato flag di compilazione che creavano rallentamenti sulla nuova versione rispetto alle precedenti. La strategia proposta in questo lavoro di tesi, che combina la modifica delle flag e la riscrittura di operazioni tra array aventi dimensione degenere, ha portato ad uno speedup di 2x, cioè un dimezzamento dei tempi di calcolo. Ho inoltre proposto sviluppi futuri di ottimizzazione per aumentare le performance di BFM-V5.

Indice

Sommario III

1 Introduzione al Biogeochemical Flux Model 1
- 1.1 Il progetto BFM . . . 2
- 1.2 Elementi di modellistica quantitativa degli ecosistemi . . . 3
- 1.3 La formulazione matematica di BFM . . . 7

2 Elementi di HPC 11
- 2.1 Struttura di un cluster HPC . . . 11
- 2.2 Domain Decomposition e sua applicazione in OGSTM-BFM . . . 13
- 2.3 Modalità di ottimizzazione . . . 14
- 2.4 La pipeline . . . 15
- 2.5 L'architettura Sandy Bridge . . . 18
- 2.6 Classificazione delle performance della pipeline . . . 18
- 2.7 Ottimizzazioni apportate dal compilatore . . . 21

3 Profiling e ottimizzazione di BFM 23
- 3.1 Informazioni preliminari . . . 23
  - 3.1.1 Hardware e software . . . 23
  - 3.1.2 Benchmark . . . 23
  - 3.1.3 Metodo di lavoro . . . 24
- 3.2 Profiling e ottimizzazione di BFM STANDALONE . . . 26
  - 3.2.1 Primo profiling di BFM STANDALONE . . . 26
  - 3.2.2 Modifica delle flag di compilazione . . . 28
  - 3.2.3 Riduzione di una dimensione di alcune operazioni tra array . . . 31
- 3.3 Profiling e ottimizzazione di OGSTM-BFM . . . 41
- 3.4 Riassunto dei risultati ottenuti . . . 44

4 Altre prove su BFM 47
- 4.1 Impiego di subroutine ausiliarie in `flux_vector` . . . 47
- 4.2 Rimozione del controllo dell'allocazione all'interno di `flux_vector` . . . 53

Conclusioni 59

Bibliografia 63

Capitolo 1

Introduzione al Biogeochemical Flux Model

Uno degli ambienti più inesplorati ed importanti per la nostra vita è quello marino. Dal punto di vista economico le aree costiere sono strategiche perché qui si concentrano la maggior parte delle attività umane. La conoscenza e la previsione di tutti i processi fisici, chimici e biologici che avvengono nell'ecosistema marino sono essenziali per valutare lo stato del mare, evitare danni ambientali e creare programmi per uno sviluppo sostenibile.

I sistemi previsionali attuati da istituti come OGS (Istituto Nazionale di Oceanografia e Geofisica Sperimentale), presso cui è stato svolto il tirocinio esposto in questo documento, affondano le loro radici nella *modellistica quantitativa degli ecosistemi*, un approccio allo studio dell'ecologia che ha acquisito una sempre maggior importanza nel corso della storia [16]. Questo approccio fa largo uso della matematica come scienza predittiva, e si prefigge l'obiettivo di prevedere l'evoluzione delle popolazioni di un ecosistema nel corso del tempo. Nel caso specifico di OGS, la modellistica dei sistemi marini è applicata "ai temi della Biogeochimica ed Ecologia Marina, l'Impatto Antropico e dei Cambiamenti Climatici sui sistemi marini, l'Approccio Ecosistemico alla Pesca ed Acquicoltura, lo Sviluppo Sostenibile e la Gestione Integrata della Zona Costiera, l'Oceanografia Operazionale ed Assimilazione di Dati, l'analisi dei sistemi integrati Ecologico-Socio-Economici" [1].

Una risorsa fondamentale per la modellistica quantitativa è oggi l'HPC (High Performance Computing, ovvero calcolo ad alte prestazioni), poiché la crescente complessità dei modelli numerici ha portato ad un notevole costo computazionale degli stessi. Ne consegue che, per poter ottenere i risultati di una previsione in tempi ragionevoli, è necessaria una progettazione dei suddetti modelli orientata al calcolo parallelo e al lancio su cluster computer adibiti al calcolo scientifico. L'oggetto di questa tesi è proprio proporre soluzioni per la versione 5 del Biogeochemical Flux Model (BFM-V5), il modulo biogeochimico del sistema previsionale Copernicus per il Mar Mediterraneo, al cui sviluppo partecipa OGS, affinché esso possa essere meno costoso dal punto di vista computazionale.

[1] http://www.ogs.trieste.it/it/content/echo-modellistica-dei-sistemi-marini

In questo capitolo sarà introdotto il progetto BFM e sarà fornito qualche concetto di base sulla modellistica quantitativa degli ecosistemi. A partire da una breve rassegna di modelli notevoli, volta ad evidenziare le varie complicazioni che possono essere incorporate in un modello, verrà poi introdotta la formulazione matematica del Biogeochemical Flux Model.

1.1 Il progetto BFM

Il *Biogeochemical Flux Model* (BFM) è un modello numerico, che descrive la dinamica dei principali processi biogeochimici che interessano l'ecosistema marino. Il codice, scritto in Fortran 90, è rilasciato come software libero sotto la licenza GNU GPLv3; i sorgenti ed il manuale [25] sono liberamente scaricabili dal sito `www.bfm-community.eu`. Il progetto è sviluppato grazie al supporto di un consorzio formato dall'Università di Bologna (UNIBO), l'Istituto di Oceanografia e Geofisica Sperimentale (OGS), il Centro Euro-Mediterraneo sui Cambiamenti Climatici (CMCC) e l'Università di Cape Town (UCT).

Il BFM è nato nel 2002, deriva dell'European Regional Seas Ecosystem Model (ERSEM) [2] e condivide con esso la maggior parte delle caratteristiche nella loro formulazione originale. Rispetto ad ERSEM, BFM si concentra maggiormente sulla biogeochimica dei livelli trofici più bassi degli ecosistemi marini. Al momento BFM considera i cicli di azoto, fosforo, silicio, carbonio e ossigeno nella fase disciolta in acqua, nonché in plancton, detriti e dominio bentonico [25]. Le dinamiche del plancton sono parametrizzate considerando un numero di gruppi funzionali: produttori (fitoplancton), consumatori (zooplancton) e decompositori (batteri). Queste classificazioni funzionali generali sono ulteriormente partizionate in sottogruppi funzionali per creare una rete trofica planctonica (diatomee, picofitoplancton, microzooplancton...).

BFM può essere accoppiato a modelli per l'idrodinamica (NEMO [13], POM [15], OPA [14], GOTM [22]), in modo da produrre simulazioni su un dominio spaziale tridimensionale. Uno dei modelli usati e sviluppati dal gruppo ECHO di OGS è OGSTM, un'evoluzione del modello OPA [12]. Le equazioni di trasporto per il modello 3D possono essere risolte a diverse scale e risoluzioni spaziali, in modo da poter considerare come domini tanto i mari (ad esempio il Mediterraneo), o loro regioni (ad esempio l'Adriatico), quanto gli oceani. Gli archi temporali di impiego del modello sono allo stesso modo molto flessibili: essi variano dalla previsione a breve termine settimanale alla simulazione di scenari multidecennali. BFM è poi progettato per essere modulare: il numero di costituenti della rete trofica può essere modificato, inoltre le stesse reti trofiche possono essere modificate in base all'ecosistema considerato (pelagico, bentonico, mare ghiacciato...).

Grazie a flessibilità e modularità, BFM ha trovato numerose applicazioni in diversi contesti e ad ogni scala: simulazioni della biogeochimica globale [24, 23], simulazioni del mare ghiacciato applicate al Mar Baltico [20] e al Mare Artico [19], simulazioni nel Mediterraneo... In Figura 1.1 sono riportati i risultati di uno studio della variazione spaziotemporale dell'alcalinità del Mediterraneo [3], ottenuti grazie all'uso di BFM.

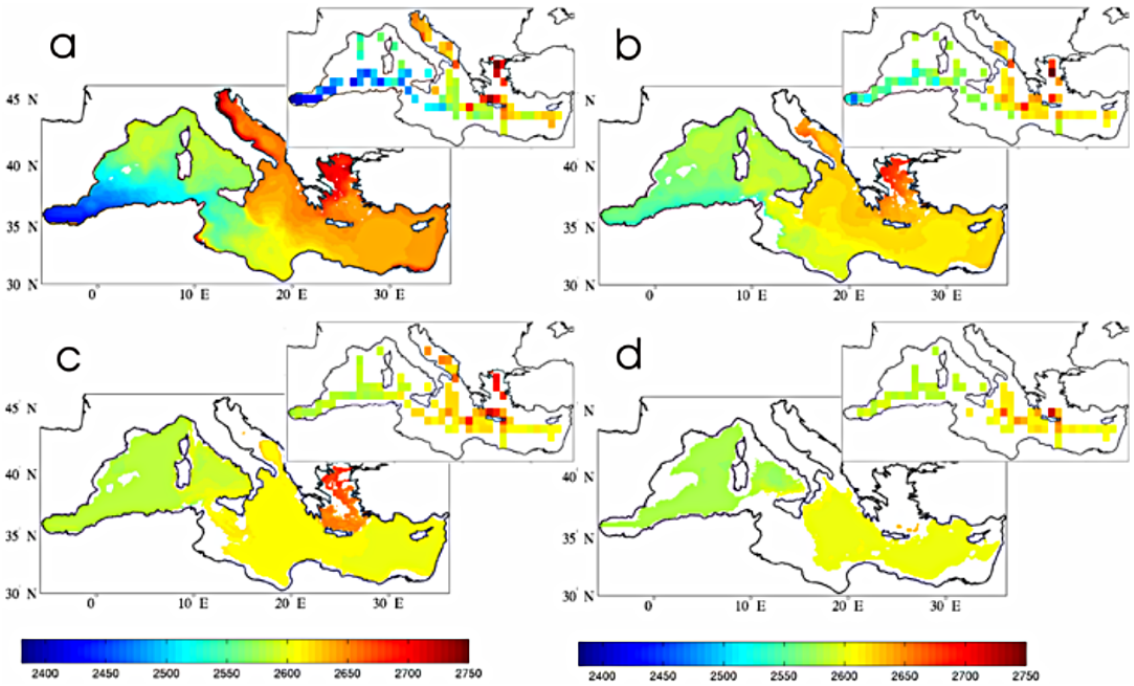


Figura 1.1. Mappe di alcalinità per strati: superficie (**a**), 100-200 m (**b**), 400-1500 m (**c**) e 1500-4000 m (**d**). Le mappe sono riferite alle medie ottenute dal modello. In alto a destra di ogni mappa sono riportate le mappe ottenute da rilevamenti reali. (G. Cossarini, P. Lazzari e C. Solidoro. «Spatiotemporal variability of alkalinity in the Mediterranean Sea». In: *Biogeosciences* (2015))

1.2 Elementi di modellistica quantitativa degli ecosistemi

Vedere qualche modello notevole prima di introdurre le equazioni alla base di BFM è senz'altro utile a contestualizzare lo stesso, in modo da poter apprezzare le complicazioni che esso deve incorporare per modellare l'ecosistema marino. Una rassegna più completa di modelli adottati per studiare la dinamica delle popolazioni possono essere trovati in J. D. Murray. *Mathematical Biology: I. An Introduction, Third edition.* 2002 e in A. B. Ryabov e B. Blasius. «Population growth and persistence in a heterogeneous environment: the role of diffusion and advection». In: *Mathematical Modeling of Natural Phenomena* (2008), dai quali sono stati tratti in gran parte i cenni che seguono.

Il primo modello di dinamica delle popolazioni introdotto nella storia è il *modello malthusiano*. Il modello è basato su un'equazione di conservazione della popolazione, secondo cui il tasso di crescita del numero di individui è proporzionale al numero di individui stesso. Denotando con P il numero di individui, con b il tasso di nascite per individuo e con d il tasso di morti per individuo, l'equazione di conservazione assume la forma:

$$\frac{dP}{dt} = bP - dP \tag{1.1}$$

e risulta essere una semplice equazione differenziale ordinaria con soluzione esponenziale:

$$P(t) = P_0 e^{(b-d)t}$$

dove la popolazione iniziale è $P(0) = P_0$. Questo modello, introdotto da T. R. Malthus nel 1798 nel suo *Saggio sui principi della popolazione*, è piuttosto grossolano: esso trascura completamente le complicazioni introdotte dalla distribuzione spaziale degli individui e dall'interazione con altre specie. Al tempo della sua ideazione era semplicemente pensato per prevedere la crescita della popolazione umana. Fu anche accompagnato da diverse controversie, poiché Malthus "rassicurava" che, nonostante la crescita esponenziale, la popolazione mondiale sarebbe comunque stata contenuta da guerre, pestilenze e carestie. Critiche a parte, incidentalmente la popolazione mondiale dal 1900 a oggi è cresciuta esponenzialmente.

Il modello malthusiano rientra nella classe dei modelli non-spaziali sintetizzabili con un'equazione differenziale del tipo:

$$\frac{dP}{dt} = \mu(P)P \tag{1.2}$$

dove il tasso di crescita $\mu(P)$ è funzione della popolazione P. Nel modello malthusiano la funzione $\mu(P)$ è costante. Un modello più raffinato, proposto da Verhulst (1838, 1845), mette in conto il fatto che nel lungo termine debba esserci una sorta di aggiustamento alla crescita esponenziale: tale modello è quello a *crescita logistica*, in cui $\mu(P) = \mu_0 \left(1 - \frac{P}{K}\right)$:

$$\frac{dP}{dt} = \mu(P)P = \mu_0 P \left(1 - \frac{P}{K}\right) \tag{1.3}$$

dove μ_0 e K sono costanti positive. Le soluzioni presentano due stati di equilibrio: $P = 0$ e $P = K$. Lo stato $P = 0$ è instabile, perché una piccola variazione dal valore 0 dà origine ad una crescita esponenziale della popolazione nel breve termine, in accordo col modello malthusiano (una piccola popolazione infatti rende trascurabile il termine $\frac{P}{K}$ nell'equazione differenziale, riportando alla (1.1)). Lo stato $P = K$ è invece di equilibrio stabile ed è quello a cui tendono le soluzioni con popolazione iniziale non nulla. Nonostante sia stata accolta più volte nel corso della storia come legge generale dell'evoluzione delle popolazioni, la crescita logistica può fornire andamenti qualitativi nei casi in cui $P = 0$ sia uno stato stazionario instabile e $P(t)$ tenda ad uno stato stazionario stabile. In generale, funziona bene per la sua semplicità algebrica e per dare un'idea preliminare di ciò che può accadere in modelli più realistici. Un raffinamento dei modelli fin qui presentati include la *finitezza delle risorse* disponibili per la crescita di una popolazione. Un semplice modello conservativo è dato dal seguente sistema di equazioni:

$$\frac{dP}{dt} = \mu(N)P - mP \tag{1.4}$$

$$\frac{dN}{dt} = -\mu(N)P + mP \tag{1.5}$$

in cui P è la popolazione di individui e N la quantità di risorse. Essendo il modello conservativo, tutta la biomassa derivante dalla morte degli individui (termine $-mP$ nella prima

equazione) torna ad essere risorsa disponibile per il consumo (termine $+mP$ nella seconda equazione); in modo analogo il termine $\mu(N)P$ costituisce un intake per gli individui e un outtake per le risorse. Grazie alla conservazione della biomassa, il sistema ammette un ovvio integrale primo: $N + P = \text{cost.}$.

In questo modello, in genere si assumono l'annullamento di $\mu(N)$ per piccole concentrazioni ($\mu(0) = 0$) e una sua saturazione per grandi N. Una parametrizzazione conveniente è data da:

$$\mu(N) = c\frac{N}{H_N + N} \tag{1.6}$$

con H_N la costante di mezza saturazione. Con questa scelta di μ, la traiettoria del sistema nello spazio delle fasi (P, N) risulta essere una linea retta, dalla condizione iniziale allo stato di equilibrio (Figura 1.2). Non tutti gli ecosistemi tendono però ad uno stato di

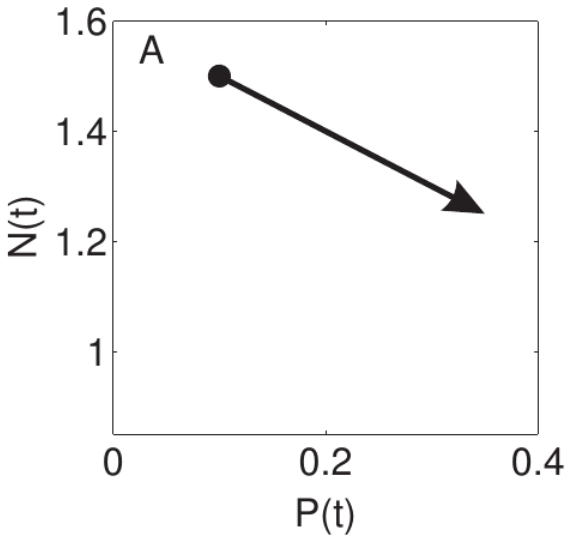


Figura 1.2. Traiettoria nello spazio delle fasi del modello a risorse finite (A. B. Ryabov e B. Blasius. «Population growth and persistence in a heterogeneous environment: the role of diffusion and advection». In: *Mathematical Modeling of Natural Phenomena* (2008))

equilibrio, e il *modello Lotka-Volterra* [26], sviluppato negli anni Venti del Novecento indipendentemente da Lotka e Volterra, è stato pensato per spiegare l'andamento oscillante di certe popolazioni. Esso introduce nelle equazioni una dinamica predatore-preda:

$$\frac{dN}{dt} = N(a - bP) \tag{1.7}$$

$$\frac{dP}{dt} = P(cN - d) \tag{1.8}$$

con a, b, c e d costanti positive. Nella formulazione presentata N rappresenta la popolazione delle prede e P quella dei predatori. Sono ben riconoscibili i tassi di nascita a delle prede e di morte d dei predatori, inseriti nelle equazioni allo stesso modo in cui erano presenti nel modello malthusiano. Il tasso di nascita dei predatori è però ora dipendente

dall'abbondanza delle prede, essendo dato dal prodotto cN; allo stesso modo il tasso di morte delle prede dipende dall'abbondanza dei predatori, essendo dato da bP. La periodicità che caratterizza le soluzioni di questo modello fa sì che la traiettoria del sistema nello spazio delle fasi (P, N) sia un'orbita chiusa (Figura 1.3).

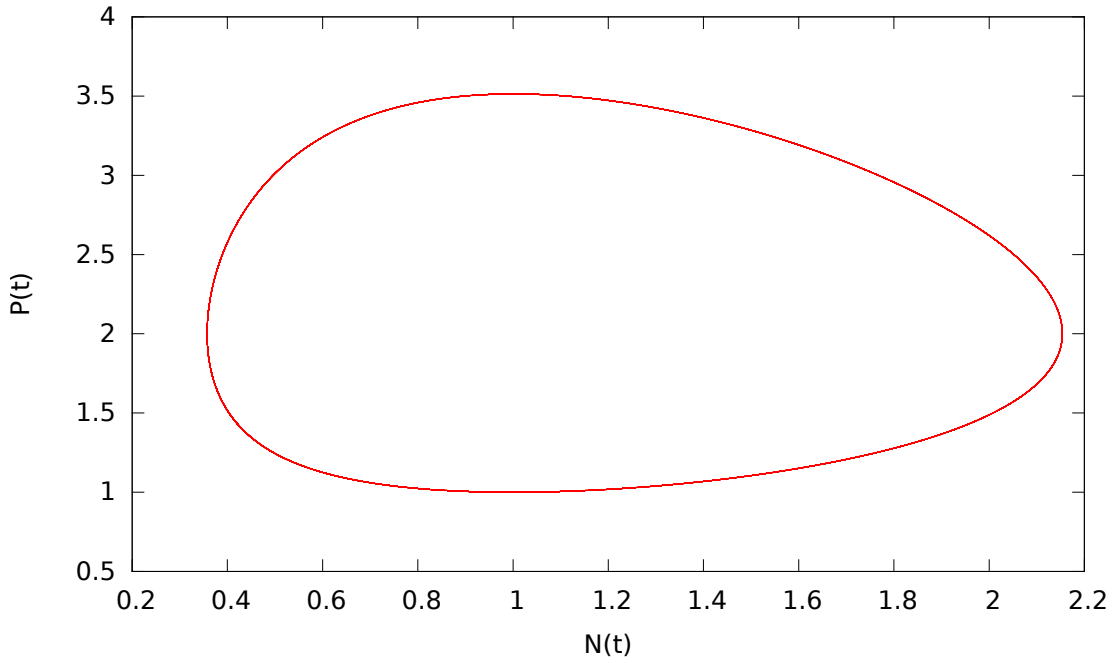


Figura 1.3. Traiettoria nello spazio delle fasi del modello Lotka-Volterra

Modelli ancora più raffinati tengono conto della *competizione* [21] che può crearsi tra più specie che consumano una stessa risorsa, e del fatto che una specie che può consumare diversi tipi di risorse possa adattare la propria dieta in base alla loro disponibilità, subordinata sia all'ambiente, sia al consumo da parte di altre specie.

Infine, un'ulteriore complicazione che si puó introdurre è l'*eterogeneità spaziale* dell'ecosistema. In un modello unidimensionale, si può considerare la variazione della popolazione lungo un asse non solo a causa dei tassi di nascita, morte e predazione, ma anche a causa della diffusione e dell'avvezione. La popolazione è funzione della coordinata x e del tempo t, e la sua dinamica è descritta da una equazione di reazione-diffusione-avvezione:

$$\begin{aligned}\frac{\partial P(x,t)}{\partial t} &= \text{riproduzione} - \text{avvezione} + \text{mescolamento} \\ &= \mu(x,P)P - v\frac{\partial P}{\partial x} + \frac{\partial}{\partial x}D\frac{\partial P}{\partial x}\end{aligned} \tag{1.9}$$

dove μ rappresenta il tasso di crescita (come nella (1.3)), v è la velcità di avvezione e D è la diffusività, che può dipendere da x o altre variabili. Il termine avvettivo può essere originato da diversi tipi di processi fisici o biologici: ad esempio, in un sistema verticale gli organismi possono scendere o salire di quota perché sono più pesanti o più leggeri del

mezzo in cui sono immersi; oppure il termine avvettivo può essere originato da correnti in un fiume o in un oceano.

Quest'ultimo modello può essere utilizzato per studiare un sistema particolarmente rilevante per arrivare a introdurre il BFM: la *distribuzione verticale di fitoplancton*. Tale sistema è significativo perché il fitoplancton è il produttore primario in quasi tutte le reti trofiche marine. I due principali fattori limitanti per la produzione di fitoplancton sono la disponibilità di nutrienti e di luce. L'importanza di un modello che tenga conto dell'eterogeneità spaziale sta nel fatto che dove è abbondante la luce non lo sono altrettanto i nutrienti, e viceversa. La luce è infatti abbondante negli strati più vicini alla superficie della colonna d'acqua, mentre i nutrienti sono localizzati negli strati profondi, perché la biomassa morta tende a sedimentare e solo dopo a tornare in fase inorganica, disponibile per la fotosintesi. Conseguentemente, il fitoplancton tenderà a concentrarsi in una zona intermedia, né troppo superficiale, né troppo profonda, dove ci sia un'abbondanza sufficiente sia di luce, sia di nutrienti.

Orientando l'asse z dalla superficie dell'acqua verso il fondo, la (1.9) si adatta alla distribuzione verticale di fitoplancton come segue:

$$\frac{\partial P(z,t)}{\partial t} = \mu(N,I)P - mP - v\frac{\partial P}{\partial z} + \frac{\partial}{\partial z}D\frac{\partial P}{\partial z} \tag{1.10}$$

dove m è il tasso di mortalià, v la velocità di affondamento del fitoplancton e D la diffusività. Il tasso di crescita $\mu(N,I)$ dipende dai valori locali di intensità della luce $I(z,t)$ e dalla concentrazione di nutrienti $N(z,t)$; inoltre può essere parametrizzato come nella (1.6), una volta fattorizzati i due contributi dell'intensità di luce e della concentrazione di nutrienti:

$$\mu(N,I) = \mu_0 f_I(I) f_N(N) \tag{1.11}$$

$$f_I(I) = \frac{I}{H_I + I},\ f_N(N) = \frac{N}{H_N + N} \tag{1.12}$$

Con queste scelte è possibile fissare il tasso massimo di crescita quando la disponibilità di luce o di nutrienti è molto grande, ovvero quando $I \to \infty$ o $N \to \infty$. Nel caso in cui entrambe le condizioni siano verificate, il tasso di crescita è massimo ed è μ_0.

1.3 La formulazione matematica di BFM

BFM incorpora tutte le complicazioni presentate nella sezione precedente: finitezza delle risorse, interazione tra specie, eterogeneità spaziale... La formulazione più generale del problema per l'ecosistema marino [11] è la seguente:

$$\frac{\partial c_i}{\partial t} = A_{\text{phys}}(c_i) + D_{\text{phys}}(c_i) + R_{\text{bio}}(c_i, c_1, \ldots, c_N, T, I) \tag{1.13}$$

dove c_i è una concentrazione generica e A_{phys} e D_{phys} sono le componenti del termine lineare relativo al trasporto, rispettivamente avvezione e diffusione. Il termine non lineare R_{bio} è invece responsabile della biologia, e quindi dei cicli presentati nella Sezione 1.1, e

dipende in generale dalle altre concentrazioni e da fattori ambientali come la temperatura T, l'intensità di luce I... A seconda delle necessità, il termine di trasporto può essere incluso con diverse dimensionalià: 0D, 1D, 3D.

Le applicazioni 0D trascurano l'eterogeneità spaziale, e quindi presuppongono una distribuzione omogenea delle concentrazioni. I termini di trasporto e avvezione sono assenti:

$$\frac{\partial c_i}{\partial t} = R_{\text{bio}}(c_i, c_1, \ldots, c_N, T, I) \tag{1.14}$$

La configurazione di BFM per il modello 0D è denominata *BFM STANDALONE* [25] ed è quella su cui è stato eseguito in gran parte il lavoro di profiling e ottimizzazione esposto in questa tesi

I modelli 1D simulano una colonna verticale, le componenti orizzontali sono omogenee. Essi sono un'estensione della (1.10):

$$\frac{\partial c_i}{\partial t} = -v\frac{\partial c_i}{\partial z} + \frac{\partial}{\partial z}D\frac{\partial c_i}{\partial z} + R_{\text{bio}}(c_i, c_1, \ldots, c_N, T, I) \tag{1.15}$$

Infine i modelli 3D contengono una formulazione completa del termine di trasporto:

$$\begin{aligned}\frac{\partial c_i}{\partial t} = &-U \cdot \nabla c_i + k_h(-1)^{\frac{n}{2}+1}\nabla_h^n c_i + \\ &-v\frac{\partial c_i}{\partial z} + \frac{\partial}{\partial z}D\frac{\partial c_i}{\partial z} + R_{\text{bio}}(c_i, c_1, \ldots, c_N, T, I)\end{aligned} \tag{1.16}$$

dove $-U \cdot \nabla c_i$ è il termine di avvezione e $k_h(-1)^{\frac{n}{2}+1}\nabla_n^n c_i$ e il termine di diffusione orizzontale. I modelli 3D hanno la possibilità di rappresentare le dinamiche laterali e simulare condizioni variabili che si verificano solitamente in diverse regioni geografiche. Come riportato nella Sezione 1.1, l'accoppiamento di BFM con modelli per l'idrodinamica permette la simulazione su un dominio spaziale 1D e 3D. Il termine R_{bio} dipende dall'ecosistema considerato: in Figura 1.4 è riportata lo schema della rete trofica simulata da BFM per il dominio pelagico.

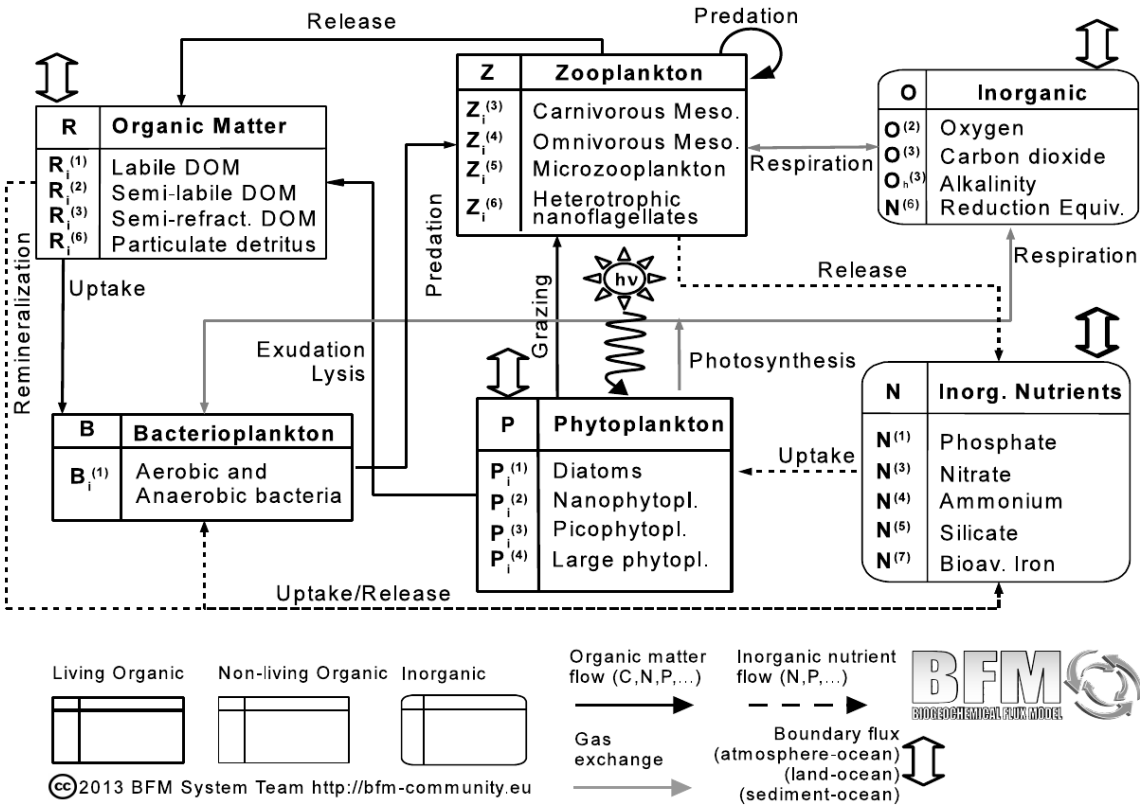


Figura 1.4. Schema della rete trofica per il dominio pelagico in BFM. (M. Vichi et al. *The Biogeochemical Flux Model (BFM): Equation Description and User Manual. BFM version 5.1.* BFM Report Series 1. Ver. 1.1. Bologna, Italy, ago. 2015. URL: http://www.bfm-community.eu)

Capitolo 2

Elementi di HPC

Per introdurre profiling e ottimizzazione del codice, è necessario spiegare alcuni concetti di High Performance Computing (HPC) che sono alla base del lavoro svolto in tirocinio.

In questo capitolo è contenuta una breve introduzione alla struttura di un cluster HPC e sono spiegate le modalità con cui è possibile sfruttarlo per simulare un dominio spaziale, come nel caso di BFM quando accoppiato con un modello per l'idrodinamica: in particolare, è presentata la tecnica della *domain decomposition*, grazie alla quale un calcolo su un dominio spaziale può essere suddiviso tra i vari nodi di un cluster.

Sono poi presentate le basi teoriche su cui è stato svolto il lavoro di profiling e ottimizzazione. Sono esposti i tre livelli a cui normalmente può presentarsi un bottleneck nell'esecuzione di un programma: sistema, applicazione, microarchitettura. Viene approfondito il livello microarchitetturale, ovvero quello che in gran parte interessa le prestazioni della *pipeline*, una delle funzionalità più importanti dei processori moderni.

Infine, sono passate in rassegna alcune ottimizzazioni apportate dai compilatori per massimizzare le prestazioni di un software: esse permettono di generare file binari che sfruttino le più recenti innovazioni introdotte nelle architetture dei microprocessori. In un lavoro di ottimizzazione al livello microarchitetturale, tali ottimizzazioni sono da valutare e calibrare attentamente una per una: questo non solo perché attivarne alcune può comportare un sensibile aumento delle prestazioni, ma anche perché attivare quelle inadeguate a certi codici può addirittura renderle controproducenti.

2.1 Struttura di un cluster HPC

Il calcolo parallelo è l'esecuzione simultanea di un programma, diviso o specificamente adattato, su più microprocessori o più core dello stesso processore, allo scopo di aumentare le prestazioni di calcolo del sistema di elaborazione [27].

La necessità del calcolo parallelo nasce dal fatto che, per elaborazioni costose, non è

sufficiente utilizzare un software seriale su un hardware potente: aumentare indefinitamente la frequenza di clock[1] di un processore richiede misure per la dissipazione del calore sempre più intensive; aumentare la capacità di una singola unità di memoria la rende più lenta. Si ha, sostanzialmente, una violazione della *legge di Moore* [28], secondo cui la "la complessità di un microcircuito, misurata ad esempio tramite il numero di transistori per chip, raddoppia ogni 18 mesi". A causa di limitazioni fisiche, il progresso non è oggi così semplice come ai tempi dell'enunciazione di questa legge, nel 1965, quando stava prendendo il via la corsa all'evoluzione dei processori.

Si può ovviare a queste criticità con l'impiego di un *cluster HPC*. Un cluster HPC è suddiviso tra i cosiddetti *nodi*: all'interno di ciascun nodo, vi é un processore con un'unità di memoria ed eventualmente un acceleratore (ad esempio una GPU). Quando si lancia un programma sul cluster, ciascun nodo ne esegue una copia, detta *processo*; a seconda del nodo sul quale si trova in esecuzione, un determinato processo esegue compiti diversi rispetto agli altri, stabiliti dal programmatore nelle fasi di progettazione e stesura del codice. Il sistema formato dai nodi, ciascuno avente una propria area di memoria a cui gli altri non possono accedere, è detto *sistema a memoria distribuita*, e necessita di un protocollo di comunicazione tra i processi. A questo proposito, esiste lo standard MPI, disponibile in diverse implementazioni, tra cui Open MPI[2] e Intel MPI[3].

All'interno di ciascun nodo del cluster HPC, vi è un ulteriore grado di parallelismo, poiché i processori o gli acceleratori possono avere più unità di calcolo, detti *core*. Il lavoro del software può essere suddiviso tra i singoli core creando dei sottoprocessi, chiamati *thread*. Ciascun nodo del cluster, nel quale tutti i core sono in grado di accedere alla stessa area di memoria, costituisce un *sistema a memoria condivisa*. In questo caso, non è necessario un protocollo di comunicazione tra i thread, ma serve piuttosto una API (Application User Interface, ossia un'interfaccia ad alto livello con l'hardware) che consenta di suddividere in thread l'esecuzione di un software. La API più diffusa è OpenMP[4]. In Figura 2.1 è schematizzata la struttura di un cluster.

Un fattore determinante nella scelta di come suddividere un'esecuzione, in processi o in thread, è il numero di essi. Un sistema a memoria condivisa è limitato dal numero di core che può contenere: l'accesso simultaneo alla stessa unità di memoria da parte di molti core è fonte di rallentamenti. Quando una esecuzione deve essere divisa in molte parti simultanee, come ad esempio in una domain decomposition che sarà approfondita nella sezione successiva, i vincoli dell'hardware impongono di utilizzare un sistema a memoria distribuita. Un sistema a memoria condivisa è invece più adatto a suddividere un'esecuzione in un numero limitato di thread; rispetto al sistema a memoria distribuita,

[1] Il clock è un dispositivo che scandisce le attivazioni dei microcircuiti logici che compongono un processore. È necessario che essi lavorino in modo sincronizzato, aspettando il completamento delle operazioni dei microcircuiti più lenti. Il compito del clock è realizzare questo sincronismo: ad ogni ciclo di clock, ciascun microcircuito deve portare a completamento un'operazione e scambiare informazioni con gli altri.

[2] https://www.open-mpi.org/

[3] https://software.intel.com/en-us/intel-mpi-library

[4] http://openmp.org/wp/

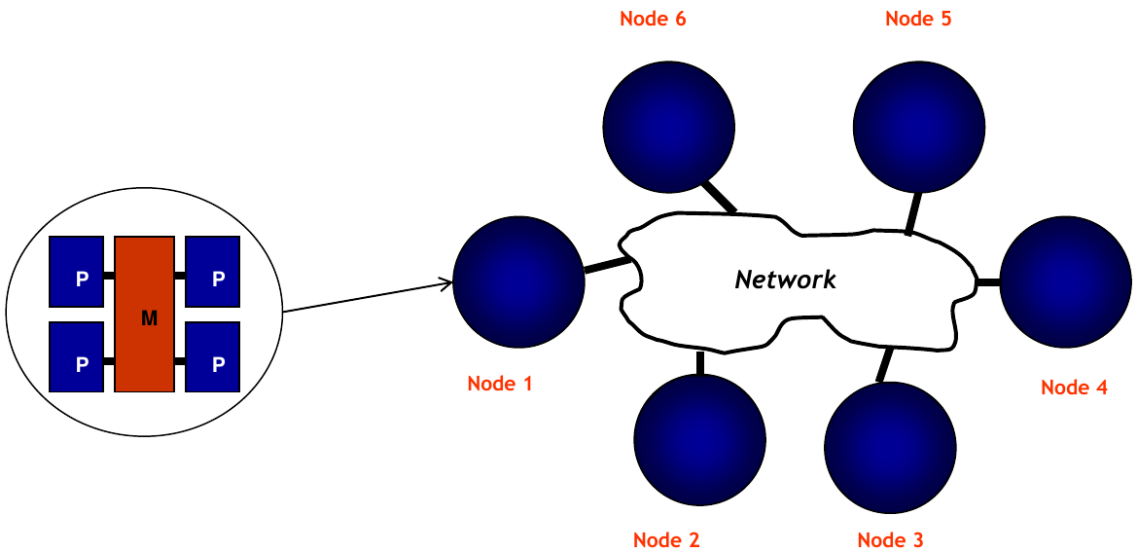


Figura 2.1. Schema di un cluster HPC (A. Emerson e G. Erbacci. *Introduction to HPC Architectures*. 25th Summer School of Parallel Computing of CINECA. 2016)

ha il vantaggio di non introdurre overhead dovuti alla comunicazione tra diversi processi (anche se ne vengono introdotti altri quando il programma in esecuzione avvia o termina uno o più thread). Il sistema a memoria condivisa dato dai diversi core di un processore può essere utilizzato per parallelizzare ulteriormente il processo che tale processore ha in carico.

2.2 Domain Decomposition e sua applicazione in OGSTM-BFM

Un cluster HPC si può utilizzare per suddividere tra diversi processi la risoluzione di equazioni differenziali alle derivate parziali definite su un dominio spaziale e dipendenti dal tempo.

Per introdurre questo argomento è possibile utilizzare l'esempio dell'equazione semplice del calore, in assenza di forzanti e con condizioni al contorno fissate [1]. Si consideri un campo $U(x, y, t)$, dipendente dalle coordinate spaziali x e y e dalla coordinata temporale t. Una volta discretizzato il dominio, il campo $U(x, y, t)$ diventa, ad un dato istante di tempo t, una matrice a due indici $U^t_{x,y}$. L'algoritmo per il timestepping aggiorna la matrice coi nuovi valori ad ogni timestep ed ha la forma seguente:

$$U^{t+\delta t}_{x,y} = U^t_{x,y} + C_x(U^t_{x+1,y} + U^t_{x-1,y} - 2U^t_{x,y}) + C_y(U^t_{x,y+1} + U^t_{x,y-1} - 2U^t_{x,y})$$

dove C_x è la conducibilità termica lungo x e C_y la conducibilità termica lungo y. Ciascun elemento di matrice al tempo successivo dipende da se stesso e dagli elementi adiacenti. La località dei dati necessari ad aggiornare ogni elemento di matrice permette di scomporre la matrice $U^t_{x,y}$ in un numero di sottomatrici, il cui aggiornamento ad ogni timestep è

gestito da diversi processi quasi completamente indipendenti. Un esempio di scomposizione verticale, è mostrato in Figura 2.2. Le uniche informazioni che un processo deve scambiare

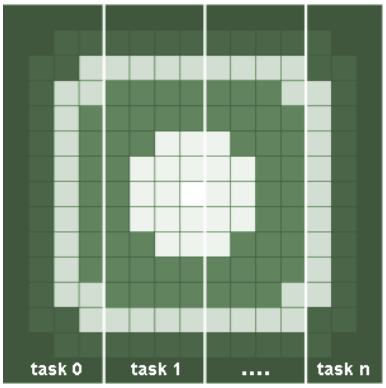


Figura 2.2. Distribuzione di "fette" verticali di una matrice a diversi processi (B. Barney. *Introduction to Parallel Computing*. 2016. URL: https://computing.llnl.gov/tutorials/parallel_comp/)

con altri, se il sistema è a memoria distribuita, sono la prima e ultima colonna della sua sottomatrice al termine di un timestep. Se ogni processo ha a carico una sottomatrice $m \times n$, l'N-esimo processo deve ricevere dall'$N+1$-esimo la $m+1$-esima colonna (che è la prima per il processo $N+1$-esimo) e dall'$N-1$-esimo la colonna 0 (che è la m-esima per il processo $N-1$-esimo), in modo da poter eseguire il timestep successivo. Il codice risulta così parallelizzato con questa tecnica, che prende il nome di *domain decomposition*.

Questo semplice ma significativo esempio permette di comprendere come l'HPC sia fondamentale per BFM. Le simulazioni su domini 3D con BFM vengono effettuate tramite l'accoppiamento con modelli per l'idrodinamica, i quali sono progettati per essere paralleli attraverso la domain decomposition. Il modello OGSTM-BFM sfrutta questa tecnica per poter effettuare simulazioni su domini estesi: il dominio (mare, oceano, ecc.) viene scomposto orizzontalmente secondo una griglia bidimensionale in tanti blocchi, e l'evoluzione di ciascuno viene gestita da un diverso nodo. OGSTM-BFM impiega lo standard MPI per la comunicazione tra processi. È bene rimarcare che il lavoro di ottimizzazione esposto nel Capitolo 3, pur avendo avuto ripercussioni sul codice parallelo OGSTM-BFM, si è concentrato sul solo codice seriale di BFM.

2.3 Modalità di ottimizzazione

Definita la struttura di un cluster, è possibile individuare tre livelli a cui operare per ottimizzare un software [18]:

- Livello di *sistema*: riguarda l'hardware del computer e il software di sistema, includendo quindi disco rigido, interfacce di rete, memoria, BIOS, sistema operativo, sistema di raffreddamento, processore, ecc. Questi componenti, affinché offrano le prestazioni attese, devono essere adeguatamente preparati per sostenere il carico di lavoro dell'applicazione.

- Livello di *applicazione*: riguarda l'uso di una corretta implementazione algoritmica, le API impiegate, utilizzo della memoria, ecc. Include anche una parallelizzazione ottimale, ovvero una buona gestione delle comunicazioni in un sistema a memoria distribuita, una ragionevole scomposizione dell'esecuzione in thread in un sistema a memoria condivisa, ecc.

- Livello *microarchitetturale*: riguarda l'uso efficiente delle risorse interne al processore da parte dell'applicazione, come la pipeline, la vettorizzazione e la branch prediction.

In Figura 2.3 è schematizzata l'efficacia dell'ottimizzazione in funzione del livello a cui viene apportata.

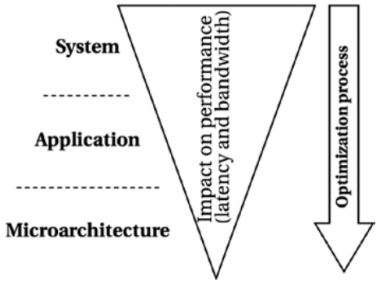


Figura 2.3. Efficacia dell'ottimizzazione in funzione del livello (G. Supalov et al. *Optimizing HPC Applications with Intel Cluster Tools.* 2014)

Come si avrà modo di vedere nel Capitolo 3, il lavoro di ottimizzazione di questa tesi è stato concentrato sul livello microarchitetturale. È assunto che agli altri livelli ci si trovi già in condizioni ottimali.

2.4 La pipeline

I concetti di base per capire il livello microarchitetturale, oggetto dell'ottimizzazione di BFM, sono esposti in questa e nelle prossime sezioni. Per un approfondimento a riguardo, si rimanda a G. Supalov et al. *Optimizing HPC Applications with Intel Cluster Tools.* 2014, da cui sono tratte in gran parte questa e le sezioni che seguono. In questa sezione si inizierà introducendo la *pipeline*, funzionalità dei processori odierni da cui le prestazioni di un qualsiasi software dipendono in modo critico.

La pipeline si può considerare come la versione informatica di una catena di montaggio industriale ed è il principio fondante del design di un core di un processore moderno. Essa introduce una parallelizzazione nell'esecuzione di un flusso di istruzioni da parte di un core. Ciascuno stadio della pipeline esegue un compito specializzato in parallelo agli altri. Un classico modello di pipeline è il seguente:

- *Instruction Fetch* (IF): carica l'istruzione indicata dall'*instruction pointer*, un registro[5] del processore che contiene l'indirizzo di memoria dell'istruzione che deve essere eseguita in un certo momento.
- *Instruction Decode* (ID): l'istruzione viene analizzata e le viene associata la funzionalità da eseguire.
- *Load Operands* (LO): vengono caricati i dati su cui l'istruzione deve essere eseguita.
- *Execution* (E): l'istruzione viene eseguita. Se si tratta di un calcolo aritmetico, viene eseguito il calcolo stesso.
- *Write-Back results* (WB): il risultato è inviato alla destinazione designata, generalmente un registro.

In un approccio privo di pipeline, ciascuno di questi passi deve essere completato prima che l'istruzione successiva sia caricata. Mentre uno di questi stadi è attivo, tutti gli altri sono in attesa, così che le risorse del processore non siano completamente sfruttate. In un approccio con pipeline, invece, ciascuno stadio della pipeline è attivo allo stesso tempo, portando all'avanzamento di un'istruzione lungo il suo iter per ogni ciclo di clock (assumendo che il completamento di uno stadio richieda un ciclo di clock). Se si assume che non ci sia dipendenza tra le istruzioni individuali, si può caricare la terza istruzione (IF) mentre la seconda è in decodifica (ID); allo stesso tempo si possono caricare i dati su cui deve operare la prima (LO), e così via. In questo modo, nessuno stadio della pipeline è mai in attesa, come mostrato in Figura 2.4.

Nei core dei processori moderni la parallelizzazione non avviene solo grazie alla pipeline ma anche in fase di esecuzione: nella microarchitettura Sandy Bridge di Intel, infatti, con un ciclo di clock è possibile eseguire fino a 4 istruzioni decodificate (core super-scalare). Non solo, più istruzioni possono essere eseguite in un ordine diverso rispetto a quello in cui sono caricate nella fase di fetching. Ciò è particolarmente conveniente quando lo stallo di una istruzione (perché magari è in attesa di un dato dalla memoria) impedisce l'esecuzione di altre, che magari non dipendono dal completamento della stessa. Del loro ordine originale tiene traccia uno stadio aggiuntivo chiamato *reservation station*, il quale controlla anche che l'esecuzione "fuori ordine" sia possibile e procede all'invio allo stadio di esecuzione. Successivamente lo stadio del *retirement* (ritiro) riordina le istruzioni dopo l'esecuzione e manda ai registri i risultati nell'ordine originale.

Il CPI (Cycles Per Instruction) rate è un parametro che valuta l'efficienza con cui viene sfruttata la parallelizzazione nello stadio di esecuzione: esso è il rapporto tra il numero di cicli di clock e il numero di istruzioni eseguite in un certo tempo (più precisamente, viene contato il numero di istruzioni che subiscono retiring). Come riportato sopra, ad oggi i core Intel possono eseguire fino a 4 istruzioni per ciclo di clock, quindi il minimo CPI rate è 0.25. È auspicabile che questo parametro sia sempre il più basso possibile (tra 0.25

[5] Un'area di memoria piccola e veloce. Nei registri del processore sono conservati anche i dati su cui devono operare le istruzioni, e i loro risultati.

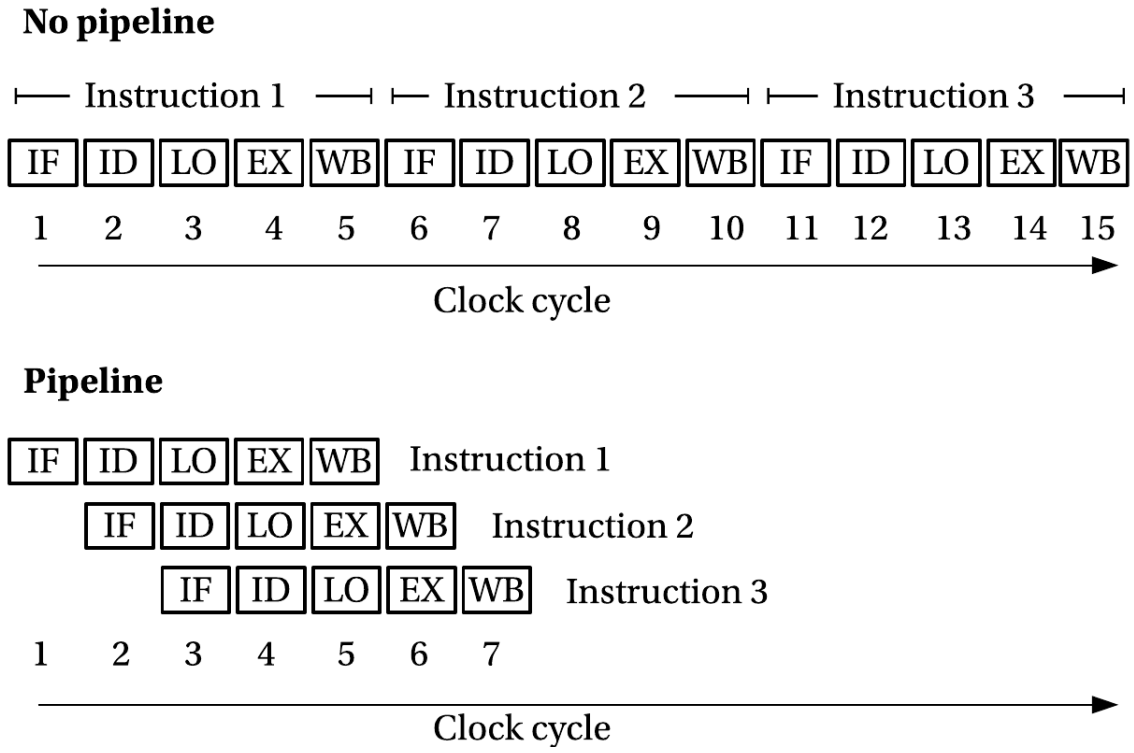


Figura 2.4. Confronto tra esecuzione con pipeline ed esecuzione senza pipeline (G. Supalov et al. *Optimizing HPC Applications with Intel Cluster Tools*. 2014)

e 0.7 è considerato accettabile), ma nella realtà intervengono diversi fattori a causare lo stallo della pipeline e quindi ad aumentare il CPI rate. Pur essendo un parametro basato sul solo stadio di esecuzione, il CPI rate è utile a dare una valutazione sommaria delle performance della pipeline. Tra i fattori che possono ostacolare il procedere della pipeline si annoverano:

- *Conflitti di dati*: si tratta dell'impossibilità di eseguire contemporaneamente istruzioni che devono accedere e scrivere su una stessa area di memoria.
- *Conflitti di controllo*: le istruzioni di controllo (`if`, controllo dei contatori dei cicli, ecc.) creano delle diramazioni nel codice e, non essendo certa la diramazione che sarà presa prima che il controllo sia eseguito, non è possibile caricare e decodificare istruzioni in anticipo mentre le ultime istruzioni prima della diramazione vengono eseguite. Per ovviare al problema, i core sono dotati di un'unità denominata *branch predictor*, che cerca di prevedere l'esito di un'istruzione di controllo in base ai risultati precedenti dello stesso controllo. Nella pipeline vengono quindi caricate le istruzioni della diramazione prevista. Se la previsione è sbagliata, le istruzioni vengono cancellate e la pipeline deve essere riempita da capo.
- *Conflitti strutturali*: emergono quando sono richieste all'hardware più funzionalità di quelle che può offrire. Ad esempio, se sono disponibili quattro registri di memoria

e sono eseguite in parallelo due istruzioni che ne richiedono due ciascuna, non può essere eseguita in parallelo anche una terza istruzione che necessiti di registri di memoria.

- *Accesso a memoria non locale*: l'accesso alla memoria principale è fonte di rallentamenti per il processore. Il progresso in termini di velocità delle CPU non è stato eguagliato da quello delle unità di memoria. Pertanto, le CPU sono state dotate di una memoria interna, piccola ma veloce, detta *cache*. L'idea alla base della cache è il principio di località: i dati usati una volta saranno probabilmente riutilizzati nel futuro prossimo. La cache, quindi, contiene copie intermedie dei dati che si trovano nella memoria principale. Avendo una capacità piuttosto limitata, la cache funziona bene se vengono frequentemente utilizzati gli stessi dati: se il programma lavora su grossi dataset, la cache dovrà frequentemente essere svuotata e aggiornata accedendo alla memoria principale, con conseguenti fasi di stallo della pipeline.

2.5 L'architettura Sandy Bridge

In Figura 2.5 è riportato lo schema della microarchitettura Sandy Bridge. Essa è del tutto analoga ad Ivy Bridge, l'architettura su cui sono basati i core del cluster PICO utilizzato da OGS per l'esecuzione di BFM. Ivy Bridge si distingue da Sandy Bridge solo per un maggiore grado di miniaturizzazione.

Lo schema mostra il percorso di un'istruzione, dal fetching al retirement. Le istruzioni conservate nella memoria principale entrano attraverso la cache e vengono decodificate in microistruzioni, per poi essere conservate in una cache apposita (Uop Cache). Lo scheduler distribuisce fino a 6 istruzioni della Uop Cache alle porte di esecuzione, a seconda delle funzionalità che richiedono. Esse sono inviate "fuori ordine" e un buffer per il riordino tiene traccia della sequenza originale. Al completamento di un'istruzione, i risultati dell'esecuzione sono inviati nel corretto ordine agli appositi registri, e le istruzioni vengono ritirate. Con un ciclo di clock può avvenire il retiring di al massimo 4 istruzioni.

Nella prossima sezione saranno ripresi più in dettaglio i singoli passi appena descritti, e suddivisi in due fasi principali utili a classificare le performance della pipeline.

2.6 Classificazione delle performance della pipeline

Per identificare i problemi dovuti all'utilizzo della pipeline da parte del processore, in questa sezione esso sarà suddiviso logicamente in base alla sequenza temporale che percorre un'istruzione al suo interno. Facendo riferimento alla Figura 2.5, sono definite due fasi principali in cui sono suddivisi passi discussi nella Sezione 2.4, di seguito riassunti:

- *Front end*: è la fase nella quale l'istruzione è preparata per l'esecuzione. Essa viene caricata, pre-decodificata, inserita all'interno di una coda (Instruction Queue), decodificata, tramutata in micro-operazioni e conservata in una cache (Uop Cache) in attesa di essere eseguita. In questa fase agisce anche il branch predictor, scegliendo quali istruzioni caricare nella pipeline in presenza di una diramazione.

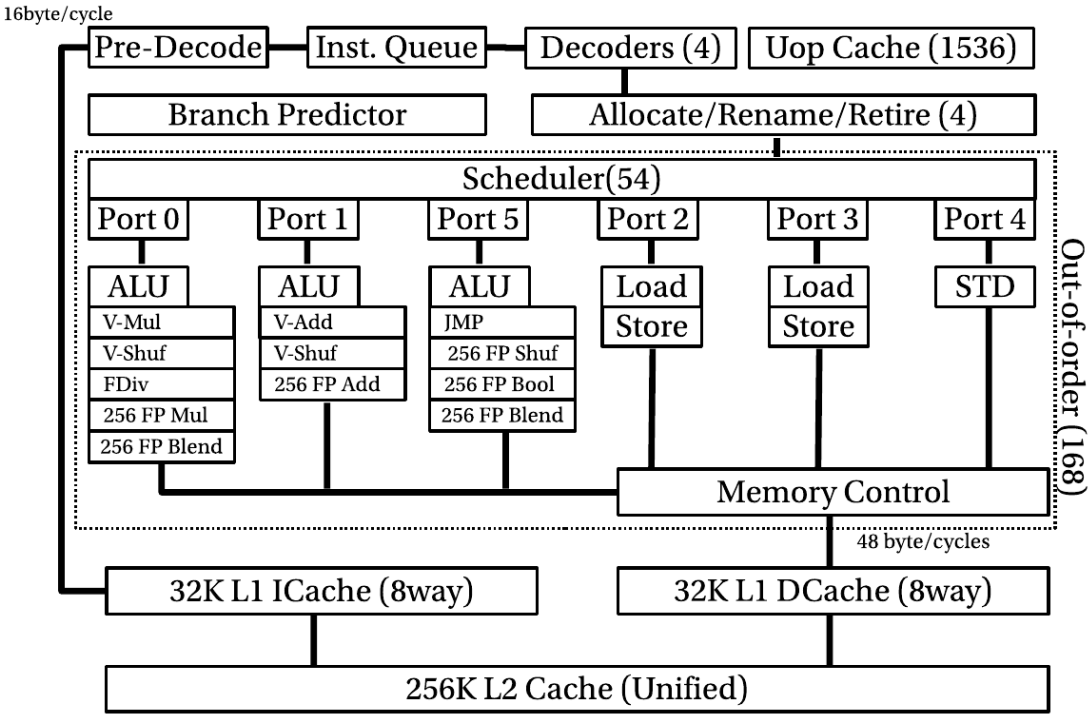


Figura 2.5. Schema del core Sandy Bridge (G. Supalov et al. *Optimizing HPC Applications with Intel Cluster Tools*. 2014)

- *Back end*: la seconda fase consiste nell'effettivo stadio di esecuzione. Le istruzioni decodificate presenti nella Uop Cache sono valutate per l'esecuzione out-of-order ed "etichettate" per l'eventuale riordino. Successivamente, se ciò è stato valutato possibile, sono eseguite secondo l'ordine che massimizza le prestazioni. Lo stadio del retiring si occupa del riordino delle istruzioni eseguite e permette di procedere alla scrittura dei risultati in memoria.

Date le due fasi spiegate sopra, è importante riuscire a identificare il bottleneck del software all'interno di una delle due, capendone anche l'origine. Si possono individuare le cause di scarse prestazioni della pipeline sulla base delle semplici deduzioni schematizzate in Figura 2.6:

- Il primo passo è controllare se una micro-operazione può essere mandata in esecuzione. Se ciò non avviene, si possono avere due casi:
  - Se le risorse del back-end (registri, porte) sono libere, ovvero non vi è uno stallo nella loro allocazione, ma le micro-operazioni non vengono ad esse passate, il codice si dice *front end bound*. Il front end non riesce a consegnare le istruzioni e il back end non può procedere all'esecuzione, risultando così sotto-utilizzato.
  - Se vi è invece uno stallo nell'allocazione delle risorse, perché tutte le unità di esecuzione risultano impegnate, il codice si dice *back end bound*. Quest'ultimo

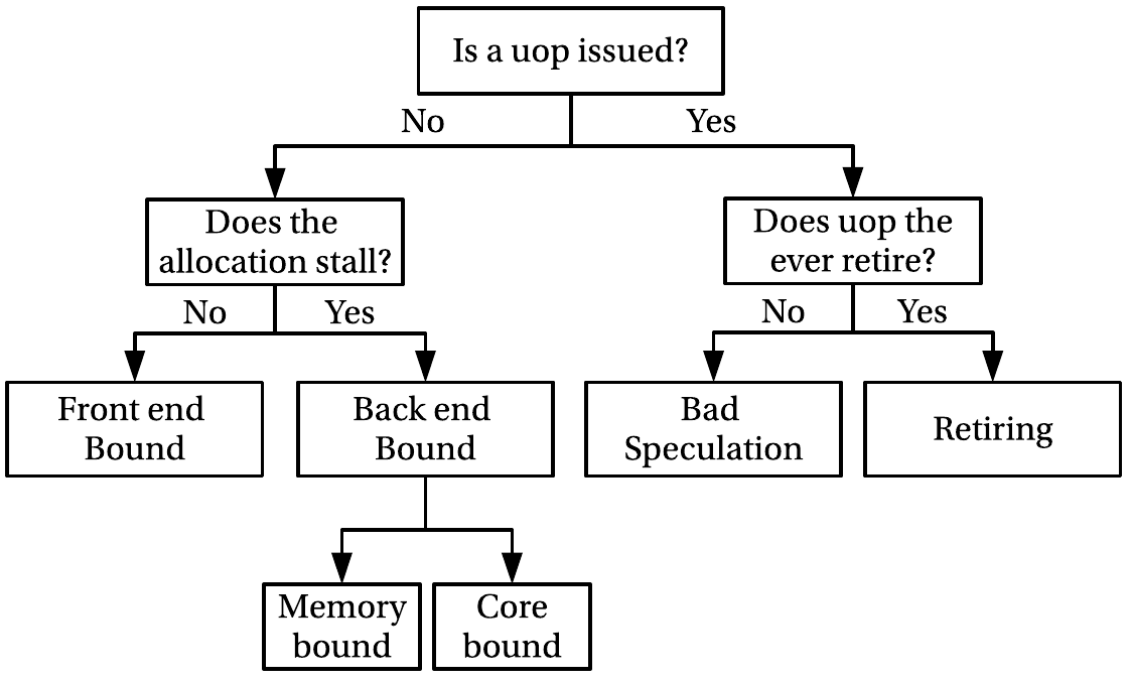


Figura 2.6. Individuazione delle criticità della pipeline (G. Supalov et al. *Optimizing HPC Applications with Intel Cluster Tools*. 2014)

caso può ulteriormente essere suddiviso nei sottocasi di codice *core bound*, in cui a impegnare le risorse è il calcolo, e codice *memory bound*, in cui a impegnare le risorse sono gli accessi alla memoria.

- Se una micro-operazione è mandata in esecuzione, rimane da controllare che subisca il retirement. Se ciò non avviene, significa che essa è stata rimossa dalla pipeline nonostante sia stata caricata, necessariamente a causa di una predizione errata del branch predictor. Questa evenienza è definita come *bad speculation*.

Per ciascuno dei problemi appena individuati possono essere individuate alcune cause comunemente riscontrabili:

- *Front-end bound*: è dovuto al mancato reperimento di un'istruzione nella ICache (Instruction Cache), a causa di un codice grosso o eccessivi inlining e loop unrolling (inlining e loop unrolling sono introdotti nella prossima sezione). Può anche essere dovuto ad inefficienze nella decodifica.

- *Back-end memory bound*: nella cache non sono presenti i dati su cui devono operare le istruzioni, a causa di accessi irregolari alla memoria, grossi dataset, conflitti nella cache.

- *Back-end core bound*: è dovuto a istruzioni a lunga latenza (divisioni), sequenze di istruzioni dipendenti le une dalle altre, codice non vettorizzato (anche la vettorizzazione è introdotta nella prossima sezione).

- *Bad speculation*: le diramazioni non vengono predette correttamente, a causa della presenza di molte istruzioni di controllo che danno diversi risultati ogni volta che vengono incontrate.

2.7 Ottimizzazioni apportate dal compilatore

Quando un programma viene compilato, il compilatore non si limita a leggere il codice e tradurlo in binario. Se infatti il codice è scritto dal programmatore in modo da essere intelligibile per l'umano, non è detto che la sua stesura risulti altrettanto ottimale per il core quando deve eseguire le istruzioni codificate. Si pensi, ad esempio, al solo fatto che uno spezzone di codice che al programmatore appare lineare in fase di esecuzione può essere eseguito out-of-order, come visto nelle sezioni precedenti.

Nella compilazione sono dunque previste diverse fasi: il codice viene letto, disassemblato e riassemblato per ottimizzare l'esecuzione. Vi sono diverse tecniche che possono essere attuate in scrittura del codice per rendere più semplice al compilatore la comprensione e l'ottimizzazione del software. Addirittura il compilatore è spesso in grado di attuare automaticamente queste tecniche, specificando opportune flag di compilazione e senza dover inserire esplicitamente direttive all'interno del codice. Alcune di queste tecniche sono:

- *Loop unrolling*: in un ciclo vengono condensate in una sola iterazione le operazioni di più iterazioni. Si passa quindi da un codice del tipo:

```
for k=1:length(b)
  b(k)=a(k)+k
end
```

  ad un codice con la seguente forma:

```
for k=1:length(b):3
  b(k)=a(k)+k
  b(k+1)=a(k+1)+k+1
  b(k+2)=a(k+2)+k+2
end
```

  Il loop unrolling permette di ridurre il numero di controlli della variabile contatore al termine di ogni iterazione.

- *Inlining*: alla chiamata di una funzione o di una subroutine all'interno del codice viene sostituito il corpo della stessa. Similmente all'unrolling, riduce il numero di diramazioni, poiché alle chiamate di funzioni e subroutine sono associate istruzioni di controllo.

- *Vettorizzazione*: i moderni processori dispongono di un set di istruzioni di tipo SIMD (Single Instruction Multiple Data), che permettono di operare con una sola istruzione

su più dati. È il caso delle operazioni tra vettori: se `a`, `b` sono vettori, il core riesce con una sola istruzione (limitatamente all'estensione dei registri che vengono usati per queste operazioni) ad eseguire l'istruzione `a+b`. Il codice si dice vettorizzato, e l'operazione di vettorizzazione avviene in fase di compilazione: il compilatore può qui stabilire se un'istruzione contenuta nel codice può essere tradotta in un'istruzione di tipo SIMD. Un modo per suggerire al compilatore di compiere la vettorizzazione, se possibile, è scrivere le operazioni tra vettori senza fare diretto riferimento ai loro elementi, quindi usare istruzioni del tipo:

```
c = a + b
```

invece di:

```
for k=1:length(a)
  c(k)=a(k)+b(k)
end
```

Capitolo 3

Profiling e ottimizzazione di BFM

In questo capitolo sono descritti i passi del lavoro di profiling e ottimizzazione di BFM STANDALONE. Sono riportati i risultati più importanti ed è valutato l'impatto sul codice parallelo OGSTM-BFM delle modifiche apportate al codice seriale. Alla fine del capitolo sono riportate delle tabelle riassuntive con i risultati ottenuti su BFM STANDALONE e OGSTM-BFM.

3.1 Informazioni preliminari

3.1.1 Hardware e software

I lanci di BFM STANDALONE sono stati eseguiti su un nodo del cluster HPC PICO del CINECA, il principale centro italiano per il supercalcolo. PICO è un GNU/Linux infiniband cluster composto da 74 nodi, usati in gran parte per applicazioni HPC, data analytics e sviluppo del software. Il nodo utilizzato ha 2 processori da 10 core Intel Xeon (E5-2670 v2) con una frequenza di clock di 2.5 GHz, in grado di eseguire 8 operazioni in virgola mobile per ciclo. I nodi sono connessi da una rete Mellanox Infiniband FDR da 56 Gb/s, che permette un'interconnessione a bassa latenza ed alta larghezza di banda.

Il sistema operativo installato, su cui sono stati eseguiti i lanci di BFM, è la distribuzione GNU/Linux CentOS 6.5. La suite presente per la compilazione di codici C/C++ e Fortran è Intel Parallel Studio XE 2016 (versione 16.0.0), di cui è stato utilizzato il compilatore Fortran `ifort`.

La Figura 3.1.1 riporta le caratteristiche dell'hardware, disponibili all'indirizzo `http://www.hpc.cineca.it/hardware/pico`) .

3.1.2 Benchmark

Il testcase standard di BFM STANDALONE è denominato `STANDALONE_PELAGIC` ed è il preset predefinito quando il codice ed è generato e compilato senza alcuna opzione [25]. Esso simula il ciclo stagionale di un'area costiera temperata, con profondità media 5 m. La simulazione dura 10 anni: la data di inizio è gennaio 2000 e quella di fine è impostata a gennaio 2010 con un time step massimo di 8460 s per il metodo adattativo

	Total Nodes	CPU	Cores per Nodes	Memory (RAM)	Notes
Compute/login node	66	Intel Xeon E5 2670 v2 @2.5Ghz	20	128 GB	
Visualization node	2	Intel Xeon E5 2670 v2 @ 2.5Ghz	20	128 GB	2 GPU Nvidia K40
Big Mem node	2	Intel Xeon E5 2650 v2 @ 2.6 Ghz	16	512 GB	1 GPU Nvidia K20
BigInsight node	4	Intel Xeon E5 2650 v2 @ 2.6 Ghz	16	64 GB	32TB of local disk

Figura 3.1. Caratteristiche dell'hardware di PICO

Runge-Kutta, e l'output è registrato ogni 30 giorni. L'arco temporale della simulazione si imposta dall'apposito namelist `Standalone.nml`, situato nella cartella di output della compilazione:

```
38 &time_nml
39     timefmt = 2
40     MaxN    = 144
41     simdays = 5760
42     start   = '2000-01-01 00:00:00'
43     stop    = '2100-01-01 00:00:00'
44 /
```

Per il codice parallelo OGSTM-BFM, `STANDALONE_PELAGIC` ha un analogo, dal punto di vista della rete trofica, denominato `OGS_PELAGIC`. La simulazione è su un bacino a forma di parallelepipedo rettangolo, risolto:

- orizzontalmente con 10×10 celle;
- verticalmente con 43 celle.

Le celle costituiscono tutte punti d'acqua, eccetto quelle al bordo del dominio che costituiscono punti di terra; in z ovviamente solo la parte di bordo associata al fondale costituisce punti di terra, l'altra sarà la superficie dell'acqua. L'arco temporale coperto dalla simulazione è fissato a 1 giorno nell'apposito namelist `Start_End_Times` collocato nella cartella di output della compilazione:

```
1 20000101-00:00:00
2 20000102-00:00:00
```

3.1.3 Metodo di lavoro

Il profiling nelle varie fasi del lavoro sono stati eseguiti utilizzando il profiler Intel VTune Amplifier XE 2016, un programma per l'analisi delle performance di software.

VTune permette differenti tipi di analisi di performance del codice in esame, a seconda delle problematiche che si vogliono analizzare. Ad esempio, esiste una serie di analisi (Basic Hotspots, Advanced Hotspots, Concurrency, Locks and Waits) che permette di valutare le prestazioni della particolare implementazione algoritmica scelta [7].

Come anticipato nella Sezione 2.3 del precedente capitolo, il lavoro di profiling e ottimizzazione del tirocinio si è concentrato sul livello microarchitetturale. L'analisi utilizzata è la General Exploration Analysis. Quest'analisi permette, grazie alla raccolta dati dei contatori presenti nei core del processore, di calcolare parametri che definiscono il comportamento del software sull'hardware utilizzato: in particolare, essa utilizza un percorso logico come quello descritto nella Sezione 2.6 per determinare se l'applicazione è front-end bound, back-end bound o affetta da bad speculation. Nell'esposizione del lavoro sono stati riportati i risultati elencati alle seguenti voci:

- *Summary*: offre un rapporto riassuntivo del profiling. Viene riportato il CPI rate per dare la valutazione più generale possibile delle prestazioni; inoltre sono riportati i parametri percentuali *Front-End Bound* e `Back-End Bound`, che stimano quale delle due fasi in cui si è suddivisa la pipeline sia critica per l'esecuzione. Per ciascuno dei due casi, come si vedrà nell'esposizione del lavoro, sono poi presentate informazioni ancora più dettagliate, utili a individuare le fonti di rallentamento.

- *Bottom up*: mostra un elenco di function e subroutine ordinate secondo il numero di clockticks (cicli di clock) richiesti, e di ciascuna sono calcolati diversi parametri per valutare le performance (CPI rate, front-end/back-end bound, bad speculation...). Occasionalmente è stata sfruttata la funzionalità di VTune di esaminare riga per riga il codice di function e subroutine elencate in Bottom up, con lo scopo di cercare le istruzioni più costose per il core.

A partire dai risultati ottenuti, sono esposte le proposte di ottimizzazione e ne viene valutato l'impatto con ulteriori profiling. Sono stati presi in considerazione specialmente i parametri che VTune evidenzia in rosso, ovvero quelli che il profiler individua come i più importanti da migliorare.

I *time to solution*[1] misurati con VTune includono l'overhead della presa dati dai contatori dal processore. Si è quindi utilizzato il profiler per la visualizzazione dei parametri legati al software, ma le misure di tempo sono state effettuate utilizzando il timer del sistema operativo, attraverso il comando `time` della shell Unix. I tempi che si ottengono in questo modo sono affetti da fluttuazioni abbastanza significative: dall'esperienza comune si ha che, utilizzando un cluster, si può avere una differenza di circa il 10% in tempo di esecuzione tra due run. I tempi riportati nei risultati sono stati ottenuti mediando i 5 tempi più piccoli misurati dopo un numero abbastanza grande (indicativamente, tra i 20 e i 30) di lanci di BFM STANDALONE.

[1] Si definisce time to solution il tempo di esecuzione necessario al software affinché questo arrivi alla soluzione del problema.

Una nota sull'impostazione dell'esposizione: è approfondito il funzionamento delle sole parti di codice che è servito ottimizzare. Non sono inserite ora spiegazioni delle specifiche parti di programma oggetto di ottimizzazione, che sarebbe difficile contestualizzare preliminarmente senza essere al corrente dei risultati del profiling. Ciò permette anche di presentare gli espedienti che si sono utilizzati per risalire a particolari informazioni contenute codice sorgente (vedere in particolare nella Sezione 3.2.3, dove si è fatto uso del comando `grep` della shell Unix).

Infine, tutto il lavoro è esposto in modo da poter essere riproducibile. Come riportato nella Sezione 1.1, il codice di BFM è scaricabile dal sito `www.bfm-community.eu`. I percorsi dei file indicati nel testo sono da intendere a partire dalla directory principale del pacchetto che si scarica dal sito.

3.2 Profiling e ottimizzazione di BFM STANDALONE

3.2.1 Primo profiling di BFM STANDALONE

Dal primo profiling di BFM STANDALONE è stato ottenuto il seguente time to solution:

$$\text{time to solution}_{\text{base}} = 3.28 \text{ s}$$

I risultati del profiling sono riportati nelle Figure 3.2 e 3.3.

I rapporti ottenuti evidenziano che il programma è *front-end bound*. Come riportato nella Sezione 2.6, il front-end del core prepara le istruzioni affinché siano eseguite nella fase di back-end. Il fatto che il codice sia front-end bound significa che ad ogni ciclo di clock il back-end è sottosfruttato a causa di uno stallo nella fase di front-end. In generale le cause che portano ad un rallentamento dovuto al front-end del core possono essere suddivise in due gruppi: problemi nel reperire le istruzioni e problemi nella decodifica.

Approfondendo più dettagliatamente i risultati riportati in Figura 3.2, si può notare che il rallentamento nel front-end è dato da un eccesso di *instruction cache misses* (come evidenziato dal parametro `ICache Misses`), ovvero il core non ha pronte le istruzioni nel primo livello della instruction cache quando potrebbe farle avanzare allo stadio di decodifica [6]. Il core è pertanto costretto ad accedere ad aree della memoria più lente (come i livelli più alti della cache o la RAM) per reperire le istruzioni da decodificare. Il parametro `Front-end Bandwidth MITE` evidenzia invece inefficienze da parte del Micro Instruction Translation Engine (MITE), un canale di decodifica ereditato dalle architetture precedenti a Sandy Bridge e al quale va ora preferito il Decoded Stream Buffer (DSB) [10] [5]. I due problemi evidenziati sono riconducibili alle cause che rendono un software front-end bound già discusse nel nella Sezione 2.6: il codice è probabilmente di grandi dimensioni e male impostato.

In Figura 3.3 è riportato lo screenshot del Bottom up del profiling. Anche qui si può vedere come le function e subroutine più costose in termini di clockticks siano front-end bound. Esse includono:

- la subroutine `flux_vector`, che si occupa di aggiornare le matrici contenenti i flussi di nutrienti tra una fase e l'altra;

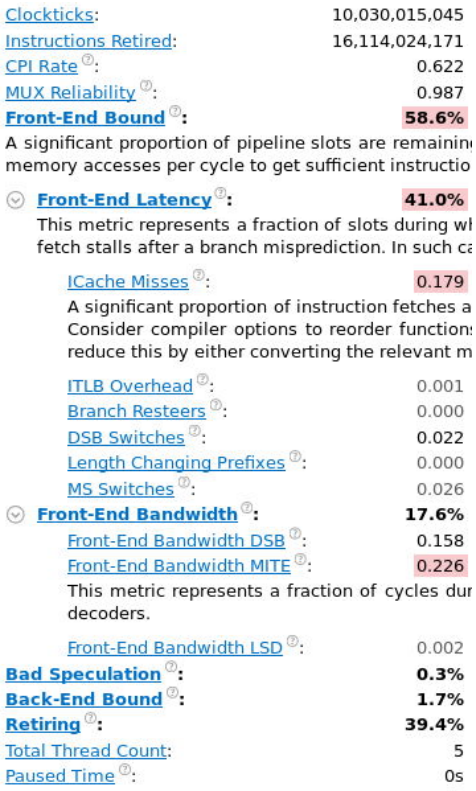


Figura 3.2. Summary del profiling di BFM STANDALONE - versione base

Function / Call Stack	Clockticks	Instructions Retired	CPI Rate	Front-End Bound	Bad Speculation	Back-E... Bound	Retiring
flux_vector	888,001,332	1,998,002,997	0.444	42.7%	0.0%	9.0%	48.9%
flux_vector	820,001,230	1,528,002,292	0.537	45.6%	4.0%	0.0%	58.4%
mesozoodynamics	708,001,062	1,052,001,578	0.673	59.8%	0.0%	19.2%	33.4%
_int_free	642,000,963	1,092,001,638	0.588	54.0%	29.1%	0.0%	36.8%
malloc	632,000,948	1,088,001,632	0.581	33.1%	17.4%	10.4%	39.2%
phytodynamics	628,000,942	664,000,996	0.946	77.9%	7.0%	0.0%	22.8%
microzoodynamics	622,000,933	916,001,374	0.679	69.9%	0.0%	10.7%	41.6%
for_allocate	600,000,900	1,226,001,839	0.489	41.2%	8.2%	8.3%	42.2%
_int_malloc	544,000,816	1,094,001,641	0.497	55.6%	0.0%	0.0%	69.8%
for_deallocate	412,000,618	974,001,461	0.423	48.1%	13.3%	0.0%	48.1%

Figura 3.3. Bottom up del profiling di BFM STANDALONE - versione base

- le subroutine `phytodinamics`, `mesozoodynamics` e `microzoodynamics` (che da ora in poi saranno indicate come *subroutine di dinamica*), che si occupano delle dinamiche biogeochimiche (trascurando quindi la parte idrodinamica, che viene inserita tramite l'accoppiamento a modelli specifici come riportato in Sezione 1.1) che interessano i nutrienti nelle varie fasi;
- `int_free`, `malloc`, `for_allocate`, `int_malloc` e `for_deallocate`, che denotano un intensivo utilizzo dell'allocazione dinamica della memoria.

I risultati del primo profiling di BFM STANDALONE danno un'idea preliminare delle inefficienze che interessano il codice dal punto di vista microarchitetturale. Nelle prossime sezioni sono presentate le soluzioni proposte per velocizzarne l'esecuzione.

3.2.2 Modifica delle flag di compilazione

Seguendo un approccio top-down nell'ottimizzazione di BFM, è stato conveniente cominciare ad apportare modifiche poco invasive, e solo successivamente occuparsi del refactoring del codice. Senza modificare il codice, quindi, il primo intervento effettuato ha riguardato le flag di compilazione.

La compilazione del programma avviene tramite makefile. Le specifiche di compilazione sono incluse attraverso il file `compilers/x86_64.LINUX.intel.inc` e le flag opzionali per il compilatore `ifort` sono le seguenti:

```
11 FFLAGS_OPT= -O2 -g -fno-math-errno -unroll=3 -opt-subscript-in-range -align
     all  -cpp -heap-arrays
```

Alcune delle flag elencate[2], con cui BFM viene normalmente compilato, possono avere un forte impatto sulle performance. Le flag a cui ci si riferisce sono:

- `-O2`: abilita ottimizzazioni quali vettorizzazione automatica, inlining e unrolling.
- `-unroll=3`: imposta a 3 il massimo fattore di unrolling in un loop.
- `-align all`: forza il compilatore a "impacchettare" i dati in memoria in blocchi a cui il processore riesce ad accedere più velocemente.
- `-heap-arrays`: alloca nello heap invece che nello stack gli array temporanei e automatici. Vengono generati array temporanei quando, ad esempio, si assegna ad un array il risultato di un'operazione coinvolgente quello stesso array; gli array automatici sono invece quelli creati all'interno di una procedura, e la cui dimensione non è nota a priori. Le memorie stack e heap sono rispettivamente quelle in cui sono conservate le variabili statiche e quelle dinamiche. L'allocazione e la deallocazione nello

[2]Da notare il fatto che quando è specificata la flag per la generazione dei simboli di debug `-g` (aggiunge in chiaro i nomi delle subroutine nell'eseguibile), è necessario esplicitare la flag di ottimizzazione `-O2`, perché la predefinita diventa `-O0`. Le prime esecuzioni di BFM nel lavoro di tirocinio, in effetti, risultavano stranamente lente: era perché era stato dato per scontato che la flag di compilazione predefinita continuasse ad essere `-O2`!

heap della memoria sono normalmente a carico del programmatore, che le incorpora nel codice sorgente attraverso apposite istruzioni. L'attivazione della suddetta flag aumenta l'utilizzo delle funzioni per l'allocazione della dinamica della memoria, il quale può avere un impatto sulle performance.

Sono state eseguite delle prove togliendo e rimettendo queste flag, o provando altre ottimizzazioni nel caso della `-O2`, e sono state tratte le seguenti conclusioni: mentre `-unroll=3` e `-align all` non sembrano avere un impatto significativo sulle performance, le flag `-heap-arrays` e `-O2` influenzano sensibilmente i tempi di esecuzione.

Il time to solution una volta rimossa `-heap-arrays` è:

$$\text{time to solution}_{\text{no -heap-arrays}} = 1.74 \text{ s}$$

che significa uno speed up[3] di 1.9x! Rieseguendo il profiling con VTune, sono stati ottenuti i risultati riportati in Figura 3.4: confrontando coi risultati precedenti in Figura 3.3, si può osservare come le istruzioni che gestiscono l'allocazione dinamica della memoria (`malloc` ecc ...) non compaiano più tra le prime elencate nel Bottom up, ovvero tra le più costose in termini di clockticks. Eliminando `-heap-arrays`, si riduce il numero di allocazioni e deallocazioni che, come evidenziato in Figura 3.3, risultano essere problematiche per il front-end e soggette a bad speculation quando sono in grande numero.

Non sono state individuate chiaramente, nel corso del tirocinio, le ragioni per cui la flag `-heap-arrays` fosse presente. In generale, essa è utile per evitare l'eventualità di stack overflow e segmentation fault [4] conseguenti all'utilizzo di array automatici o temporanei, nel caso di dataset particolarmente grossi. Per le sue applicazioni nella pratica, OGSTM-BFM è però lanciato sul cluster senza limitare lo stack, cioè specificando l'opzione `limited-stack-size=unlimited`. La flag `-heap-arrays` risulta dunque inutile per evitare il totale riempimento dello stack. Lanci sul cluster delle vecchie versioni di OGSTM-BFM non hanno evidenziato particolari differenze nei tempi di esecuzione con l'inclusione o meno di `-heap-arrays`; mentre, alla versione attuale, quella flag diventa invece determinante.

L'altra flag critica a livello di prestazioni è `-O2`. Generalmente questa è la flag di ottimizzazione consigliata, tuttavia bisogna considerare che il programma in esame è front-end bound. Come già detto, il fatto che il bottleneck sia localizzato nel front-end è spesso dovuto a codici grossi e impostati male; il problema della flag `-O2` è che attiva ottimizzazioni che incrementano ulteriormente le dimensioni del codice, come il loop unrolling e il function inlining, e quindi la situazione potrebbe essere quella in cui l'ottimizzazione peggiora le performance invece di migliorarle. Si è allora testato BFM STANDALONE dopo averlo ricompilato con una ottimizzazione meno intensa, ovvero la `-O1`. Il time to solution dopo questa modifica è risultato:

$$\text{time to solution}_{\text{no -heap-arrays, -O1}} = 1.42 \text{ s}$$

[3] Si definisce speed up il rapporto tra il time to solution di partenza e quello più piccolo conseguente all'ottimizzazione.

Function / Call Stack	Clockticks	Instructions Retired	CPI Rate	Front-End Bound	Bad Speculation	Back-E... Bound	Retiring
flux_vector	682,001,023	1,426,002,139	0.478	33.9%	0.0%	18.5%	52.4%
flux_vector	600,000,900	1,340,002,010	0.448	45.8%	6.4%	0.0%	53.2%
phytodynamics	532,000,798	534,000,801	0.996	63.1%	2.1%	1.8%	33.1%
mesozoodynamics	484,000,726	608,000,912	0.796	64.8%	0.0%	0.0%	35.2%
microzoodynamics	440,000,660	600,000,900	0.733	67.5%	2.5%	3.7%	26.2%
pelglobaldynamics	218,000,327	240,000,360	0.908	88.3%	0.0%	0.0%	30.3%
correct_flux_output	192,000,288	392,000,588	0.490	37.2%	20.1%	5.5%	37.2%
pelbacdynamics	180,000,270	166,000,249	1.084	82.5%	0.0%	0.0%	21.4%
integrationrk2	134,000,201	290,000,435	0.462	53.4%	16.4%	0.0%	45.1%
[Outside any known module]	116,000,174	54,000,081	2.148	33.2%	0.0%	62.1%	4.7%

Figura 3.4. Bottom up del profiling di BFM STANDALONE - senza `-heap-arrays`

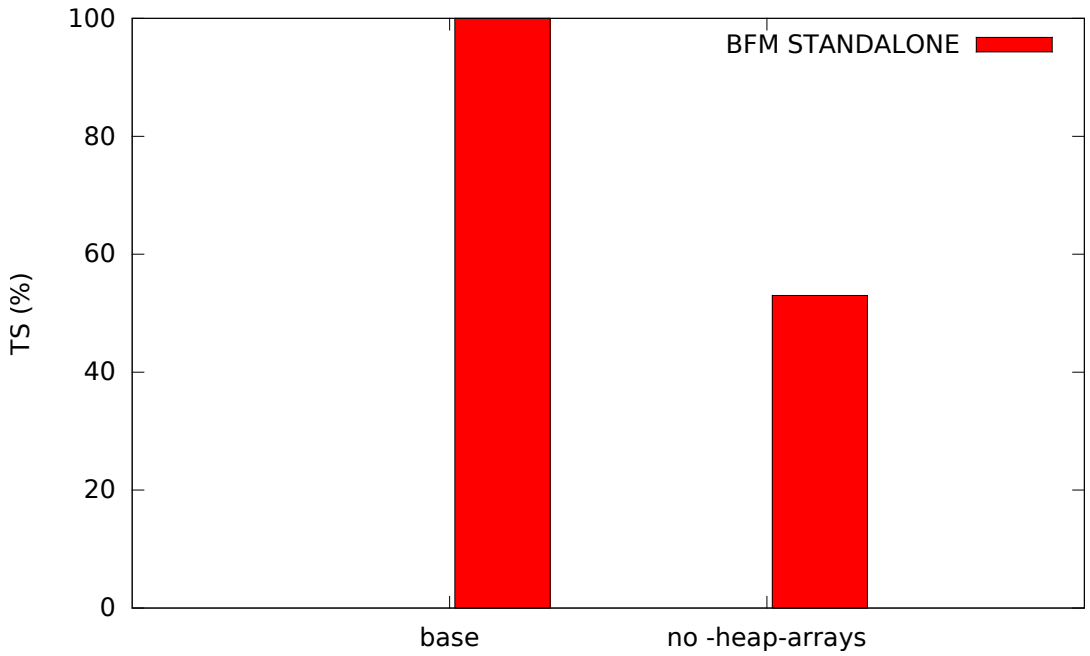


Figura 3.5. Time to solution percentuali rispetto al time to solution del codice di partenza - senza `-heap-arrays`

il quale, a seguito di un ulteriore speed up di 1,2x, ha confermato l'intuizione. Le Figure 3.6 e 3.7 mostrano i risultati del profiling con VTune.

Si può osservare come, con la flag `-O1`, il parametro percentuale `Front-End Bound` sia diminuito (Figura 3.6); inoltre la subroutine `flux_vector` è diventata back-end bound, e non più front-end bound, grazie all'alleggerimento del codice ottenuto dalla riduzione del livello di ottimizzazione. Rimangono invece front-end bound le subroutine di dinamica `mesozoodynamics`, `phytodynamics` e `microzoodynamics`, questo perché probabilmente il loro codice è intrinsecamente problematico e si può migliorare, indipendentemente da come lo modifichi l'ottimizzazione apportata dal compilatore.

Per concludere, l'effetto combinato della rimozione di `-heap-arrays` e il passaggio a `-O1` ha permesso di ottenere complessivamente il seguente speed up:

$$\text{speed up}_{\text{fixed compile flags}} = 2.3\text{x}$$

con un passaggio del time to solution da 3.28 s a 1.42 s.

3.2.3 Riduzione di una dimensione di alcune operazioni tra array

Si potrebbero ora ignorare i nuovi avvertimenti di VTune in merito ai problemi che interessano BFM e ritenere soddisfacente il grosso speed up ottenuto. In realtà è stato trovato che, intervenendo sulle subroutine di dinamica (rimaste front-end bound) e su `flux_vector`, si può ottenere un ulteriore speed up. Questa sezione tratterà l'ottimizzazione delle prime quattro subroutine elencate in Figura 3.7: `flux_vector` e le subroutine di dinamica. Per cominciare, si descrive il contesto in cui operano queste subroutine.

Nel BFM STANDALONE, il *main program* chiama in sequenza tutte le subroutine necessarie all'inizializzazione del programma, all'esecuzione della simulazione e alla sua terminazione. Il codice del main program è contenuto nel file `src/standalone/standalone.F90`. Tra le subroutine chiamate nel file sorgente indicato, è presente `EcologyDynamics`: questa subroutine si occupa di gestire le dinamiche dell'ecosistema, nello specifico il ciclo dei nutrienti tra le varie fasi. A seconda del preset utilizzato, `EcologyDynamics` chiama diverse subroutine per l'evoluzione del sistema marino: nel caso qui in esame, il preset è `STANDALONE_PELAGIC` e la subroutine chiamata è `PelagicSystemDynamics`. A sua volta, `PelagicSystemDynamics` si appoggia ad altre subroutine che gestiscono separatamente ogni fase, ovvero le già viste subroutine di dinamica: `PhytoDynamics`, `MesoZooDynamics`, `MicroZooDynamics`... e sono proprio le subroutine indicate come le più problematiche dal profiling. Anche `flux_vector` entra in questo discorso, in quanto è una subroutine utilizzata intensivamente dalle subroutine di dinamica.

Anche se, dopo le modifiche apportate alle flag di compilazione, la subroutine `flux_vector` non risulta più front-end bound a livelli critici, si è andati ad ottimizzare la più costosa in termini di clockticks. Si è quindi sfruttata la funzionalità di VTune (3.1.3), di mostrare il codice di una subroutine riga per riga e, dopo aver compiuto un profiling, individuare le istruzioni più costose e segnalarne il problemi. Sono state cercate dentro `flux_vector` le righe di codice aventi il maggior impatto sulle performance: in Figura 3.9 è riportata la parte di codice di `flux_vector` che VTune ritiene più costosa.

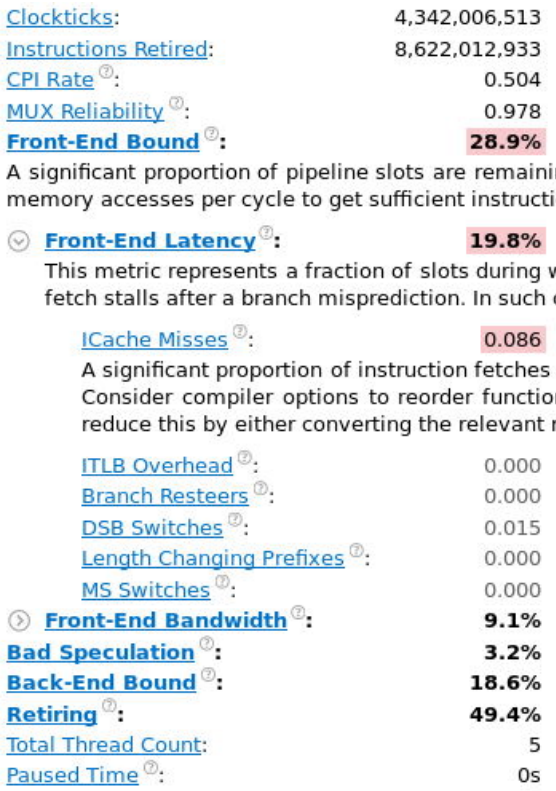


Figura 3.6. Summary del profiling di BFM STANDALONE - senza `-heap-arrays` con `-O1`

Function / Call Stack	Clockticks	Instructions Retired	CPI Rate	Front-End Bound	Bad Speculation	Back-E... Bound	Retiring
flux_vector	1,124,001,686	2,818,004,227	0.399	7.3%	0.0%	25.1%	69.5%
mesozoodynamics	360,000,540	794,001,191	0.453	42.8%	6.1%	2.2%	48.9%
phytodynamics	346,000,519	714,001,071	0.485	35.0%	0.0%	7.8%	57.2%
microzoodynamics	296,000,444	580,000,870	0.510	27.9%	0.0%	38.7%	35.3%
correct_flux_output	256,000,384	630,000,945	0.406	15.0%	2.1%	24.8%	58.0%
calchplus	158,000,237	90,000,135	1.756	7.0%	3.5%	79.1%	10.4%
integrationrk2	126,000,189	294,000,441	0.429	17.5%	17.5%	17.1%	48.0%
__libm_exp_e7	114,000,171	164,000,246	0.695	28.9%	4.8%	27.6%	38.6%
pelglobaldynamics	108,000,162	304,000,456	0.355	25.5%	0.0%	3.2%	71.3%
[Outside any known module]	92,000,138	62,000,093	1.484	59.8%	6.0%	34.2%	0.0%

Figura 3.7. Bottom up del profiling di BFM STANDALONE - senza `-heap-arrays` con `-O1`

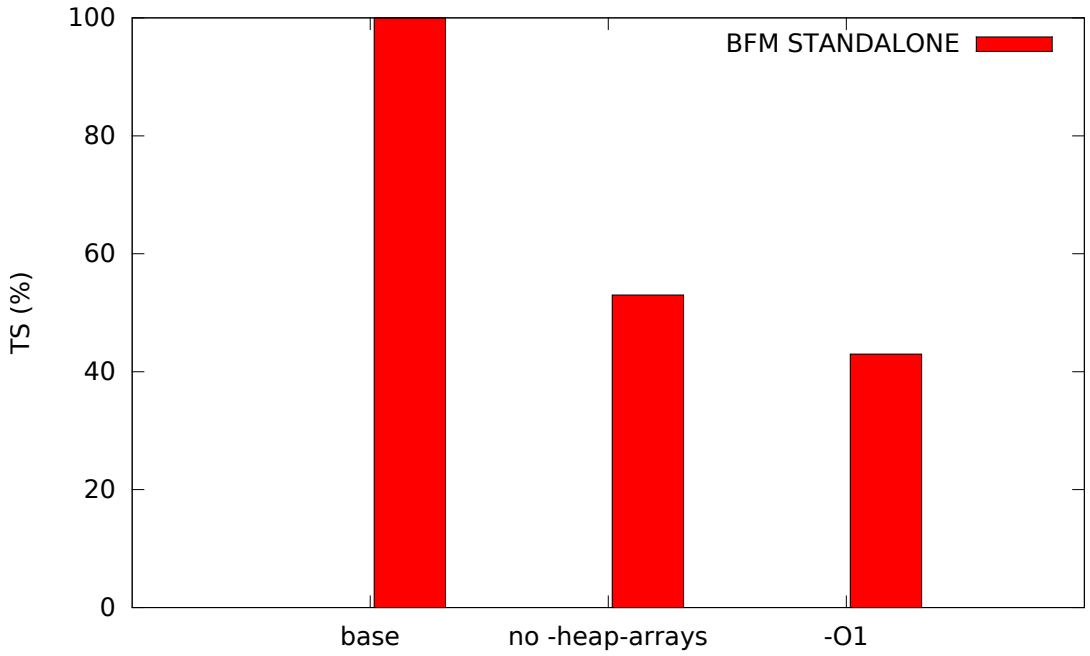


Figura 3.8. Time to solution percentuali rispetto al time to solution del codice di partenza - senza `-heap-arrays` con `-O1`

So. Lin.	Source	Clockticks
62	if (origin /= destination) then	2,000,003
63	select case (iiSub)	8,000,012
64	case (iiPel)	
65	D3SOURCE(origin,:) = D3SOURCE(origin,:) - &	216,000,324
66	flux*DAY_PER_SEC	10,000,015
67	D3SOURCE(destination,:) = D3SOURCE(destination,:) + &	144,000,216
68	flux*DAY_PER_SEC	0
69	if(allocated(D3FLUX_MATRIX) .AND. &	50,000,075
70	allocated(D3FLUX_MATRIX(origin,destination)%p)) then	206,000,309
71	do j=1, SIZE(D3FLUX_MATRIX(origin,destination)%p)	16,000,024
72	D3FLUX_FUNC(ABS(D3FLUX_MATRIX(origin,destination)%p(j)), :) = &	166,000,249
73	D3FLUX_FUNC(ABS(D3FLUX_MATRIX(origin,destination)%p(j)), :) + &	
74	(SIGN(1, D3FLUX_MATRIX(origin,destination)%p(j)) * flux)	36,000,054
75	end do	2,000,003
76	endif	

Figura 3.9. Statistiche sulle istruzioni di `flux_vector` per BFM STANDALONE - senza `-heap-arrays` con `-O1`

A parte il controllo dell'allocazione di `D3FLUX_MATRIX(origin,destination)%p`, le istruzioni più costose sono quelle che aggiornano gli array `D3SOURCE` e `D3FLUX_FUNC`.

Si è quindi investigato sulla provenienza ed il dimensionamento di questi array. Con l'utilizzo del comando `grep` della shell Unix, è stato trovato che questi array sono dichiarati nel file `src/BFM/General/ModuleMem.F90` e allocati dalla subroutine `AllocateMem` contenuta nel file `src/BFM/General/AllocateMem.F90`. In particolare, erano di interesse le allocazioni di `D3SOURCE`:

```
74  allocate(D3SOURCE(1:NO_D3_BOX_STATES,1:NO_BOXES),stat=status)
```

e di `D3FLUX_FUNC`:

```
546 allocate(D3FLUX_FUNC(1:NO_D3_BOX_FLUX, 1:NO_BOXES),stat=status)
```

Siccome le operazioni di aggiornamento di questi array fissano il primo indice e corrono lungo il secondo, si è cercato quanti elementi debbano essere aggiornati; in altre parole, quale sia il valore della variabile `NO_BOXES`, che definisce il numero di elementi nella seconda dimensione dell'array.

Cercando `NO_BOXES` sempre con `grep`, è stato trovato che il valore di `NO_BOXES` è fissato dal programma a cui viene accoppiato BFM: nel caso qui in esame, BFM gira in modalità STANDALONE e l'assegnazione del valore di `NO_BOXES` è contenuta nel file `src/standalone/standalone.F90`, ed avviene nel modo seguente:

```
186 NO_BOXES_X  = nboxes
187 NO_BOXES_Y  = 1
188 NO_BOXES_Z  = 1
189 NO_BOXES    = NO_BOXES_X * NO_BOXES_Y * NO_BOXES_Z
```

Siccome nello stesso file è contenuta anche la riga:

```
169 nboxes      = 1
```

abbiamo che `NO_BOXES=1`. A questo punto è lecito chiedersi se effettuare le operazioni in questo modo:

```
D3SOURCE(origin,:) = D3SOURCE(origin,:) - &
                     flux*DAY_PER_SEC
```

sia conveniente. In effetti la notazione adottata, in cui si utilizzano i due punti '`:`' per compiere un ciclo su tutti gli indici, non rende conto di quello che deve essere eseguito veramente: un'operazione tra scalari. Più in generale, ogni volta che avviene un'operazione tra array aventi una dimensione allocata con `NO_BOXES`, di fatto ciò che deve essere eseguito è un'operazione tra array aventi una dimensione degenere (hanno cioè una dimensione in meno), perché uno dei loro indici (solitamente l'ultimo) può assumere un solo valore.

Il problema è che non è certo che il compilatore non sia in grado di capire autonomamente questa sottigliezza, poiché la compilazione è un processo automatico. Il compilatore, solitamente, esegue le varie fasi meccanicamente, quindi si presuppone che l'utilizzo di '`:`' venga interpretato dal compilatore come l'apertura di un ciclo.

Per capire il comportamento del compilatore `ifort`, si è generato il report di ottimizzazione in fase di compilazione, per vedere come le operazioni tra array aventi una dimensione degenere vengano interpretate. A questo scopo, modificando il file `compilers/x86_64.LINUX.intel.inc`, è stata aggiunta la flag `-qopt-report5`:

```
11 FFLAGS_OPT= -O2 -g -fno-math-errno -unroll=3 -qopt-subscript-in-range -
       align all -cpp -heap-arrays -qopt-report5
```

Per ottenere nel report le segnalazioni di apertura dei cicli, altrimenti non presenti, si è temporaneamente passati alla flag di ottimizzazione `-O2`. Dai risultati riportati alla fine, comunque, si vedrà che i ragionamenti presentati valgono anche utilizzando la flag `-O1`.

Dopo aver compilato con queste flag, i report vengono generati nella sottodirectory della cartella di compilazione `build/tmp/STANDALONE_PELAGIC`. Essi mantengono lo stesso nome del sorgente, con estensione cambiata in `.mem`. Per vedere come sono trattate le operazioni dentro `flux_vector` è stato esaminato il file relativo all'ottimizzazione di `src/BFM/General/ModuleMem.F90`, perché è quello in cui è incluso dal preprocessore `FluxFunctions.h90`, a sua volta contenente `flux_vector`. All'interno del file sono state trovate le seguenti informazioni:

```
706 LOOP BEGIN at /gpfs/work/IscrC_MYMEDBIO/BFM-PROFILING/bfm/build/../src/BFM/
       General/FluxFunctions.h90(65,19)
```

e le righe di codice indicate di `FluxFunctions.h90` sono le seguenti:

```
65 D3SOURCE(origin,:) = D3SOURCE(origin,:) - &
66      flux*DAY_PER_SEC
```

Il report fa chiaramente capire che a quella riga di codice viene iniziato un loop, pur essendo sufficiente un'operazione scalare. Lo stesso avviene per le operazioni successive che utilizzano '`:`' invece di indicare esplicitamente l'unico indice permesso, cioè 1. Se un loop introduce un overhead dovuto ai controlli della variabile contatore, tutto ciò comporta uno spreco di risorse.

Si è quindi pensato di verificare se un'operazione tra scalari potesse risultare meno costosa. Per farlo, sono state riscritte, ove possibile, le istruzioni dentro `flux_vector` mettendo un 1 al posto di ':', ed è stato ottenuto il seguente codice:

```
65 D3SOURCE(origin,1) = D3SOURCE(origin,1) - &
66      flux(1)*DAY_PER_SEC
67 D3SOURCE(destination,1) = D3SOURCE(destination,1) + &
68      flux(1)*DAY_PER_SEC
69 if( allocated( D3FLUX_MATRIX ) .AND. &
70       allocated( D3FLUX_MATRIX(origin,destination)%p ) ) then
71     do j=1, SIZE(D3FLUX_MATRIX(origin,destination)%p)
72      D3FLUX_FUNC(ABS(D3FLUX_MATRIX(origin,destination)%p(j)),1)=&
73       D3FLUX_FUNC(ABS(D3FLUX_MATRIX(origin,destination)%p(j)),1)+&
74       (SIGN(1,D3FLUX_MATRIX(origin,destination)%p(j))*flux(1))
75     end do
76 endif
```

Il codice presentato è lo stesso riportato in Figura 3.9, ma con le modifiche sopra riportate. Si è ricompilato (con `-O1`, `-O2` serviva solo a ottenere le informazioni sulle aperture di cicli nel report di ottimizzazione), eseguito un profiling e ottenuto il seguente time to solution:

$$\text{time to solution}_{\text{fixed } \texttt{flux\_vector}} = 1.32 \text{ s}$$

a cui corrisponde uno speed up di 1.1x rispetto al tempo precedente di 1.42 s. La Figura 3.10 mostra il riassunto dei risultati del profiling: i miglioramenti apportati aiutano il fetching e il decoding delle istruzioni nella fase di front-end, rendendo il programma ora anche back-end bound invece che solo front-end bound.

Ottenuto questo risultato, è stato spontaneo chiedersi quanti altri array vengano allocati usando `NO_BOXES` in una delle loro dimensioni. La risposta è *molti*, e incidentalmente sono usati intensivamente proprio dalle subroutine front-end bound rimaste da ottimizzare: `MesoZooDynamics`, `MicroZooDynamics` e `PhytoDynamics`, ovvero le subroutine di dinamica. Per mostrare come si è proceduto a modificare il codice di queste subroutine, si riporta come esempio il lavoro compiuto sul file `src/BFM/Pel/Phyto.F90`. Esso contiene la dichiarazioni di una serie di array `allocatable`, in particolare:

```
86 real(RLEN),allocatable,save,dimension(:) :: phytoc,phyton,phytop,phytos,
     phytol
```

e molti altri, i quali vengono allocati tutti allo stesso modo:

```
108 if (first==0) then
109      first=1
110      allocate(phytoc(NO_BOXES),stat=AllocStatus)
111      if (AllocStatus  /= 0) stop "error allocating phytoc"
112      allocate(phyton(NO_BOXES),stat=AllocStatus)
113      if (AllocStatus  /= 0) stop "error allocating phyton"
114      allocate(phytop(NO_BOXES),stat=AllocStatus)
115      if (AllocStatus  /= 0) stop "error allocating phytop"
116      allocate(phytos(NO_BOXES),stat=AllocStatus)
117      if (AllocStatus  /= 0) stop "error allocating phytos"
118      allocate(phytol(NO_BOXES),stat=AllocStatus)
119      if (AllocStatus  /= 0) stop "error allocating phytol"
```

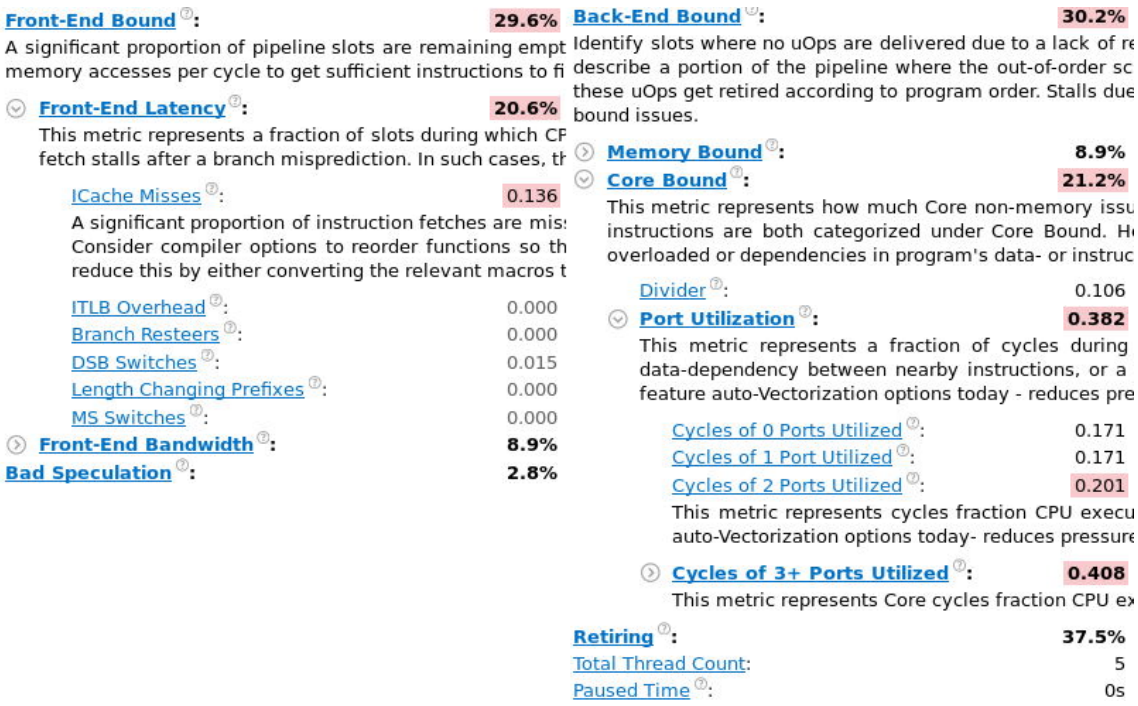


Figura 3.10. Summary del profiling di BFM STANDALONE - con indici espliciti in `flux_vector`

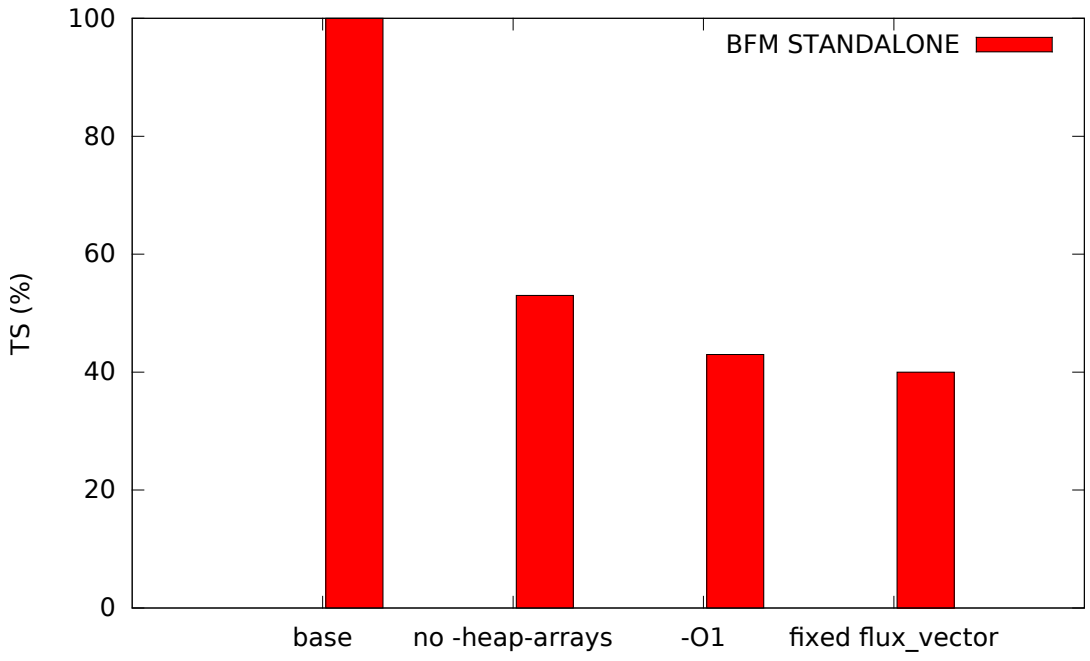


Figura 3.11. Time to solution percentuali rispetto al time to solution del codice di partenza - con indici espliciti in `flux_vector`

```
120       allocate(r(NO_BOXES),stat=AllocStatus)
121       if (AllocStatus  /= 0) stop "error allocating r"
```

Ora si prenda ad esempio una riga del codice contenuto nel file dopo le dichiarazioni delle variabili:

```
415   sra   =    p_pu_ra(phyto)*( sum - sea - seo)   ! activity
```

A parte `p_pu_ra(phyto)`, che è un elemento di un array monodimensionale (quindi uno scalare), tutte le variabili contenute nell'istruzione sono vettori allocati allo stesso modo di quelli presentati prima. Le subroutine di dinamica sono disseminate di istruzioni di questo tipo, e per ciascuna di esse viene inutilmente aperto un ciclo. Sono state quindi riscritte anche queste istruzioni esplicitando l'unico indice possibile, nel modo seguente:

```
415   sra(1)   =    p_pu_ra(phyto)*( sum(1) - sea(1) - seo(1))
```

Dopo aver effettuato questa operazione *molto pazientemente* per i file `Phyto.F90`, `MesoZoo.F90` e `MicroZoo.F90`, si è ricompilato ed eseguito il profiling, ottenendo il seguente time to solution:

$$\text{time to solution}_{\text{fixed dynamics subroutines}} = 1.15 \text{ s}$$

che equivale ad uno speed up di 1.1x, rispetto al tempo precedente di 1.32 s. I risultati del profiling sono riportati nelle Figure 3.12 e 3.13.

Nel Summary in Figura 3.12 è possibile vedere come il programma continui ad essere non solo front-end bound, ma in parte anche back-end bound. Per risolvere il problema VTune consiglia di utilizzare l'autovettorizzazione: il punto è che essa viene abilitata attivando la flag `-O2` [8], che è proprio quella che è stata abbandonata. Se si volessero sfruttare a pieno le ottimizzazioni introdotte dalla `-O2`, bisognerebbe riscrivere il codice in modo che questa flag non causi gli effetti collaterali visti in precedenza. L'idea sarebbe sempre rivederlo per migliorarne l'organizzazione e, dove possibile, ridurne le dimensioni. Nell'immediato, è più pratico accontentarsi di una `-O1` e non modificare il codice. Si potrebbero anche attivare manualmente le funzionalità di autovettorizzazione utilizzando apposite flag di compilazione, ma non vi è stato sufficiente tempo per continuare la ricerca in questa direzione durante il tirocinio.

Un problema che continua a persistere è che il codice è front-end bound. Confrontando i Bottom up nelle Figure 3.7 e 3.13, si può vedere che la subroutine `MicroZooDynamics` ha cessato di essere segnalata come front-end bound; rimangono tali, però, `PhytoDynamics` e `MesoZooDynamics`. Effettivamente, permangono delle righe di codice in cui gli array allocati con `NO_BOXES` vengono trattati come array, e non come scalari: ad esempio, in `flux_vector` stessa l'argomento `flux` viene sempre passato come array, nonostante si possa trattare come scalare; a meno di ridefinire il tipo di dato accettato da `flux_vector` (ma farlo costringerebbe ad un refactoring dell'intero codice), non è possibile effettuare il passaggio in altro modo. `flux_vector` non è nemmeno l'unica subroutine dove tutto ciò avviene, lo stesso vale per `MM_vector`, che viene chiamata in diverse occasioni all'interno di `MesoZooDynamics` e simili. Insomma, le modifiche apportate sono solo un primo passo e sono servite a mostrare che, riducendo di uno le dimensioni degli array che hanno una dimensione allocata con `NO_BOXES`, si può ottenere uno speed up. Sarebbe ora interessante ridefinire tutte le variabili che vengono allocate con `NO_BOXES` togliendo loro

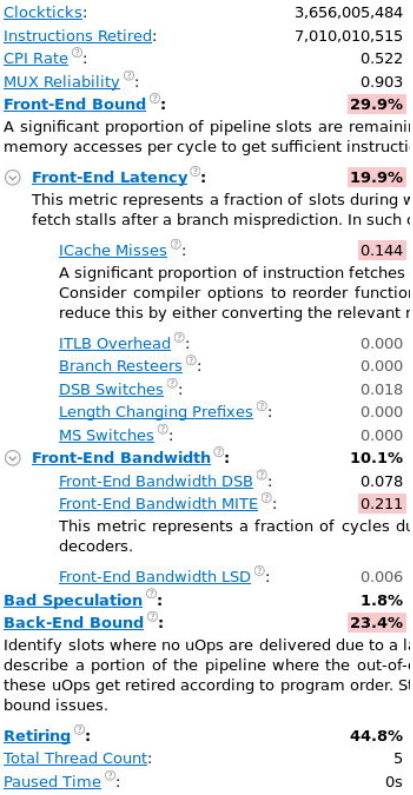


Figura 3.12. Summary del profiling di BFM STANDALONE - con indici espliciti nelle subroutine di evoluzione

Function / Call Stack	Clockticks	Instructions Retired	CPI Rate	Front-End Bound	Bad Speculation	Back-E... Bound	Retiring
flux_vector	848,001,272	1,986,002,979	0.427	13.6%	11.0%	16.3%	59.0%
correct_flux_output	292,000,438	588,000,882	0.497	7.5%	0.0%	51.0%	45.2%
microzoodynamics	238,000,357	460,000,690	0.517	11.6%	0.0%	58.4%	55.5%
phytodynamics	228,000,342	368,000,552	0.620	31.4%	4.8%	20.4%	43.4%
mesozoodynamics	206,000,309	432,000,648	0.477	32.0%	0.0%	22.6%	53.4%
[Outside any known module]	118,000,177	58,000,087	2.034	100.0%	0.0%	0.0%	0.0%
pelglobaldynamics	118,000,177	370,000,555	0.319	55.9%	0.0%	0.0%	69.9%
integrationrk2	112,000,168	312,000,468	0.359	39.3%	14.7%	0.0%	88.4%
__libm_exp_e7	108,000,162	128,000,192	0.844	10.2%	25.5%	38.9%	25.5%
fixed_quota_flux_vector	98,000,147	178,000,267	0.551	11.2%	0.0%	60.7%	28.1%

Figura 3.13. Bottom up del profiling di BFM STANDALONE - con indici espliciti nelle subroutine di evoluzione

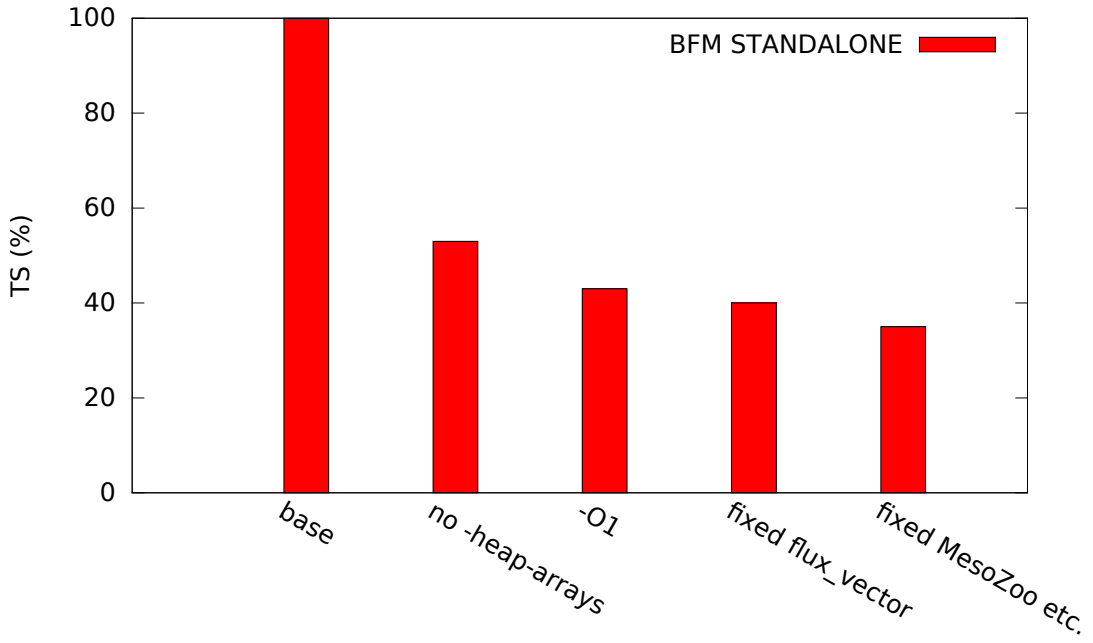


Figura 3.14. Time to solution percentuali rispetto al time to solution del codice di partenza - con indici espliciti nelle subroutine di evoluzione

una dimensione e riscrivere tutto il codice di conseguenza: se tutto il resto del programma si comporta come `MicroZooDynamics`, si potrebbe sperare che cessi di essere front-end bound. Se ciò non avvenisse, sarebbe allora necessario rivedere davvero il layout del codice delle subroutine di dinamica.

È bene sottolineare che il valore della variabile `NO_BOXES` non è uguale ad 1 in ogni caso, altrimenti non si spiegherebbero l'esistenza della variabile e il suo utilizzo per allocare una particolare dimensione di diversi array: per certi utenti di BFM, `NO_BOXES` assume un altro valore. Nella configurazione utilizzata da OGS, comunque, la dimensione diventa degenere, ossia `NO_BOXES=1`, quindi il lavoro svolto ha certamente un'utilità a livello pratico, pur non essendo del tutto generale.

Per concludere questa sezione, si riporta lo speed up complessivo ottenuto esplicitando gli indici 1 dove possibile. Si è passati da un tempo di esecuzione medio di 1.42 s, ottenuto alla fine della sezione precedente, a 1.15 s. Lo speed up totale è:

$$\text{speed up}_{\text{explicit indices}} = 1.2\text{x}$$

più modesto rispetto a quello della sezione precedente, ma comunque notevole.

3.3 Profiling e ottimizzazione di OGSTM-BFM

All'interno del sistema previsionale Copernicus, il software BFM-V5 è sempre accoppiato al software di trasporto OGSTM. Come si vedrà a breve dal primo profiling, BFM causa, in questa condizione, un bottleneck nelle performance del sistema accoppiato. Per questa ragione si è cominciato lavorando su BFM STANDALONE, più snello da compilare ed eseguire, per poi vedere come i miglioramenti ivi introdotti si ripercuotano anche nell'accoppiamento.

Il profiling di OGSTM-BFM è stato eseguito utilizzando il preset `OGS_PELAGIC`, introdotto nella Sezione 3.1.2. Dopo il primo profiling è stato ottenuto il seguente time to solution:

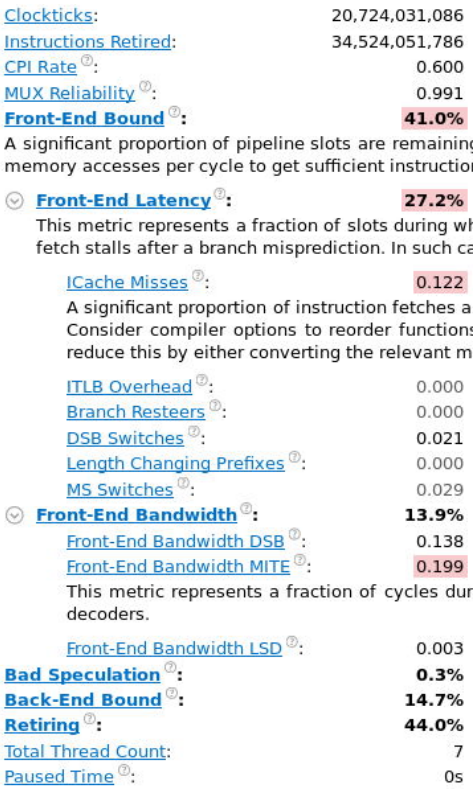
$$\text{time to solution}_{\text{base}} = 6.31 \text{ s}$$

e le Figure 3.15 e 3.16 riportano i risultati dell'analisi. Si può osservare come il programma risulti front-end bound e come le subroutine responsabili di ciò siano le stesse individuate in BFM STANDALONE.

Effettuando il cambio delle flag di compilazione, togliendo `-heap-arrays` e passando a `-O1`, il time to solution diventa:

$$\text{time to solution}_{\text{ no -heap-arrays, -O1}} = 3.66 \text{ s}$$

che corrisponde ad uno speed up di 1.7x; il profiling fornisce i risultati riportati nelle Figure 3.17 e 3.18. Si può osservare come in OGSTM-BFM basti effettuare il cambio delle flag di compilazione per rendere il programma prevalentemente back-end bound, diversamente da quanto visto in BFM STANDALONE. Lo speed up è più piccolo del 1.9x ottenuto in BFM STANDALONE, ma comunque notevole.



Clockticks:	20,724,031,086
Instructions Retired:	34,524,051,786
CPI Rate:	0.600
MUX Reliability:	0.991
Front-End Bound:	**41.0%**
A significant proportion of pipeline slots are remaining memory accesses per cycle to get sufficient instructio	
Front-End Latency:	**27.2%**
This metric represents a fraction of slots during wh fetch stalls after a branch misprediction. In such ca	
ICache Misses:	0.122
A significant proportion of instruction fetches a Consider compiler options to reorder functions reduce this by either converting the relevant m	
ITLB Overhead:	0.000
Branch Resteers:	0.000
DSB Switches:	0.021
Length Changing Prefixes:	0.000
MS Switches:	0.029
Front-End Bandwidth:	**13.9%**
Front-End Bandwidth DSB:	0.138
Front-End Bandwidth MITE:	0.199
This metric represents a fraction of cycles dur decoders.	
Front-End Bandwidth LSD:	0.003
Bad Speculation:	**0.3%**
Back-End Bound:	**14.7%**
Retiring:	**44.0%**
Total Thread Count:	7
Paused Time:	0s

Figura 3.15. Summary del profiling di OGSTM-BFM - versione base

Function / Call Stack	Clockticks	Instructions Retired	CPI Rate	Front-End Bound	Bad Speculation	Back-E... Bound	Retiring	Module
flux_vector	1,730,002,595	3,496,005,244	0.495	43.6%	0.0%	9.7%	51.5%	ogstm.xx
flux_vector	1,578,002,367	2,946,004,419	0.536	41.5%	9.8%	4.2%	44.6%	ogstm.xx
smolar_	1,518,002,277	3,990,005,985	0.380	4.0%	7.2%	0.0%	88.8%	ogstm.xx
mesozoodynamics	1,242,001,863	1,892,002,838	0.656	60.2%	0.0%	14.1%	39.9%	ogstm.xx
phytodynamics	1,046,001,569	1,344,002,016	0.778	58.9%	0.0%	20.6%	30.0%	ogstm.xx
microzoodynamics	1,040,001,560	1,620,002,430	0.642	64.5%	0.0%	0.0%	41.2%	ogstm.xx
[Outside any known module]	988,001,482	412,000,618	2.398	17.3%	1.7%	56.6%	24.5%	
malloc	928,001,392	1,520,002,280	0.611	45.0%	13.6%	0.0%	45.6%	libc-2.12.so
_int_free	900,001,350	2,010,003,015	0.448	36.1%	8.6%	19.3%	36.1%	libc-2.12.so
for_allocate	872,001,308	1,836,002,754	0.475	49.2%	8.2%	0.0%	46.0%	ogstm.xx

Figura 3.16. Bottom up del profiling di OGSTM-BFM - versione base

Clockticks:	11,946,017,919
Instructions Retired:	22,844,034,266
CPI Rate:	0.523
MUX Reliability:	0.984
Front-End Bound:	**18.7%**
Bad Speculation:	**1.8%**
Back-End Bound:	**27.3%**

Identify slots where no uOps are deliver
describe a portion of the pipeline wher
these uOps get retired according to prog
bound issues.

Retiring:	**52.3%**
Total Thread Count:	7
Paused Time:	0s

Figura 3.17. Summary del profiling di OGSTM-BFM - compile flag corrette

Function / Call Stack	Clockticks	Instructions Retired	CPI Rate	Front-End Bound	Bad Speculation	Back-E... Bound	Retiring	Module
flux_vector	2,104,003,156	5,140,007,710	0.409	9.7%	12.0%	21.3%	57.0%	ogstm.xx
smolar_	1,478,002,217	4,046,006,069	0.365	2.6%	6.7%	14.8%	75.9%	ogstm.xx
[Outside any known module]	966,001,449	414,000,621	2.333	25.6%	0.0%	52.7%	22.2%	
trczdf_	638,000,957	460,000,690	1.387	2.6%	0.0%	77.6%	19.8%	ogstm.xx
trchdf_	630,000,945	1,594,002,391	0.395	1.7%	0.0%	10.1%	90.8%	ogstm.xx
phytodynamics	572,000,858	1,240,001,860	0.461	32.7%	24.0%	0.0%	47.1%	ogstm.xx
__intel_memset	564,000,846	402,000,603	1.403	2.0%	6.8%	77.6%	13.7%	ogstm.xx
mesozoodynamics	556,000,834	1,166,001,749	0.477	39.6%	2.0%	4.0%	54.4%	ogstm.xx
microzoodynamics	502,000,753	1,006,001,509	0.499	27.4%	3.3%	16.7%	52.6%	ogstm.xx
calchplus	278,000,417	132,000,198	2.106	9.9%	0.0%	80.2%	9.9%	ogstm.xx

Figura 3.18. Bottom up del profiling di OGSTM-BFM - compile flag corrette

Infine, riducendo di una dimensione le operazioni tra array allocati con `NO_BOXES` in `flux_vector` e nelle subroutine di dinamica, attraverso l'uso di indici 1 espliciti, il time to solution diventa:

$$\text{time to solution}_{\text{explicit indices}} = 3.18 \text{ s}$$

per uno speed up di 1.2x; il profiling fornisce i risultati riportati nelle Figure 3.19 e 3.20. Confrontando quest'ultima figura con la Figura 3.18, è interessante osservare come soprattutto `flux_vector` diventi meno costosa in termini di clockticks dopo l'ultima modifica apportata. Lo speed up è in linea con quello ottenuto su BFM STANDALONE.

Prima di concludere, si osservi che sulla versione finale di OGSTM-BFM gravano gli stessi problemi riscontrati per la versione finale di BFM STANDALONE: il passaggio alla flag di compilazione `-O1` fa sì che il programma sia meno front-end bound, ma lo rende inefficiente quando si tratta di eseguire le istruzioni decodificate nella fase di back-end.

3.4 Riassunto dei risultati ottenuti

Nella Tabelle 3.1 e 3.2 sono riportati i risultati ottenuti per ciascuno dei passaggi effettuati durante l'ottimizzazione di BFM STANDALONE e OGSTM-BFM.

Tabella 3.1. Risultati ottenuti nell'ottimizzazione di BFM STANDALONE

	Tempo (s)	Riduzione tempo (%)	Speed up
versione base	3.28	—	—
no `-heap-arrays`	1.74	47	1.9x
`-O1`	1.42	18	1.2x
fixed `flux_vector`	1.32	7	1.1x
fixed `MesoZoo` etc.	1.15	13	1.1x
totale	—	65	2.8x

Tabella 3.2. Risultati ottenuti nell'ottimizzazione di OGSTM-BFM

	Tempo (s)	Riduzione tempo (%)	Speed up
versione base	6.31	—	—
no `-heap-arrays` `-O1`	3.66	42	1.7x
fixed `flux_vector` fixed `MesoZoo` etc.	3.18	13	1.2x
totale	—	50	2.0x

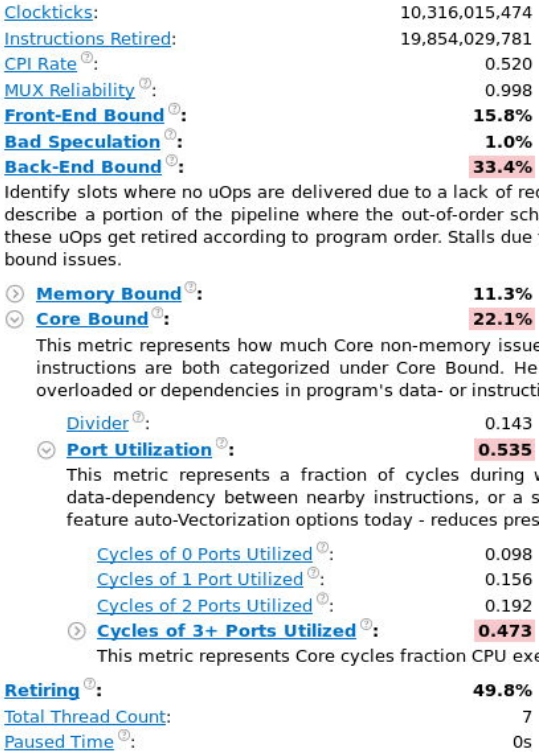


Figura 3.19. Summary del profiling di OGSTM-BFM - indici espliciti

Function / Call Stack	Clockticks	Instructions Retired	CPI Rate	Front-End Bound	Bad Speculation	Back-E... Bound	Retiring	Module
smolar_	1,460,002,190	3,908,005,862	0.374	3.8%	7.5%	29.2%	59.5%	ogstm.xx
flux_vector	1,340,002,010	3,582,005,373	0.374	13.1%	6.6%	5.6%	74.7%	ogstm.xx
[Outside any known module]	908,001,362	316,000,474	2.873	20.6%	4.8%	52.8%	21.8%	
trchdf_	640,000,960	1,594,002,391	0.402	1.7%	0.0%	37.3%	63.6%	ogstm.xx
trczdf_	624,000,936	470,000,705	1.328	2.6%	0.0%	73.6%	25.6%	ogstm.xx
__intel_memset	528,000,792	404,000,606	1.307	6.2%	1.0%	66.7%	26.0%	ogstm.xx
phytodynamics	382,000,573	726,001,089	0.526	30.2%	5.8%	19.4%	44.6%	ogstm.xx
microzoodynamics	372,000,558	864,001,296	0.431	20.7%	0.0%	24.6%	59.1%	ogstm.xx
mesozoodynamics	328,000,492	792,001,188	0.414	38.6%	0.0%	12.8%	50.3%	ogstm.xx
calchplus	282,000,423	164,000,246	1.720	3.9%	5.9%	80.5%	9.8%	ogstm.xx

Figura 3.20. Bottom up del profiling di OGSTM-BFM - indici espliciti

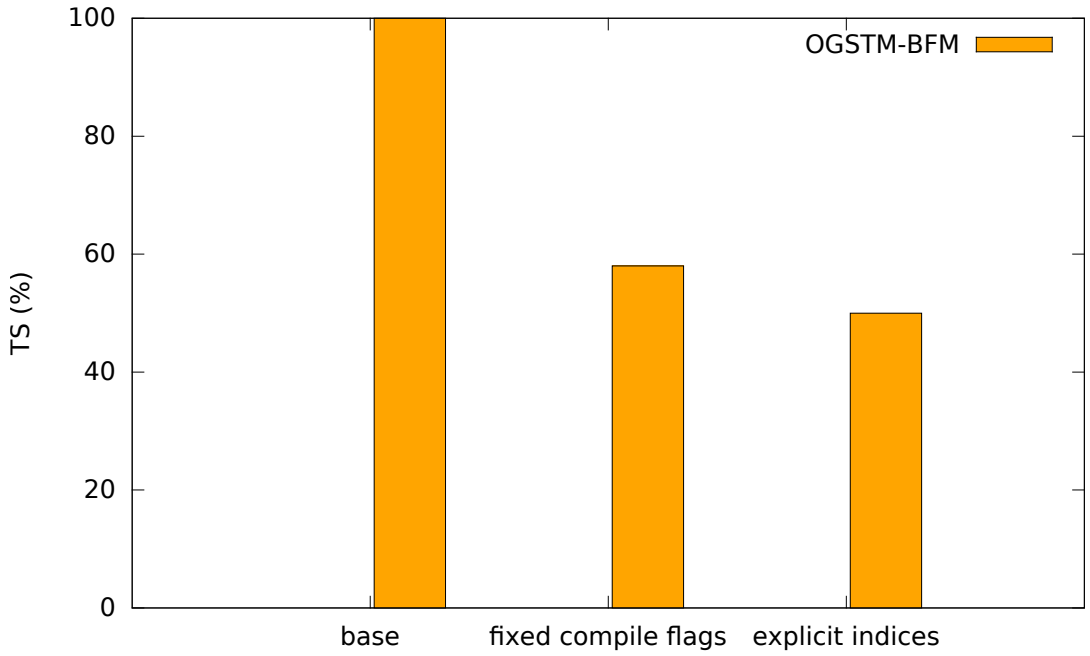


Figura 3.21. Time to solution percentuali rispetto al time to solution del codice di partenza - OGSTM-BFM ad ogni passo di ottimizzazione

Capitolo 4

Altre prove su BFM

Nel precedente capitolo sono state trattate le modifiche apportate a BFM che hanno permesso di ottenere i risultati più notevoli dal punto di vista pratico. I risultati mostrati sono però il prodotto di una scrematura del lavoro di tirocinio, la quale ha eliminato vari tentativi andati a vuoto lungo il percorso di ottimizzazione. Sarebbe un peccato non rendere conto di qualche modesto risultato che, anche non avendo immediate ricadute sullo speed up finale di BFM, è costato tempo e lavoro; inoltre può risultare sempre interessante, dal punto di vista puramente fenomenologico, vedere come reagisce il programma a certe modifiche, e interrogarsi sul perché reagisca in un certo modo.

Si riportano quindi nelle sezioni successive alcune prove di minore rilevanza, assieme ai risultati ottenuti.

4.1 Impiego di subroutine ausiliarie in `flux_vector`

All'inizio del tirocinio non è stata immediatamente valutata l'idea di modificare alcuna flag di compilazione all'infuori di quella di ottimizzazione. Come risultato, si è passati alla flag `-O1` e sono poi state effettuate diverse prove senza rimuovere la flag `-heap-arrays`. Le modifiche discusse in questa sezione sono valide compilando mantenendo `-heap-arrays`, pertanto non hanno rilevanza inserite nel contesto discusso nel precedente capitolo, in cui `-heap-arrays` viene rimossa.

Si riportano di seguito i primi risultati ottenuti mantenendo `-heap-arrays` e passando a `-O1`. Il time to solution è:

$$\text{time to solution}_{\texttt{-O1}} = 2.64 \text{ s}$$

il quale comporta uno speed up rispetto alla versione base di 1.2x, in linea con quanto visto nel capitolo precedente. Come ci si aspetta, cambiando la flag di ottimizzazione da `-O2` a `-O1`, pur mantenendo `-heap-arrays`, il programma risulta meno front-end bound della versione di partenza, come anche le sue subroutine: a questo proposito, si confrontino i risultati del profiling di questa versione (Figure 4.1 e 4.2) con quelli del profiling della versione base (Figure 3.2 e 3.3).

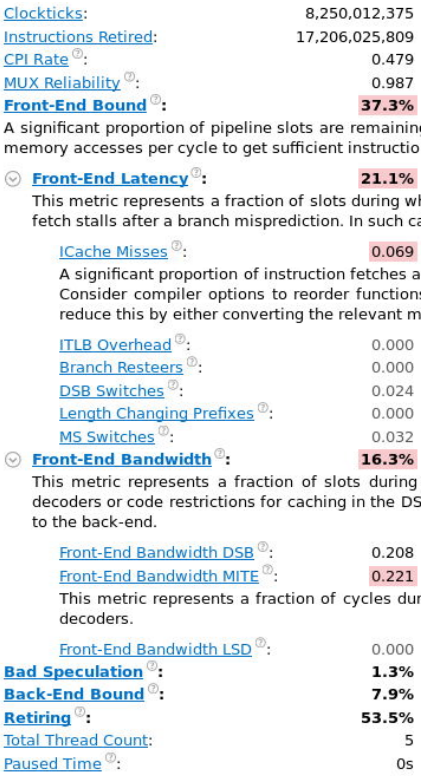


Figura 4.1. Summary del profiling di BFM STANDALONE - con `-O1`

Function / Call Stack	Clockticks	Instructions Retired	CPI Rate	Front-End Bound	Bad Speculation	Back-E... Bound	Retiring
flux_vector	1,482,002,223	3,442,005,163	0.431	22.6%	6.3%	13.9%	57.2%
_int_free	620,000,930	1,170,001,755	0.530	37.3%	18.6%	6.0%	38.1%
for_allocate	542,000,813	1,310,001,965	0.414	44.6%	0.0%	0.0%	62.9%
_int_malloc	532,000,798	1,298,001,947	0.410	42.4%	4.1%	0.0%	54.8%
malloc	504,000,756	1,134,001,701	0.444	40.4%	0.0%	11.6%	61.1%
phytodynamics	476,000,714	892,001,338	0.534	40.4%	8.1%	11.0%	40.4%
mesozoodynamics	466,000,699	1,072,001,608	0.435	49.6%	0.0%	0.0%	55.5%
microzoodynamics	416,000,624	934,001,401	0.445	51.6%	0.0%	2.2%	56.9%
for_deallocate	394,000,591	820,001,230	0.480	41.9%	0.0%	13.5%	64.2%
correct_flux_output	332,000,498	680,001,020	0.488	5.0%	23.2%	32.1%	39.8%

Figura 4.2. Bottom up del profiling di BFM STANDALONE - con `-O1`

Anche se a questo punto è noto che rimuovere `-heap-arrays` sistemerebbe diversi problemi tra quelli mostrati in Figura 4.2, in una fase del tirocinio ci è interrogati direttamente su come si potesse cambiare il codice per migliorare le prestazioni. Assecondando la Figura 4.2, si è quindi deciso che convenisse intervenire sulla subroutine più costosa: `flux_vector`, la quale è segnalata come front-end bound.

Si è provveduto, col supporto di VTune, a cercare dentro `flux_vector` le righe di codice aventi il maggior impatto sulle performance: in Figura 4.3 è riportata la parte di codice di `flux_vector` che VTune ritiene più costosa. Le istruzioni più problematiche

So. Li.	Source	Clockticks	Instructions Retired	CPI Rate	Front-End Bound	Bad Spe...	Back-End Bound
61	#endif						
62	if (origin /= destination) then	2,000,003	12,000,018	0.167	0.0%	0.0%	0.0%
63	select case (iiSub)	8,000,012	20,000,030	0.400	68.8%	0.0%	31.2%
64	case (iiPel)						
65	D3SOURCE(origin,:) = D3SOURCE(origin,:) - &	196,000,294	406,000,609	0.483	28.1%	39.3%	0.0%
66	flux*DAY_PER_SEC	2,000,003	8,000,012	0.250	0.0%	0.0%	100.0%
67	D3SOURCE(destination,:) = D3SOURCE(destination,:) + &	152,000,228	386,000,579	0.394	7.2%	0.0%	31.2%
68	flux*DAY_PER_SEC	2,000,003	2,000,003	1.000	0.0%	0.0%	100.0%
69	if(allocated(D3FLUX_MATRIX) .AND. &	20,000,030	90,000,135	0.222	0.0%	0.0%	0.0%
70	allocated(D3FLUX_MATRIX(origin,destination)%p)) then	276,000,414	612,000,918	0.451	17.9%	0.0%	16.3%
71	do j=1, SIZE(D3FLUX_MATRIX(origin,destination)%p)	34,000,051	44,000,066	0.773	0.0%	32.4%	2.9%
72	D3FLUX_FUNC(ABS(D3FLUX_MATRIX(origin,destination)%p(j)), :) =	294,000,441	880,001,320	0.334	22.4%	0.0%	0.0%
73	D3FLUX_FUNC(ABS(D3FLUX_MATRIX(origin,destination)%p(j)), :						
74	(SIGN(1, D3FLUX_MATRIX(origin,destination)%p(j)) * flux)	120,000,180	260,000,390	0.462	9.2%	9.2%	22.1%
75	end do	8,000,012	30,000,045	0.267	68.8%	0.0%	0.0%
76	endif						

Figura 4.3. Statistiche sulle istruzioni di `flux_vector` per *BFM Standalone* con `-O1`

sono, come visto anche nel capitolo precedente, quelle che si occupano di aggiornare gli array globali `D3SOURCE` e `D3FLUX_FUNC`. Specialmente le istruzioni contenute nelle righe 65 e 72 vengono indicate come front-end bound. Siccome i risultati in Figura 4.1 suggeriscono che i problemi per il front-end siano nel reperimento delle istruzioni nella instruction cache e nella loro decodifica (come visto anche nella Sezione 3.2.1, si è cercato di riscrivere queste righe di in modo che queste operazioni fossero facilitate, migliorando l'organizzazione del codice.

Si è quindi pensato di considerare i vari tipi di dato coinvolti nelle istruzioni:

- trascurando le questioni sull'effettivo dimensionamento dovute al fatto che `NO_BOXES=1`, `D3SOURCE` e `D3FLUX_FUNC` sono array a due indici, di cui uno fissato nelle operazioni considerate;
- `flux` è un vettore monodimensionale;
- `DAY_PER_SEC` è uno scalare.

Le operazioni considerate sono tutte, di fatto, tra vettori monodimensionali, ma sono coinvolti anche array bidimensionali; perché queste operazioni siano effettuate, negli array bidimensionali si procede per righe (varia il secondo indice), mentre in `flux` si procede in colonna (varia il primo e unico indice). La procedura potrebbe risultare, in qualche modo,

contorta da tradurre in codice macchina per il compilatore, che deve agire su due indici diversi: ciò potrebbe comportare la generazione di un codice macchina poco performante. Si è pertanto provato a dichiarare con più chiarezza l'operazione che deve essere eseguita, e forzarne l'esecuzione lungo un indice di colonna unico e ben definito.

Per farlo si è demandato l'aggiornamento di `D3SOURCE` e `D3FLUX_FUNC` a delle *subroutine ausiliarie*, nelle quali l'operazione risultasse scritta in modo più chiaro. Si è quindi modificato il file `src/BFM/General/FluxFunctions.h90`, contenente il codice di `flux_vector`, aggiungendo le seguenti subroutine:

```
24 !=========-=-=========-=-==========-=-========
25 subroutine D3SOURCE_update(o_vec,d_vec,flux_vec)
26 !=========-=-=========-=-==========-=-========
27   use constants, only:  DAY_PER_SEC
28   real(RLEN), dimension(:) :: o_vec,d_vec,flux_vec
29
30   o_vec=o_vec-flux_vec*DAY_PER_SEC
31   d_vec=d_vec+flux_vec*DAY_PER_SEC
32
33 endsubroutine D3SOURCE_update
34 !=========-=-=========-=-==========-=-========


56 !=========-=-=========-=-==========-=-========
57 subroutine D3FLUX_FUNC_update(vec,index,flux_vec)
58 !=========-=-=========-=-==========-=-========
59   real(RLEN), dimension(:) :: vec,flux_vec
60   integer::index
61
62     vec(:)=vec(:)+SIGN(1,index)*flux_vec(:)
63
64 endsubroutine D3FLUX_FUNC_update
65 !=========-=-=========-=-==========-=-========
```

le quali sono state sostituite nel codice in Figura 4.3 nel seguente modo:

```
106 if ( origin /= destination ) then
107   select case ( iiSub )
108     case (iiPel)
109
110       ! NEW
111       call D3SOURCE_update(D3SOURCE(origin,:),D3SOURCE(destination,:),flux
              (:))


124       ! NEW ALTERNATE
125       if( allocated( D3FLUX_MATRIX ) .AND. &
126            allocated( D3FLUX_MATRIX(origin,destination)%p ) ) then
127          do j=1,SIZE(D3FLUX_MATRIX(origin,destination)%p)
128            call D3FLUX_FUNC_update(D3FLUX_FUNC(ABS(D3FLUX_MATRIX(origin,
                   destination)%p(j)),:), &
129              D3FLUX_MATRIX(origin,destination)%p(j), flux(:))
130          end do
131       endif
```

Riorganizzando così il codice, le istruzioni dentro le subroutine assumono una forma più semplice di quella precedente: una volta passata la "fetta" dell'array a due indici necessaria, le operazioni si riducono a banali somme di vettori monodimensionali.

Si è dunque testata questa nuova versione con subroutine ausiliarie, e ottenuto il seguente time to solution:

$$\text{time to solution}_{\text{ancillary subroutines}} = 2.46 \text{ s}$$

che implica uno speed up di 1.07x. Le Figure 4.4 e 4.5 riportano i risultati del profiling. Si puó notare come `flux_vector` sia ora meno front-end bound di prima ed abbia cessato di essere segnalata come problematica in questo senso da VTune. Il risultato ottenuto, anche se modesto, avvalora la diagnosi: con un codice più intelligibile, la fase di front-end è facilitata per il processore.

È bene rimarcare che il miglioramento delle prestazioni è dovuto al passaggio da istruzioni come questa:

```
D3SOURCE(origin,:) = D3SOURCE(origin,:) - &
    flux(:)*DAY_PER_SEC
```

a istruzioni più snelle come questa:

```
o_vec=o_vec-flux_vec*DAY_PER_SEC
```

e non semplicemente all'"impacchettamento" in subroutine di parti di codice front-end bound. Per mostrare questo fatto, è stata fatta una prova ricreando le subroutine ausiliarie semplicemente incollandovi dentro il codice così com'era in `flux_vector`, al netto di piccole modifiche dovute al passaggio di argomenti:

```
!-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
subroutine D3SOURCE_update(orig,dest,flux_vec)
!-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
  use constants, only:  DAY_PER_SEC
  real(RLEN), dimension(:) :: flux_vec
  integer                  :: orig, dest

  D3SOURCE(orig,:) = D3SOURCE(orig,:) - &
      flux_vec(:)*DAY_PER_SEC
  D3SOURCE(dest,:) = D3SOURCE(dest,:) + &
      flux_vec(:)*DAY_PER_SEC
endsubroutine D3SOURCE_update
!-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

!-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
subroutine D3FLUX_FUNC_update(i,orig,dest,flux_vec)
!-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
  real(RLEN), dimension(:) :: flux_vec
  integer::i,orig,dest

  D3FLUX_FUNC( ABS(D3FLUX_MATRIX(orig,dest)%p(i)), : ) =       &
       D3FLUX_FUNC( ABS(D3FLUX_MATRIX(orig,dest)%p(i)), : ) + &
       (SIGN( 1, D3FLUX_MATRIX(orig,dest)%p(i) ) * flux_vec(:) )

endsubroutine D3FLUX_FUNC_update
!-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
```

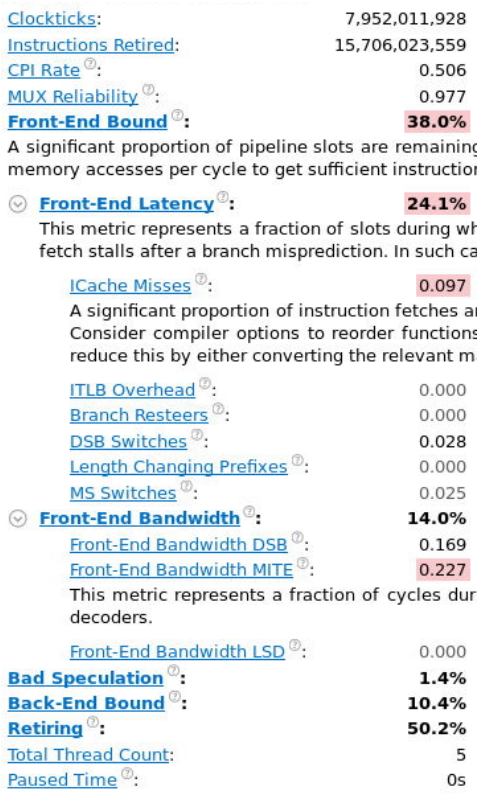


Figura 4.4. Summary del profiling di BFM STANDALONE - con subroutine ausiliarie

Function / Call Stack	Clockticks	Instructions Retired	CPI Rate	Front-End Bound	Bad Speculation	Back-E... Bound	Retiring
flux_vector	1,228,001,842	2,364,003,546	0.519	9.4%	19.3%	27.9%	43.4%
_int_free	510,000,765	822,001,233	0.620	34.5%	21.6%	4.0%	39.9%
mesozoodynamics	504,000,756	1,132,001,698	0.445	48.0%	0.0%	4.0%	52.4%
microzoodynamics	474,000,711	882,001,323	0.537	34.8%	10.4%	14.1%	40.6%
phytodynamics	442,000,663	776,001,164	0.570	52.3%	10.0%	2.9%	34.8%
for_allocate	416,000,624	1,028,001,542	0.405	43.6%	0.0%	2.2%	59.5%
malloc	406,000,609	986,001,479	0.412	28.4%	0.0%	7.9%	67.7%
_int_malloc	382,000,573	1,072,001,608	0.356	49.0%	0.0%	0.0%	86.4%
d3source_update	344,000,516	912,001,368	0.377	3.2%	0.0%	63.2%	83.1%
correct_flux_output	316,000,474	734,001,101	0.431	8.7%	0.0%	37.3%	54.0%

Figura 4.5. Bottom up del profiling di BFM STANDALONE - con subroutine ausiliarie

Ricompilando e lanciando BFM con queste subroutine, non si osserva alcun guadagno sul time to solution:

$$\text{time to solution}_{\text{non-optimized ancillary subroutines}} = 2.66 \text{ s}$$

e anche il profiling (Figura 4.6) non mostra miglioramenti nelle performance di `flux_vector`, che rimane front-end bound; anzi, le subroutine ausiliarie sono evidenziate come back-end bound e affette da bad speculation. Quest'ultimo test è dunque un'ulteriore prova a

Function / Call Stack	Clockticks	Instructions Retired	CPI Rate	Front-End Bound	Bad Speculation	Back-E... Bound	Retiring
▷flux_vector	764,001,146	1,702,002,553	0.449	27.4%	0.0%	15.8%	68.4%
▷_int_free	606,000,909	1,182,001,773	0.513	34.5%	9.1%	8.3%	48.1%
▷_int_malloc	570,000,855	1,358,002,037	0.420	41.5%	6.8%	0.0%	58.9%
▷for_allocate	560,000,840	1,258,001,887	0.445	53.0%	0.0%	0.0%	56.0%
▷malloc	544,000,816	1,224,001,836	0.444	41.5%	14.2%	0.0%	51.6%
▷mesozoodynamics	500,000,750	1,084,001,626	0.461	42.9%	0.0%	10.9%	51.7%
▷phytodynamics	484,000,726	872,001,308	0.555	51.1%	12.5%	1.1%	35.2%
▷d3source_update	480,000,720	1,236,001,854	0.388	13.8%	16.0%	8.3%	61.9%
▷d3flux_func_update	468,000,702	1,116,001,674	0.419	10.6%	0.0%	35.4%	58.8%
▷microzoodynamics	360,000,540	876,001,314	0.411	38.2%	0.0%	14.4%	58.1%

Figura 4.6. Statistiche sulle istruzioni di `flux_vector` per BFM STANDALONE - con subroutine ausiliarie non ottimizzate

favore del fatto che convenga creare un codice più pulito effettuando operazioni con array della stessa dimensione, pur scontando il passaggio di una riga di array bidimensionale come argomento.

Come affermato all'inizio di questa sezione, questa miglioria cessa di funzionare se si rimuove la flag `-heap-arrays`: i test eseguiti sembrano suggerire, anzi, che in quel caso le subroutine ausiliarie arrivino a peggiorare le performance. Effettivamente, ricordando i risultati in Figura 3.4, compilando senza `-heap-arrays` e con `-O1` la subroutine `flux_vector` riduce il suo parametro percentuale `Front-End Bound` al punto da non essere più evidenziata da VTune: in queste condizioni non necessita di alcuna miglioria e aggiungervi chiamate a subroutine, in assenza di inlining, introduce degli overhead. Nondimeno è stato interessante vedere come inserita nel contesto visto ora, in cui è fatto largo uso dell'allocazione dinamica della memoria, essa diventi front-end bound e possa essere ottimizzabile. In questo caso è conveniente spendere un overhead dovuto alla chiamata a subroutine per migliorare l'organizzazione del codice.

4.2 Rimozione del controllo dell'allocazione all'interno di `flux_vector`

Pur avendo già apportato i piccoli miglioramenti a `flux_vector` esposti nella sezione precedente, sono state nuovamente prese in esame le statistiche relative a ciascuna sua riga

di codice per vedere quali sono ora le più costose. In Figura 4.7 è riportato il risultato: il controllo dell'allocazione di `D3FLUX_MATRIX(origin,destination)%p` risulta essere affetto da bad speculation.

So. Li.	Source	Clockticks	Instructions Retired	CPI Rate	Fro... Bou..	Bad Specula...	Back-End Bound
114	! D3SOURCE(origin,:) = D3SOURCE(origin,:) - &						
115	! flux(:)*DAY_PER_SEC						
116	! D3SOURCE(destination,:) = D3SOURCE(destination,:) + &						
117	! flux(:)*DAY_PER_SEC						
118							
119							
120	! NEW						
121	! if (allocated(D3FLUX_MATRIX)) &						
122	! call D3FLUX_FUNC_update(D3FLUX_FUNC,D3FLUX_MATRIX(origin,destination)%p,fl						
123							
124	! NEW ALTERNATE						
125	if(allocated(D3FLUX_MATRIX) .AND. &	68,000,102	142,000,213	0.479	0.0%	16.2%	27.2%
126	allocated(D3FLUX_MATRIX(origin,destination)%p)) then	236,000,354	520,000,780	0.454	9.3%	28.0%	6.8%
127	do j=1,SIZE(D3FLUX_MATRIX(origin,destination)%p)	44,000,066	58,000,087	0.759	12.5%	75.0%	0.0%
128	call D3FLUX_FUNC_update(D3FLUX_FUNC(ABS(D3FLUX_MATRIX(origin,dest	194,000,291	240,000,360	0.808	2.8%	0.0%	57.5%
129	D3FLUX_MATRIX(origin,destination)%p(j), flux(:))						
130	end do	22,000,033	32,000,048	0.688	25.0%	0.0%	50.0%
131	endif						

Figura 4.7. Statistiche sulle istruzioni di `flux_vector` per BFM STANDALONE - con subroutine ausiliarie

Quando possibile, è preferibile evitare l'utlizzo di `if` all'interno del proprio codice, perché costringono all'utilizzo della branch prediction (introdotta brevemente nella Sezione 2.4), la quale può non andare sempre a buon fine e portare alla distruzione di parti della pipeline contenenti istruzioni decodificate che non devono essere eseguite. In questo caso non è chiaro dalla statistica se la bad speculation sia dovuta al fatto che il processore non riesce a prevedere correttamente l'esito del controllo, o se effettuare il controllo stesso è sorgente di bad speculation. In ogni caso, si è pensato di provare a rimuovere questo controllo.

La prima cosa che ci si è chiesti è perché questo controllo sia necessario. La risposta è che in BFM il dataset con cui si lavora impone la compilazione di una versione apposita del programma. A seconda di come si decide di popolare l'ecosistema marino, quando si compila nella cartella `build` usando il comando:

```
./bfm_configure -gcd STANDALONE_PELAGIC
```

l'opzione `-g` specifica un passo preliminare di rigenerazione di alcune parti del codice sorgente. La generazione di queste parti di codice sorgente avviene tramite degli script in Perl contenuti nella directory `build/scripts/bin`. Tra i file modificati dallo script vi sono, come è ragionevole aspettarsi, anche quelli che definiscono l'occupazione della memoria di BFM, che includono i già visti `ModuleMem.F90` e `AllocateMem.F90`. Non vengono invece modificate le subroutine di dinamica (`PhytoDynamics` e simili), e tantomeno `flux_vector`, che da esse viene chiamata. Alla chiamata di `flux_vector`, queste subroutine le passano gli argomenti `origin` e `destination`, che possono portare ad un `D3FLUX_MATRIX(origin,destination)%p` allocato o meno. Queste subroutine sono pensate per essere generali e indipendenti dal dataset, ma il prezzo da pagare è dover inserire il controllo dell'allocazione

di D3FLUX_MATRIX(origin,destination)%p perché, in caso di esito positivo del controllo, flux_vector deve fare uso di p per aggiornare D3FLUX_FUNC (si veda, a questo proposito, il codice riportato in Figura 4.7).

Un'idea per rimuovere il controllo dell'allocazione è fare chiamate a subroutine diverse a seconda che D3FLUX_MATRIX(origin,destination)%p sia allocata o meno. Nello specifico, si è modificato il codice per lavorare nel modo seguente: flux_vector è stata privata del controllo dell'allocazione e tutte le sue chiamate sono state mantenute nei casi in cui vengano passati degli argomenti origin e destination che portano ad un p allocato; nei casi in cui p non risulta allocato, viene chiamata una nuova subroutine no_flux_vector, identica a flux_vector tranne che nelle righe comprese tra 125 e 130 riportate in Figura 4.7

Il primo passo è stato quindi creare no_flux_vector a partire da flux_vector, inserendola nel file src/BFM/General/FluxFunctions.h90. Successivamente il problema é stato capire dove effettivamente no_flux_vector andasse sostituita a flux_vector. Si è quindi contrassegnata ogni chiamata di flux_vector con un output sul terminale contenente il nome del file del sorgente e la riga di codice in cui flux_vector viene chiamata. Per farlo è stato scritto un semplice programma in Perl che esegue questo compito su tutti i sorgenti in src/BFM/Pel in cui sono contenute chiamate a flux_vector, nello specifico: MesoZoo.F90, MicroZoo.F90, PelBac.F90, PelChem.F90, Phyto.F90. Il codice del programma è riportato nel Listing 4.1. Bisogna osservare che il programma contras-

Listing 4.1. Script *Perl* per l'inserimento nei sorgenti di segnalazioni ad ogni chiamata di flux_vector

```
#!/usr/bin/perl -w
require 5.004;

open(OLD, " < $ARGV[0]")          or die "can't open $ARGV[0]: $!";
open(NEW, " > $ARGV[1]")          or die "can't open $ARGV[1]: $!";

$delta=0;

while ($line=<OLD>) {
    chomp($line);
    if ($line =~ m/(\s*)call\sflux_vector/) {
      $delta++;
      print NEW $1, "print \*\, ", "\"\'$line\'", " at line ", $.+$delta, "
          of $ARGV[0]\"\n";
    }
    if ($line =~ m/(\s*)call\sfixed_quota_flux_vector/) {
      $delta++;
      print NEW $1, "print \*\, ", "\"\'$line\'", " at line ", $.+$delta, "
          of $ARGV[0]\"\n";
    }
    print NEW $line, "\n"            or die "can't write $ARGV[1]: $!";
}

close(OLD)                        or die "can't close $ARGV[0]: $!";
close(NEW)                        or die "can't close $ARGV[1]: $!";
```

segna anche le chiamate alla subroutine `fixed_quota_flux_vector`: questa subroutine chiama `flux_vector` al suo interno, quindi anche per essa è stato necessario distinguere i casi in cui `D3FLUX_MATRIX(origin,destination)%p` risulta allocata o meno; per i casi in cui non lo è, analogamente a quanto fatto prima, è stata creato una subroutine `fixed_quota_no_flux_vector` che chiama `no_flux_vector`.

Con questa modifica al codice, ogni chiamata di `flux_vector` è segnalata durante l'esecuzione. Successivamente alla segnalazione di ciascuna chiamata, si è fatto sì che venisse anche specificato se gli argomenti `origin` e `destination` che venivano passati portavano ad una `D3FLUX_MATRIX(origin,destination)%p` allocata o meno. A questo scopo è bastato modificare `flux_vector` nel modo seguente:

```
125 if( allocated( D3FLUX_MATRIX ) .AND. &
126     allocated( D3FLUX_MATRIX(origin,destination)%p ) ) then
127   do j=1,SIZE(D3FLUX_MATRIX(origin,destination)%p)
128     call D3FLUX_FUNC_update(D3FLUX_FUNC(ABS(D3FLUX_MATRIX(origin,
          destination)%p(j)),:), &
129       D3FLUX_MATRIX(origin,destination)%p(j), flux(:))
130   end do
131 else if (.NOT.(allocated( D3FLUX_MATRIX ))) then
132   print *, "D3FLUX_MATRIX not allocated"
133 else if (.NOT.allocated( D3FLUX_MATRIX(origin,destination)%p)) then
134    print *,"D3FLUX_MATRIX element not allocated"
135 endif
```

Apportata anche questa modifica, dopo un'esecuzione di BFM è stato ottenuto un log di tutte le chiamate a `flux_vector`. Siccome le chiamate alle subroutine di dinamica (`PhytoDynamics` e simili) vengono iterate, in modo da fare evolvere il sistema, è stata inserita un'istruzione `stop` in fondo al file `src/BFM/Pel/Ecology.F90`, così da da non avere un log lunghissimo pieno di informazioni ripetute. Si riporta di seguito, come esempio, uno stralcio del log ottenuto dopo un lancio di BFM STANDALONE:

```
'    call fixed_quota_flux_vector(check_fixed_quota, iiPel, ppzooc, &' at
    line 278 of MesoZoo.F90
'    call fixed_quota_flux_vector(check_fixed_quota, iiPel, ppzoon, &' at
    line 281 of MesoZoo.F90
D3FLUX_MATRIX element not allocated
```

Stando alle indicazioni riportate, la riga 278 di `MesoZoo.F90` (nella versione modificata dallo script *Perl* comprensiva delle istruzioni per la stampa su terminale) contiene una chiamata a `fixed_quota_flux_vector` in cui `origin` e `destination` sono tali che `D3FLUX_MATRIX(origin,destination)%p` sia allocato; alla riga 281 accade invece il contrario, come testimonia la stampa di `D3FLUX_MATRIX element not allocated`, quindi qui è stata operata una sostituzione con `fixed_quota_no_flux_vector`. Seguendo questo metodo, sono state operate tutte le sostituzioni necessarie e, al termine del lavoro, sono state rimosse dai file corretti le istruzioni per la stampa su terminale con uno script Perl simile a quello visto precedentemente.

È stato infine eseguito il profiling di BFM senza il controllo dell'allocazione. Il time to solution ottenuto è:

$$\text{time to solution}_{\text{no allocation check}} = 2.38 \text{ s}$$

che corrisponde ad uno speed up di 1.03x. I risultati del profiling sono riportati nelle Figure 4.8 e 4.9. Specialmente nella seconda si può osservare, confrontandola con la Figura 4.5, come la bad speculation in `flux_vector` sia diminuita. È anche vero, tuttavia, che lo speed up ottenuto è piuttosto moderato, e probabilmente insufficiente a giustificare le complicazioni che si introducono nel codice per rimuovere il controllo dell'allocazione in `flux_vector`. Per queste ragioni il risultato ottenuto è stato inserito in questa sezione, e non tra i più importanti ai fini dell'ottimizzazione visti nel capitolo precedente. Esso ha comunque fornito un'informazione preziosa: nel processo di ottimizzazione di `flux_vector`, non è particolarmente rilevante rimuovere il controllo dell'allocazione, bensì conviene concentrarsi su altre sue parti.

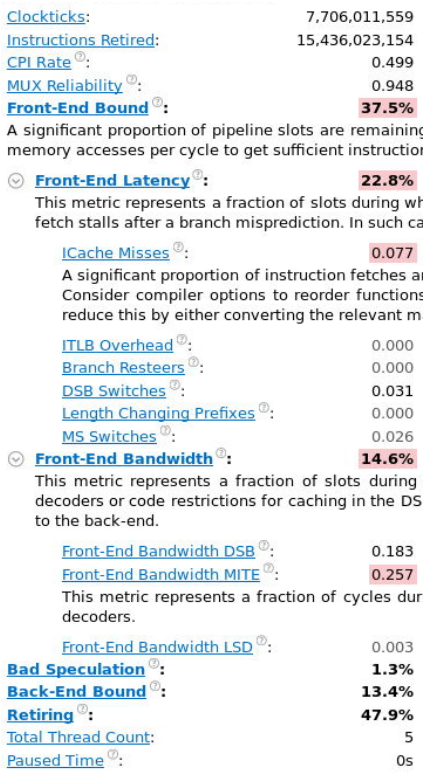


Figura 4.8. Summary del profiling di BFM STANDALONE - senza controllo dell'allocazione

Function / Call Stack	Clockticks	Instructions Retired	CPI Rate	Front-End Bound	Bad Speculation	Back-E... Bound	Retiring
flux_vector	774,001,161	1,344,002,016	0.576	12.1%	6.4%	31.8%	49.7%
_int_free	470,000,705	862,001,293	0.545	38.6%	0.0%	18.1%	43.3%
malloc	468,000,702	920,001,380	0.509	30.6%	14.1%	3.6%	51.7%
microzoodynamics	450,000,675	960,001,440	0.469	45.2%	0.0%	14.4%	47.7%
for_allocate	446,000,669	1,084,001,626	0.411	50.6%	0.0%	10.0%	44.4%
_int_malloc	444,000,666	1,178,001,767	0.377	43.4%	8.7%	0.0%	50.8%
phytodynamics	434,000,651	738,001,107	0.588	39.3%	0.0%	16.4%	52.0%
mesozoodynamics	416,000,624	974,001,461	0.427	47.6%	4.0%	0.0%	54.2%
d3source_update	326,000,489	1,010,001,515	0.323	1.7%	0.0%	42.6%	82.7%
for_deallocate	320,000,480	704,001,056	0.455	41.2%	1.7%	8.9%	48.1%

Figura 4.9. Bottom up del profiling di BFM STANDALONE - senza controllo dell'allocazione

Conclusioni

Il lavoro di profiling e ottimizzazione svolto nel corso del tirocinio ha fatto ottenere diversi risultati, diversificati per l'impatto sulle performance del codice e per il tipo di intervento richiesto. Specialmente quest'ultimo punto non è da sottovalutare, poiché la varietà degli interventi compiuti offre uno spunto di riflessione, ovvero: l'abbondanza e l'eterogeneità di approcci che è possibile adottare nell'ottimizzazione di un software consegue nel fatto che spesso è richiesto tanto lavoro prima di raggiungere risultati soddisfacenti.

È il caso di citare, a questo proposito, tutti gli interventi discussi nel Capitolo 4. L'idea alla base di quegli interventi era ottimizzare la subroutine richiedente più risorse secondo le statistiche del profiler, ovvero `flux_vector`. Si è dunque provato a rendere la subroutine meno front-end bound, inserendo per prima cosa delle *subroutine ausiliarie*: esse sono servite a passare al compilatore istruzioni riscritte in modo ad esso piú chiaro, in modo che generasse un codice in linguaggio macchina che il processore potesse eseguire con migliori prestazioni. Le intenzioni erano facilitare il fetching e il decoding delle istruzioni, individuati come problematici dal profiler.

L'altro intervento su `flux_vector` riportato nel Capitolo 4 è la *rimozione del controllo dell'allocazione*. Questo intervento ha permesso se non altro di testare un espediente per individuare punti di interesse nel codice sorgente: l'espediente è consistito nell'inserire nei sorgenti dei "marker", consistenti nella stampa runtime di output sul terminale, attraverso un programma in Perl. Essi hanno permesso di stabilire gli esiti di istruzioni di controllo, i quali risultavano ostici da prevedere consultando il solo codice sorgente. Conoscendo questi esiti, è stato possibile rimuovere le istruzioni di controllo, poiché in ogni punto del codice era nota la diramazione che sarebbe stata presa. È bene rimarcare che questo metodo è applicabile nei soli casi in cui l'esito delle istruzioni di controllo sia già codificato nei sorgenti: in BFM, le istruzioni per il controllo dell'allocazione all'interno di `flux_vector` servono a tenere conto del fatto che il codice sorgente cambia a seconda del preset con cui si compila; una volta fissato il preset, è stato quindi possibile rimuovere le istruzioni di controllo, perché il loro esito era indipendente dalle condizioni che si creavano runtime.

Questi due interventi fin qui elencati hanno, nel complesso, dato risultati poco significativi dal punto di vista delle performance, ma hanno quantomeno una valenza didattica: le subroutine ausiliarie sono inutili una volta rimossa la flag `-heap-arrays`, ma hanno permesso di mostrare come, in un contesto molto front-end bound, risulti conveniente effettuare una chiamata a funzione per passare al compilatore un'istruzione più facilmente traducibile in codice macchina; invece la rimozione del controllo dell'allocazione in

`flux_vector` non ha dato risultati abbastanza soddisfacenti da giustificare certe complicazioni nel codice, ma ha permesso di testare un metodo che, quando applicabile, potrebbe essere utile a velocizzare programmi le cui prestazioni sono minate dalla presenza di troppe istruzioni di controllo.

I risultati più importanti per le performance, riportati nel Capitolo 3, sono stati però ottenuti abbandonando l'idea di dover ottimizzare il codice di `flux_vector` per il solo motivo che era individuata come la subroutine più costosa dal profiler. Si è optato piuttosto per una visione più olistica del lavoro di ottimizzazione, e deciso che fosse meglio ripartire dalle *flag di compilazione*, con interventi meno invasivi, per poi dedicarsi solo successivamente al refactoring del codice nel dettaglio. La rimozione della flag `-heap-arrays` è così risultata l'intervento più importante dal punto di vista delle prestazioni, ma probabilmente il meno interessante da discutere. Rimane il rammarico per non essere riusciti a risalire, nel corso del tirocinio, alle effettive ragioni per cui quella flag sia stata inserita.

Già più interessante è il fatto che BFM risulti più veloce utilizzando la flag di ottimizzazione `-O1` invece della standard `-O2`. BFM ricade pertanto nella classe di software per i quali conviene disattivare, o quantomeno limitare, inlining e unrolling, affinché il programma risultante dalla compilazione non abbia dimensioni eccessive. Come visto più volte nel corso del documento, è in genere conveniente utilizzare inlining e unrolling per evitare le diramazioni introdotte da cicli e chiamate a funzione; ma talvolta queste ottimizzazioni possono aumentare troppo le dimensioni del codice e renderlo front-end bound. In questi casi, è meglio utilizzare una flag corrispondente ad un'ottimizzazione meno intensa.

L'ultima modifica rilevante dal punto di vista delle prestazioni è quella apportata alle operazioni tra array aventi una dimensione degenere. Si è trovato che in questi casi è meglio esplicitare l'indice corrispondente alla dimensione degenere, così da evitare l'apertura inutile di un ciclo che si ha usando invece i due punti ':'. È bene rimarcare che questa misura non è generale, ma valida per i soli casi in cui la variabile `NO_BOXES` è uguale a 1. Si parla comunque di casi che trovano applicazione all'interno di OGS, nei quali quindi vale la pena considerare di compilare una versione speciale del programma.

Il lavoro di ottimizzazione si è concluso con ques'ultimo intervento. L'incidenza sulle prestazioni di ogni passo del lavoro è riassunta nell'istogramma in Figura 4.10, in cui sono rappresentati i time to solution in percentuale rispetto a quello del codice di partenza. Lo speed up complessivo ottenuto sul codice accoppiato OGSTM-BFM è di 2.0x.

Potenzialmente, si potrebbe però ancora procedere nel lavoro, poiché con le ottimizzazioni apportate si ottiene un programma che inizia ad essere back-end bound (specialmente nell'accoppiamento OGSTM-BFM). Il profiler VTune consiglia di attivare l'autovettorizzazione per gestire il problema, ma essa viene disattivata una volta che si passa alla flag `-O1`. Un'idea per proseguire ulteriormente l'ottimizzazione del codice è trovare e inserire manualmente le flag di compilazione ottimali per l'autovettorizzazione, lavoro per cui non è stato trovato il tempo nel corso del tirocinio.

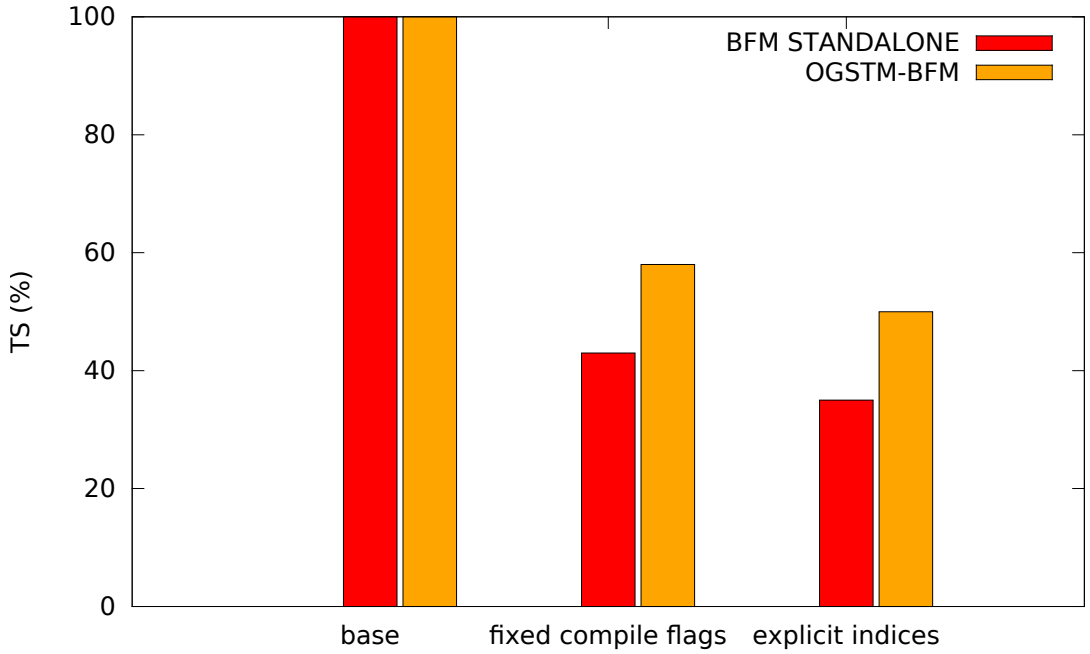


Figura 4.10. Time to solution percentuali rispetto al time to solution del codice di partenza

Svolgimento del tirocinio

L'esperienza di tirocinio è stata valutata in modo positivo. L'attività di tirocinio è stata svolta presso la sedi dell'Istituto Nazionale di Oceanografia e Geofisica Sperimentale (OGS) a Trieste e del CINECA a Bologna, in 3 periodi separati:

- il primo dal 23 al 27 maggio 2016 presso la sede di OGS;
- il secondo dal 6 al 17 giugno 2016 presso la sede del CINECA (25th Summer School of Parallel Computing);
- il terzo dal 27 giugno al 1 luglio 2016 presso la sede del CINECA.

L'intera attività di stage è stata seguita da vicino dai tutor aziendali: il Dott. Paolo Lazzari e il Dott. Eric Pascolo presso la sede di OGS; il Dott. Eric Pascolo e il Dott. Fabio Affinito presso la sede del CINECA.

Obiettivi e Modalità del tirocinio

- **Apprendimento dei rudimenti del calcolo parallelo con MPI e OpenMP (25th Summer School of Parallel Computing of CINECA)**
- **Apprendimento di varie tecniche di profiling e ottimizzazione del codice**
- **Imparare a gestire e lavorare su un codice di produzione scientifica**

Bibliografia

[1] B. Barney. *Introduction to Parallel Computing*. 2016. URL: https://computing.llnl.gov/tutorials/parallel_comp/.

[2] J. Barretta, W. Ebenöh e P. Ruardij. «The European Regional Seas Ecosystem Model, a complex marine ecosystem model». In: *Journal of Sea Research* (1995).

[3] G. Cossarini, P. Lazzari e C. Solidoro. «Spatiotemporal variability of alkalinity in the Mediterranean Sea». In: *Biogeosciences* (2015).

[4] Intel Developers. *Determining Root Cause of Segmentation Faults SIGSEGV or SIGBUS errors*. 2011. URL: https://software.intel.com/en-us/articles/determining-root-cause-of-sigsegv-or-sigbus-errors.

[5] Intel Developers. *DSB Switches*. 2016. URL: https://software.intel.com/en-us/node/544418.

[6] Intel Developers. *ICache Misses*. 2016. URL: https://software.intel.com/en-us/node/596831.

[7] Intel Developers. *Performance Analysis Setup*. 2016. URL: https://software.intel.com/en-us/node/544018.

[8] Intel Developers. *Using Automatic Vectorization*. 2016. URL: https://software.intel.com/en-us/node/522572.

[9] A. Emerson e G. Erbacci. *Introduction to HPC Architectures*. 25th Summer School of Parallel Computing of CINECA. 2016.

[10] P. Konsor. *MITE Micro-ops to IDQ*. 2012. URL: https://software.intel.com/en-us/forums/intel-performance-bottleneck-analyzer/topic/308522.

[11] P. Lazzari. «Integration of biogeochemical and transport models in Mediterranean Sea». PhD Thesis. Università di Trieste: Facoltà di Scienze Matematiche, Fisiche e Naturali, 2008.

[12] P. Lazzari et al. «Pre-operational short-term forecasts for Mediterranean Sea biogeochemistry». In: *Ocean Science* (2010).

[13] G. Madec e the NEMO team. *NEMO ocean engine*. 2016. URL: http://www.nemo-ocean.eu/About-NEMO/Reference-manuals.

[14] G. Madec et al. *OPA 8.1 Ocean General Circulation Reference Manual*. 1998. URL: http://poc.obs-mip.fr/DOCUMENTS/opa.pdf.

[15] G. L. Mellor. *Users Guide for a Three-Dimensional, Primitive Equation, Numerical Ocean Model*. 1998. URL: http://www.pd.infn.it/AOD/immagini_models/POM_manual.pdf.

[16] J. D. Murray. *Mathematical Biology: I. An Introduction, Third edition*. 2002.

[17] A. B. Ryabov e B. Blasius. «Population growth and persistence in a heterogeneous environment: the role of diffusion and advection». In: *Mathematical Modeling of Natural Phenomena* (2008).

[18] G. Supalov et al. *Optimizing HPC Applications with Intel Cluster Tools*. 2014.

[19] L. Tedesco, M. Vichi e D. N. Thomas. «Process studies on the ecological coupling between sea ice algae and phytoplankton». In: *Ecological Modelling* (2012).

[20] L. Tedesco et al. «An enhanced sea-ice thermodynamic model applied to the Baltic Sea». In: *Boreal Environmental Research* (2008).

[21] G. D. Tilman. *Resource competition and community structure*. 1982.

[22] L. Umlauf, H. Burchard e K. Bolding. *GOTM: Sourcecode and Test Case Documentation*. 2014. URL: http://www.nemo-ocean.eu/About-NEMO/Reference-manuals.

[23] M. Vichi, S. Masina e A. Navarra. «A generalized model of pelagic biogeochemistry for the global ocean ecosystem. Part 2: Numerical Simulations». In: *Journal of Marine Systems* (2006).

[24] M. Vichi, N. Pinardi e S. Masina. «A generalized model of pelagic biogeochemistry for the global ocean ecosystem. Part 1: Theory». In: *Journal of Marine Systems* (2006).

[25] M. Vichi et al. *The Biogeochemical Flux Model (BFM): Equation Description and User Manual. BFM version 5.1*. BFM Report Series 1. Ver. 1.1. Bologna, Italy, ago. 2015. URL: http://www.bfm-community.eu.

[26] V. Volterra. «Fluctuations in the Abundance of a Species considered Mathematically». In: *Nature* (1926).

[27] Autori di Wikipedia. *Calcolo parallelo*. 2016. URL: https://it.wikipedia.org/wiki/Calcolo_parallelo.

[28] Autori di Wikipedia. *Legge di Moore*. 2016. URL: https://it.wikipedia.org/wiki/Legge_di_Moore.