Università di Pisa
Facoltà di Scienze Matematiche Fisiche e Naturali

Corso di Laurea Specialistica in Tecnologie Informatiche

Tesi di laurea

Caratteristiche della programmazione di applicazioni context-aware e una proposta di modello ad alte prestazioni

Candidato
Daniele Buono

Relatore
Prof. Marco Vanneschi

Controrelatore
Prof. Antonio Cisternino

Anno Accademico 2008/2009

A Serena

Quando lavorate, siete un flauto
che attraverso il suo cuore
trasforma in musica
il mormorio delle ore
(Kahlil Gibran)

Ringraziamenti

Questa tesi rappresenta per me un traguardo molto importante, la fine degli studi ed un primo passo verso quell'indipendenza cercata ormai da molto tempo. Si tratta forse del punto di svolta più grande mai affrontato fino a questo momento: la fine del periodo dello studio e l'entrata nel mondo del lavoro. E probabilmente anche l'ultima occasione per salutare e ringraziare tutte quelle persone che mi hanno accompagnato e sostenuto fino ad ora.

Grazie alla mia famiglia: babbo, mamma e Jessica, che nonostante le opinioni divergenti mi hanno sostenuto come potevano in questo percorso lungo una vita. Grazie anche ai miei nonni, vicini e lontani: Gabriella e Virgilio, Nicola e Valentina. Devo molto a tutti, e senza di voi non sarei diventato quello che sono.

Ringrazio di cuore tutti i ragazzi che ho incontrato in questo percorso universitario, che mi hanno tenuto compagnia a mensa, nello studio e nello svago.

Il mitico "gruppo del laboratorio I", amici di sempre e compagni indimenticabili: Bacai, Profe, Sandro, Gabri, Lotta e Maria Luisa. Una menzione d'onore spetta a Cino, "collega" fin dalle scuole medie e sempre vicino. Le nostre mitiche partite a Risiko (con le mie "armate papali") nei laboratori dell'università rimarranno sicuramente nella storia.

Grazie anche ai nuovi amici incontrati nel percorso (lungo quasi due anni) verso la laurea specialistica. Paolo, l'essere più NERD mai conosciuto che ha preso il laboratorio per una camera da letto, ma che solo insieme al compagno di viaggio e di studio Fabio riesce a dare il meglio di sé. Daniele, con cui ho passato l'ultima estate (domeniche incluse) in lab e mi ha spronato e fatto compagnia per tutto il tempo. Il Lotta, alias ex rasta, che si è trovato in lab dopo un tirocinio scelto quasi per caso, ma che si è dimostrato un amico unico... se non fosse per quella storia dei fisici... E tutti i passanti, più o meno importanti: Marina, Nicola, Maximiliano, Alberto, Alice e Michele.

Senza però dimenticare lo "zoccolo duro" del laboratorio, compagni di lavoro ma anche di panuozzo. Ale, promosso a "sistemista di laboratorio"

e a tuttofare. Marco, nuovo acquisto e con un po' di fortuna compagno di dottorato. Gabriele, il primo tesista su ASSISTANT e ormai pietra fondamentale del laboratorio. Grazie in particolare, a Carlo, ormai in Spagna ma per molto tempo il mio fornitore preferito di consigli. E a Massimiliano, revisore ufficiale dei disegni della tesi e soprattutto ingegnere informatico quasi redento. Un abbraccio a Silvia, vera amica con cui ho condiviso l'ultimo ed interminabile progetto della laurea.

Grazie anche al mitico gruppo degli sfidanti: Giacomo, matematico atipico che mi ha fatto rivalutare l'intera specie; Andrea, alias ingegner Ballini, che ha portato in laboratorio un po' di fiorentino. Sabri, con i suoi programmi che non funzionavano, ma anche i suoi abbracci e la sua tenacia nel terminare il master. Paolo, ex tirocinante entrato ormai a pieno titolo tra gli sfidanti.

Vorrei poi ringraziare in modo speciale ed unico due persone che mi hanno guidato in questi anni, in due aspetti molto differenti ma ugualmente importanti per la mia crescita.

Il Professor Marco Vanneschi, che ho iniziato ad ammirare con le indimenticabili lezioni di Architettura degli Elaboratori, quando spiegava il funzionamento della parte per me più oscura di un computer. Dal tirocinio (scelto con molta paura) è diventato una guida per tutto il percorso di studi.

La seconda persona importantissima è Don Marco. Ex parroco del mio paese, mi ha insegnato a vivere con un vero obiettivo; per me non è stato solo "il prete" ma anche un amico, un padre. Mi ha riavvicinato a Dio e mi ha insegnato a camminare con Gesù. A lui devo molte delle esperienze indimenticabili della mia vita: Gavinana, i campeggi, Loreto, Re. Mi manchi davvero tanto.

Infine ringrazio la mia anima gemella, compagna di mille avventure e aiuto in tutte le difficoltà. Mi dimostri il tuo amore ogni giorno, e so benissimo che quando mi sgridi, o litighi con me, lo fai per migliorarmi. Ma nonostante tutto mi apprezzi ed accetti per quello che sono; mi supporti nelle scelte importanti, mi aiuti e non mi neghi mai una spalla su cui piangere. Ti sei sobbarcata dei miei mille problemi con la famiglia e con l'università. E tutto questo chiedendo in cambio solo di farmi la barba!

I tuoi consigli sono sempre stati preziosissimi. Perché in fondo, diciamolo, SERENA HA SEMPRE RAGIONE!

Ormai siamo una cosa sola, e viviamo nell'attesa di quel momento importante che ci unirà per tutta la vita.

E per chiudere un cerchio iniziato due anni fa, concludo come per la relazione di tirocinio: un grazie a Dio. Lui che mi ha donato i talenti, mi ha permesso di coltivarli e mi ha sempre guidato nel cammino.
Senza chiedere mai niente in cambio.

Indice

Elenco delle figure

Elenco delle tabelle

Elenco dei listati

Capitolo 1

Introduzione

Questa tesi si propone di analizzare un nuovo ed importante ambito di ricerca, che unisce due settori dell'informatica storicamente ben distinti e molto lontani tra loro. Stiamo parlando del **"High Performance Pervasive Computing"**, formalizzato con questo termine in [58] ma già presente nel mondo della ricerca nella forma delle *"Pervasive Grid"*[52, 62].

Il termine stesso riporta ai due settori di ricerca interessati: "Grid" e "Pervasive" Computing. Agli inizi del nuovo secolo i ricercatori dei due i settori hanno iniziato a guardare con interesse le innovazioni introdotte nell'altro. Entrambi trattano infatti applicazioni distribuite tra nodi eterogenei in ambienti di esecuzione dinamici, anche se con ottiche estremamente differenti.

Da una parte troviamo il "Pervasive Computing", nato alla fine degli anni 80 dalla mente visionaria di Mark Weiser[82]. In questo settore si studiano tecnologie hardware e software per la progettazione di applicazioni distribuite su un insieme molto dinamico di dispositivi dotati di capacità computazionali e comunicative (dai comuni PC a cellulari, PDA, wearable devices, etc), che possano aiutare l'utente nei suoi compiti quotidiani. Una caratteristica base di queste applicazioni è la totale integrazione con l'ambiente e con gli utenti stessi, in modo da adattarsi alle richieste e alle necessità di essi.

Dall'altra il "Grid Computing", promosso inizialmente da Foster[46], si propone di utilizzare risorse sparse geograficamente per l'esecuzione di applicazioni computazionalmente pesanti e si è rivelato negli ultimi anni come il paradigma principale per lo sviluppo di applicazioni distribuite su grandi distanze. In questo caso si studiano sempre problemi legati all'eterogeneità e alla dinamicità dei nodi, anche se di entità minore e nell'ottica di applicazioni parallele ad alte prestazioni.

Lo sviluppo della tecnologia e della miniaturizzazione dei componenti ha nel tempo avvicinato i due settori: fino a qualche anno fa i dispositivi embedded ormai di uso comune (orologi digitali, navigatori GPS, cellulari, etc.) erano quasi impensabili.

Quando fu coniato il concetto di "Pervasive Computing" lo scalino considerato insormontabile era quello dell'hardware: riuscire ad ottenere dispositivi compatti, con risorse computazionali adeguate e consumi ridotti era, a quel tempo, impossibile. Con l'evoluzione dell'hardware questo problema è venuto meno, ma lo sviluppo delle tecnologie software non è andato di pari passo: i ricercatori si sono trovati con i dispositivi immaginati da Weiser ma senza gli strumenti adatti per programmarci applicazioni pervasive. A questo punto è emersa l'idea di utilizzare dei tool già sviluppati: quelli creati per "Grid Computing" sembravano un buon punto di partenza[38, 70], in quanto già costruiti con i meccanismi per gestire eterogeneità e dinamicità dei nodi.

Dal punto di vista del "Grid Computing", invece, il fattore scatenante è stata la potenza computazionale fornita dagli attuali dispositivi embedded, allineata ai computer di qualche anno fa. Nasce così il concetto di "Pervasive Grid", ovvero una griglia composta dai nodi tipici di un ambiente pervasivo utilizzati, se necessario, cooperando con o sostituendo i nodi normalmente presenti nelle griglie.

Nell'ottica di applicazioni HPC sono sempre stati considerati nodi come server, workstation ed eventualmente PC. Il trend dei dispositivi embedded, però, sottolinea una particolare evoluzione nell'ultimo periodo: gli smartphone sono dotati di processori molto efficienti,con velocità che superano i 600Mhz e acceleratori grafici dedicati. La stessa ARM ha da poco presentato una nuova architettura per dispositivi portatili ed embedded, multi-core (fino ad 4 core per processore) con frequenze che raggiungeranno i 2Ghz[14].

"Pervasive Grid"

Rispetto ad una normale griglia computazionale, quelle pervasive estremizzano sia i concetti di eterogeneità che di dinamicità. Abbiamo nodi profondamente differenti sia dal punto di vista prestazionale che architetturale che software, e la frequenza con cui i nodi entrano o escono nella griglia aumenta considerevolmente: stiamo infatti parlando di dispositivi per natura mobili e di connessioni di tipo wireless, che non offrono certo le caratteristiche di stabilità di quelle cablate.

Ma il problema più importante che si pone è la tipologia di applicazioni che vogliamo sviluppare su queste griglie. Potremmo considerarle come normali griglie computazionali dove, se risulta conveniente, vengono utilizzati anche i dispositivi mobili per aumentare il grado di parallelismo dell'ap-

plicazione e, di conseguenza, le prestazioni della stessa. Questi dispositivi verrebbero perciò trattati a tutti gli effetti come normali nodi della griglia.

Oppure si potrebbe usare un approccio più alla "pervasive computing", dove le normali applicazioni pervasive utilizzano, quando ne hanno bisogno, il resto della griglia per calcoli intensivi.

Entrambi gli approcci rimangono però troppo legati al settore di provenienza, e non permettono di trarre il massimo beneficio dalla "Pervasive Grid". Ci immaginiamo un nuovo approccio, che amalgami in modo molto più forte le due tecnologie; da questo nasce il termine "**High Performance Pervasive Computing**"[1], che vuole dare la possibilità di esprimere applicazioni *pervasive* con requisiti di *alte prestazioni*.

Un esempio di applicazione

Ci si potrebbe chiedere quale tipo di applicazioni abbia bisogno contemporaneamente di caratteristiche pervasive e di alte prestazioni. Un esempio pratico è emerso all'interno del progetto FIRB In.Sy.Eme[76], focalizzato allo studio di sistemi informativi integrati per la gestione di emergenze sul territorio. Queste si differenziano notevolmente dai tipici casi d'uso di applicazioni pervasive, utilizzate normalmente per realizzare ambienti "intelligenti"[27, 69, 82] sia per i requisiti computazionali che per l'estensione del territorio gestito dall'infrastruttura.

Parliamo di aree grandi decine (se non centinaia) di chilometri all'aperto, ben diverse da quelle contenute di una stanza o di un palazzo. Queste aree sono sotto il controllo costante di sensori di vario tipo, per monitorarle sia in una fase previsionale, che in una di decision-making durante le emergenze.

In generale gestire questo flusso di informazioni ed utilizzarlo per previsioni accurate è una operazione non banale che richiede grandi capacità di calcolo, per eseguire algoritmi di simulazione, data mining, etc. Questo rende necessario l'utilizzo di sistemi paralleli, e di tutte le tecniche e tecnologie tipiche dell'HPC. Allo stesso tempo, però, soprattutto in caso di emergenza, abbiamo a che fare con un grande numero di dispositivi di piccole dimensioni in un'area geografica ristretta. In questi casi, proprio a causa dell'emergenza stessa, le risorse normalmente utilizzate per analizzare i dati potrebbero non essere raggiungibili; nell'ottica di aiutare al meglio gli operatori si rende *necessario* l'utilizzo delle poche risorse accessibili sul luogo, per generare delle previsioni "peggiori" ma che possano comunque dare agli operatori un'idea di cosa potrebbe accadere.

[1] Che spesso abbrevieremo con la sigla HPPC

Gli strumenti per costruire applicazioni HPPC

L'obiettivo principale di questa tesi è di proporre uno strumento per la programmazione di questa tipologia di applicazioni. Essendo un settore relativamente nuovo non si trovano ancora strumenti nati e sviluppati per questo. Abbiamo perciò studiato a fondo i sistemi di sviluppo attualmente presenti sia nell'ambito del "Pervasive Computing" che in quello del "Grid Computing" e, più in generale, del "High Performance Computing".

Da questo studio vedremo come nessuno degli strumenti attualmente esistenti riesca a racchiudere tutte le caratteristiche da noi richieste. Proponiamo perciò un nuovo modello, che basa le sue radici sul progetto ASSIST, sviluppato nel Dipartimento di Informatica dell'Università di Pisa per la realizzazione di applicazioni parallele ad alte prestazioni. A questo modello, che supporta nativamente le griglie computazionali, aggiungiamo la possibilità di definire applicazioni pervasive, supportando le due caratteristiche principali di quest'ultime: *context awareness* e *adattività*. L'estendere ASSIST per gestire queste due problematiche ci ha portato a chiamare il nuovo modello ASSISTANT: ASSIST with Adaptivity and coNText awareness.

1.1 ASSISTANT: un modello per applicazioni pervasive ad alte prestazioni

Il modello di programmazione ASSISTANT non viene trattato per la prima volta in questa tesi: il nostro gruppo di ricerca ci sta lavorando da più di un anno, sempre nell'ottica del progetto FIRB In.Sy.Eme (Integrated Systems for Emergencies). Le prime idee sono state introdotte nella tesi [58]; successivamente sono stati prodotti anche degli articoli ([21, 22, 42]) e alcune prime sperimentazioni per valutare le scelte operate a livello di modello.

Nonostante questo, lo studio su ASSISTANT non è ancora da considerarsi completo. Molti degli aspetti chiave sono stati studiati in modo superficiale, nell'ottica di avere una visione d'insieme da approfondire successivamente.

Con questa tesi abbiamo perciò iniziato ad analizzare in modo più approfondito alcuni dei problemi non ancora studiati. Durante il lavoro sono stati modificati alcuni aspetti sintattici e semantici del modello; per questo motivo la versione riportata qui differisce lievemente da quelle già presentate nei precedenti lavori. In ogni caso neanche questa versione è da considerarsi definitiva: basti pensare che alcuni sviluppi nati anche all'interno della tesi non sono stati riportati, per motivi di tempo e di spazio.

La decisione di sviluppare il nuovo modello a partire da uno strumento per applicazioni parallele nasce principalmente dall'esperienza del nostro gruppo di ricerca in questo settore.

Questo non è però l'unico motivo: lo studio degli strumenti per applicazioni pervasive ha sottolineato come nessuno di questi permettesse lo sviluppo di programmi paralleli. Inserire al loro interno meccanismi per descrivere computazioni di questo tipo avrebbe richiesto profonde modifiche all'intera architettura di questi modelli. Al contrario la programmazione parallela strutturata ci permette di inserire caratteristiche di adattività e context awareness senza dover stravolgere completamente il modello, mantenendo quindi le caratteristiche di programmabilità e performance che la contraddistinguono.

La scelta del modello di programmazione parallela strutturata da utilizzare come base è caduta naturalmente su ASSIST, in quanto studiato da anni all'interno del nostro gruppo di ricerca e molto più flessibile dei modelli a skeleton classici. Il lavoro svolto è però in gran parte indipendente da ASSIST, e può essere studiato anche su altri modelli, come ad esempio quello presentato in [5], che include già alcune forme di adattività.

Gli eventi

Una applicazione adattiva e context aware richiede, per definizione, una certa "coscienza" dell'ambiente in cui si trova. Con il termine "ambiente" nel caso di applicazioni pervasive ci si riferisce non solo all'ambiente di esecuzione (le macchine utilizzate dal programma), ma anche all'ambiente fisico in cui si trova.

La modalità più utilizzata per dotare l'applicazione di questa conoscenza è l'utilizzo di un meccanismo di "eventi", che vengono generati in caso di modifica dell'ambiente e notificati all'applicazione. In questo modo siamo liberi di definire un comportamento "asincrono", dove il programma continua normalmente la sua esecuzione finché non si verifica un particolare evento. La gestione dell'evento può causare delle modifiche nel comportamento dell'applicazione, che diventa così context-aware. Una caratteristica importante di un modello ad eventi di questo tipo è la sua naturale predisposizione all'adattività: basta definire e trattare degli eventi anche sulla variazione delle caratteristiche delle risorse, per rendere l'applicazione adattiva.

Nella nostra proposta gli eventi diventano perciò una delle componenti fondamentali per rendere le applicazioni ad alte prestazioni *pervasive*: un programma continua la sua naturale esecuzione fino all'arrivo di un evento, quando (in base a delle politiche decise dallo sviluppatore) deciderà se e come modificare il suo comportamento e la sua esecuzione. Le applicazioni possono

perciò essere adattive e context-aware allo stesso tempo, in base alla tipologia di eventi trattati.

L'adattività

La strutturazione di un programma parallelo utilizzando le forme di parallelismo note offre molti vantaggi, come avere un modello dei costi, conoscere implementazioni ottimizzate della struttura parallela, poter applicare ottimizzazioni legate all'architettura di esecuzione e scrivere programmi parametrici sul grado di parallelismo. Tutte queste caratteristiche sono importantissime in ambienti di esecuzione dinamici, in quanto permettono al programmatore di descrivere il programma in una forma generica e lasciare al supporto le eventuali ottimizzazioni per le varie architetture.

Possiamo perciò dire che i modelli di programmazione parallela strutturata offrono naturalmente dei meccanismi di adattività trasparenti all'applicazione, tramite la realizzazione di supporti in grado di modificare l'implementazione dell'applicazione per sfruttare al meglio l'architettura di esecuzione corrente. In caso di aggiunta o rimozione di nodi si può cambiare il grado di parallelismo, oppure spostare parti dell'applicazione su nodi differenti e più potenti. Nell'eventualità di un cambio radicale dell'ambiente di esecuzione possiamo persino passare ad una nuova implementazione della forma di parallelismo per migliorare le prestazioni.

Nel caso degli ambienti di griglia queste tecniche sono molto importanti, e soprattutto sufficienti, per sfruttare al meglio i nodi disponibili. Di studi in letteratura su questi concetti ce ne sono moltissimi, e dimostrano come sia possibile realizzare moduli "autonomici", in grado di modificare automaticamente il proprio comportamento per rispettare determinati "contratti" forniti dal programmatore[4, 79].

Al contrario siamo fermamente convinti che, per ambienti molto più eterogenei e dinamici come quelli tipici del pervasive computing, queste tecnologie non siano più sufficienti. Il forte divario tra le varie risorse hardware presenti rende molto meno praticabile un approccio "trasparente" al programmatore, in cui tutta la gestione dell'adattività viene demandata ad un "supporto intelligente".

Spostare una computazione sviluppata per ambienti ad alte prestazioni su un insieme di dispositivi mobili è in realtà una operazione molto complicata: non si tratta solo di questioni puramente tecnologiche, come poter disporre di una versione compilata (ed ottimizzata) per entrambe le architetture (che magari utilizzano supporti e meccanismi di comunicazione differenti), di spostare i dati della computazione verso i nuovi nodi o di gestire dei formati di

dato differenti[2]. Per risolvere questi problemi ci basta sfruttare la conoscenza fornita dalle forme di parallelismo, e sono già stati tutti trattati con successo sulle griglie computazionali.

Nel nostro caso le differenze sulle risorse sono così marcate che possono influire sulla corretta esecuzione dell'applicazione stessa. Le risorse hardware disponibili sui PDA non saranno sicuramente sufficienti a garantire l'esecuzione di un programma pensato per un cluster: anche ammettendo che la memoria di questi dispositivi possa contenere i dati della computazione (un server può gestire qualche gigabyte di dati, il pda poche centinaia di megabyte) i tempi esecuzione sui dispositivi mobili saranno spesso talmente alti da risultare comunque inutili. C'è bisogno di un cambiamento più radicale nell'applicazione, di modifiche sostanziali agli algoritmi eseguiti, che permettano un comportamento accettabile anche su dispositivi con caratteristiche computazionali sensibilmente diverse.

Anche in questo caso le forme di parallelismo ci possono aiutare, con la loro caratteristica di *composizionalità* e la possibilità di definire le applicazioni tramite grafi generici di entità parallele cooperanti (che per comodità nella trattazione chiameremo "moduli").

In questo caso le applicazioni ereditano alcune caratteristiche tipiche della programmazione a componenti, soprattutto la completa separazione tra i differenti moduli che la compongono, e possiamo riutilizzare le tecniche studiate in quel settore per definire un concetto di "compatibilità", che ci permette di "sostituire" delle parti senza modificare l'intera applicazione.

In questo modo possiamo definire più "versioni" di ogni singola entità parallela, adatte a differenti ambienti di esecuzione. Questo concetto è differente da quello già spiegato precedentemente di "implementazione" di una forma parallela. In quel caso, infatti, la semantica del modulo non variava tra le due implementazioni: si trattava cioè di semplici "ottimizzazioni" del codice parallelo. Al contrario, quello che vogliamo definire ora sono differenti versioni, con semantiche diverse ma "compatibili", che possano essere scambiate tra loro senza modificare il comportamento globale dell'applicazione.

Per chiarire il concetto riportiamo un esempio pratico, tratto dal calcolo numerico. Per la risoluzione di sistemi lineari esistono molti algoritmi differenti. Alcuni sono di tipo diretto, altri di tipo iterativo. Assumiamo di avere un modulo che risolve un sistema lineare. Questo può essere implementato con differenti algoritmi, ma il suo risultato sarà sempre la soluzione del sistema. Certo, potrebbe non essere *esattamente* uguale, in quanto ogni al-

[2]L'esempio più importante in questo ambito è la differenza, tra i formati "big endian" e "little endian" ancora presente nel mondo dei calcolatori.

goritmo ha una sua stabilità ed una sua precisione. Due moduli con lo stesso algoritmo, ma con grado di parallelismo differente, forniranno un risultato identico ma con tempi di esecuzione differenti. Due moduli con algoritmi diversi forniranno, invece, un risultato simile ma non necessariamente identico; non hanno perciò la stessa semantica, ma sono compatibili, in quanto producono un risultato concettualmente identico.

La nostra idea è quella di poter descrivere applicazioni dotate di differenti "versioni" di ogni singola parte parallela, adatte ad essere eseguite su diverse tipologie di dispositivi. Questo approccio richiede però l'intervento del programmatore, in quanto il supporto non è in grado di trovare automaticamente le differenti versioni.

L'approccio all'adattività non è più completamente trasparente, e richiede l'intervento dello sviluppatore dell'applicazione. Si tratta comunque dell'approccio utilizzato nell'ambito del "pervasive computing", dove le applicazioni adattive vengono da sempre definite con l'ausilio del programmatore [61, 65].

Riprendendo ora l'esempio di prima, possiamo permetterci di spostare un modulo da un cluster ad un pda, in quanto il programmatore avrà fornito due differenti versioni: una per il cluster, l'altra per il pda. Chiaramente le due versioni devono essere profondamente differenti: ci immaginiamo che la versione per pda produca risultati meno accurati o addirittura stimati, che però in molti casi sono molto più utili rispetto a non avere assolutamente un risultato, o averlo in un tempo troppo elevato.

Il concetto di "versioni" diverse per ogni parte dell'applicazione non è comunque in contrasto con le tecniche di adattività sull'implementazione della forma parallela. In generale queste due possono, e per noi devono, essere presenti contemporaneamente nel modello; in questo modo il supporto del modello ed il programmatore stesso cooperano per realizzare applicazioni studiate per differenti tipologie di architetture ed ottimizzate per i singoli ambienti di esecuzione.

La context awareness

A questo punto abbiamo già introdotto tutte le caratteristiche necessarie per definire una applicazione context-aware: abbiamo un modello di eventi, che permette di modificare il comportamento dell'applicazione in base alla loro ricezione. Se questo non bastasse, possiamo anche definire differenti versioni di un modulo, anche profondamente diverse tra loro. Questi due strumenti permettono di modificare in modo molto flessibile l'intera applicazione o delle

sue parti per renderla adatta all'ambiente (inteso nel senso più ampio del termine).

I modelli per applicazioni pervasive forniscono, normalmente, tecniche per ragionare sugli eventi ricevuti al fine di dedurre nuove informazioni. La nostra proposta è di dedicare parti dell'applicazione alla generazione di queste nuove informazioni, tramite la creazione di nuovi eventi in base ad un lavoro su quelli ricevuti. In questo modo si possono modellare anche algoritmi di inferenza o data mining molto complicati, magari in versione parallela.

1.2 Struttura della tesi

L'obiettivo principale della tesi è stato quello di raccogliere tutto il lavoro svolto fino a questo momento sull'ambiente di sviluppo per descrivere ASSISTANT in modo completo e, al tempo stesso, correggere eventuali incoerenze nate dallo studio sommario di alcuni aspetti del modello.

La tesi è naturalmente suddivisa in due parti: una di presentazione dello stato dell'arte, ed una sul nuovo modello di programmazione ASSISTANT.

Nella prima parte illustriamo le problematiche del "High Performance Pervasive Computing" e dei suoi due parenti stretti, cercando di raccogliere gli attuali sviluppi nel campo dei modelli di programmazione sia per il "Pervasive Computing che per l'"High Performance Computing". Nella seconda presentiamo invece il nuovo modello. Cerchiamo però di mantenere la trattazione il più possibile generale e separata da ASSISTANT stesso: solamente nell'ultimo capitolo presenteremo la sintassi del linguaggio, dove emergeranno tutte le caratteristiche approfondite nei capitoli precedenti.

Vediamo la suddivisione dei capitoli più in dettaglio:

- **Capitolo 2, Pervasive Computing e Applicazioni ad Alte Prestazioni**: in questo capitolo presentiamo i due settori che hanno portato alla nascita dell'HPPC. Vengono illustrati i problemi di ricerca risolti e quelli ancora aperti, per fornire al lettore una panoramica del settore di ricerca in cui è ambientata la tesi.

- **Capitolo 3, Modelli di programmazione Context-Aware**: analizziamo i modelli più promettenti per lo sviluppo di applicazioni pervasive. Ognuno dei sistemi trattati contiene caratteristiche interessanti e degne di nota, ma non si candida come ambiente definitivo per lo sviluppo di applicazioni pervasive. In questo capitolo mostriamo le tecniche più utilizzate per supportare adattività e context-awareness e, al tempo stesso, come i modelli esistenti non permettano di de-

scrivere applicazioni molto complesse, tantomeno con requisiti di alte prestazioni.

- **Capitolo 4, ASSIST:** gemello del precedente, questo capitolo introduce gli ultimi risultati sui modelli di programmazione parallela. A differenza del capitolo 3, però, ci focalizziamo interamente su un unico ambiente, ASSIST. Rispetto ai modelli visti precedentemente, ASSIST permette di risolvere tutti i problemi fondamentali nell'ambito dell'HPC, e può quindi essere considerato come uno strumento completo. La trattazione estesa ci permette, inoltre, di illustrare in modo completo quelle che saranno le basi del nuovo modello proposto nella tesi.

- **Capitolo 5, Un nuovo modello per applicazioni pervasive ad alte prestazioni**: finalmente iniziamo la trattazione di ASSISTANT. Dopo una prima fase in cui motiviamo la scelta di creare un nuovo modello utilizzando le solide basi di ASSIST, presentiamo per la prima volta in modo dettagliato i concetti che vogliamo introdurre nel nuovo ambiente di sviluppo. Introduciamo una modellazione concettuale di una applicazione "a strati" e definiamo in modo dettagliato i meccanismi da noi proposti per risolvere i problemi di adattività e context awareness.

- **Capitolo 6, Un esempio completo**: in questo capitolo cerchiamo di dare un'idea più concreta dell'utilità del modello, presentando una applicazione studiata nell'ambito del progetto In.Sy.Eme con requisiti di alte prestazioni in un ambiente fortemente pervasivo. Cerchiamo di proporre una trattazione il più generale possibile, senza introdurre ulteriori dettagli su ASSISTANT ma solo i concetti già descritti nel capitolo 5. Le parti parallele sono descritte con ASSIST, ma solamente perché si tratta dell'unico modello HPC presentato nella tesi. Lo scopo di questo capitolo è anche sensibilizzare il lettore alla modalità di analisi e di studio necessari per la realizzazione di una applicazione HPPC.

- **Capitolo 7, ASSISTANT: ASSIST with Adaptivity and coNText awareness**: in questo capitolo viene presentata la sintassi di ASSISTANT in modo completo. Avendo già studiato una applicazione complessa, tutti gli esempi saranno basati su questa, col duplice obiettivo di trattare anche quei problemi non banali che non emergerebbero da esempio "scolastico", e al tempo stesso fornire una implementazione in ASSISTANT dell'applicazione trattata nel capitolo 6.

- **Capitolo 8, Conclusioni**: terminiamo la tesi, descrivendo a grandi linee alcune idee sui futuri lavori che interesseranno ASSISTANT.

Parte I

Lo stato dell'arte

Capitolo 2

Pervasive Computing e Applicazioni ad Alte Prestazioni

Il concetto di *Pervasive Computing*, noto anche come *Ubiquitous Computing*, è stato introdotto alla fine degli anni '80 dal ricercatore Mark Weiser[82].
In questo capitolo cerchiamo di riassumere le idee iniziali del *Pervasive Computing*, l'evoluzione del termine in questi ultimi venti anni e alcuni dei principali problemi di ricerca, risolti o ancora presenti.
Di questi ci focalizziamo sulla *Context-Awareness*, concetto fondamentale per lo sviluppo delle applicazioni pervasive immaginate da Weiser.
Infine presentiamo le ultime ricerche sull'High Performance Computing, descrivendo come queste convergano verso piattaforme pervasive e, quindi, verso un nuovo modello di pervasive computing orientato ad applicazioni ad alte prestazioni.

2.1 Il Pervasive Computing

Durante il suo lavoro allo Xerox Palo Alto Research Center, Weiser sviluppò il concetto di computer "onnipresenti". Una rivoluzione per l'epoca, da poco infatti si parlava di personal computer, e Weiser già immaginava un mondo dove in ogni stanza fossero presenti decine, se non centinaia, di computer dalle forme e dimensioni più varie, collegati tra loro per mezzo di una rete. Ma il *pervasive computing* non rappresentava solo la presenza di tanti computer miniaturizzati. Infatti, per Weiser, un requisito fondamentale era cambiare completamente l'interazione con i computer per renderla talmente semplice

e comune da trasformarla in gesto ovvio e spontaneo. In questo modo le persone si sarebbero "dimenticate" di avere a che fare con dei computer.

Weiser illustrava il *pervasive computing* come l'esatto opposto della *realtà virtuale*. Nel primo caso, infatti, i computer sono talmente "immersi" e "onnipresenti" nel mondo reale che ne diventano parte integrante. Nel secondo invece sono le persone che, per utilizzare i computer, devono entrare in un "mondo virtuale", differente da quello reale.
In questa ottica Weiser presentò tre tipologie di *ubiquitous computer*, che riprendevano equivalenti "cartacei" utilizzati oramai da anni in tutti gli uffici: il *post-it*, il *blocco note* e la *lavagna*. La versione "digitale" di questi oggetti voleva in qualche modo riprendere tutti gli usi di quella originale, aggiungendone altri tipici dei calcolatori.

Le idee di Weiser erano però ancora troppo futuristiche, e i problemi da affrontare molteplici. Al PARC furono sviluppati dei primi prototipi degli *ubiquitous computer* sopra presentati: *Tab*,*Pad* e *Boards*, ma i ricercatori si scontrarono con problematiche sia hardware che software[83].

- Il livello di miniaturizzazione delle componenti hardware non era sufficiente per permettere lo sviluppo di soluzioni che fossero piccole come i Post-it, ma al tempo stesso abbastanza potenti per le idee di Weiser.
- Le connessioni di rete, soprattutto quelle wireless, erano ancora ad uno stato embrionale e non offrivano la banda necessaria a condividere grandi moli di dati.
- Nello sviluppo dei componenti elettronici ancora non si teneva conto del consumo energetico, aspetto fondamentale per questo tipo di dispositivi.
- Infine i problemi software: lo sviluppo su questo tipo di piattaforme doveva tenere in considerazione una miriade di problemi non banali e non ancora affrontati da nessun settore della ricerca.

Ma le supposizioni di Weiser sullo sviluppo delle tecnologie erano corrette. Nell'arco di circa dieci anni l'evoluzione hardware aveva già fatto passi da gigante, e al giorno d'oggi strumenti come cellulari e PDA, GPS, reti wireless e sensori ambientali sono ormai commercializzati e accessibili a tutti. Si aprono quindi nuove frontiere per il *pervasive computing*, oggi definito come

> "un ambiente pieno di oggetti con capacità computazionali e comunicative, talmente integrato con gli utenti da rappresentare una *tecnologia che svanisce*"[65]

In tale ambiente troviamo tutti quei dispositivi digitali dotati di CPU, su cui necessariamente si trovano un *sistema operativo* e dei *programmi applicativi*. Alcuni esempi di tali dispositivi, chiamati spesso "embedded" sono:

- Telefoni digitali (cellulari o anche semplici cordless).
- Palmari (PDA) e smartphone.
- Computer portatili.
- Navigatori GPS.
- SmartCard (carte di credito, badge elettronici, etc).
- Wearable device.
- Sensori (sensori di temperatura, di movimento, etc).
- Televisori.

Questi strumenti devono "cooperare" tra loro utilizzando le proprie capacità trasmissive, nell'ottica di assecondare le necessità dell'utente.

Uno degli esempi più comuni nel campo del *pervasive computing* è quello delle *Smart House*, "case intelligenti" in cui i dispositivi digitali intuiscono le intenzioni dell'utente e lo assecondano. Ad esempio accendendo la luce quando una persona si sposta in una nuova stanza, oppure chiamando numeri d'emergenza se l'utente si sente male. Anche questi due semplici esempi richiedono l'interazione di molti dispositivi: nel primo caso, i sensori che rilevano il movimento dell'utente all'interno delle stanze e gli attuatori che accendono/spengono la luce. Nel secondo, uno o più sensori che rilevano un comportamento anomalo dell'utente, o anche un wearable device che si accorge di una variazione dei valori dello stato fisico della persona (ad esempio battito cardiaco accelerato o assente); questi possono interagire con una qualche base di dati che identifichi il problema come grave, cerchi su internet il numero dell'emergenza sanitaria locale e lo passi ad un telefono digitale, il cui compito sarà chiamare il numero e comunicare all'operatore tutte le informazioni raccolte dai sensori e dalla base di dati.

2.2 La ricerca sul Pervasive Computing

Abbiamo già detto che le supposizioni di Weiser sullo sviluppo delle tecnologie erano corrette. Già oggi una stanza contiene decine di computer dotati

di connessioni wireless; eppure non ci troviamo ancora nel mondo da lui auspicato. I computer sono davvero "onnipresenti", e spesso di dimensioni così piccole da farci dimenticare che lo sono; ma manca quella "coesione", quella "cooperazione" tra dispositivi tipica del *pervasive computing*.

Le limitazioni hardware che rendevano questo concetto poco più di una "visione futuristica" sono ormai superati; nonostante questo, dal punto di vista del software esistono ancora molteplici problemi di ricerca aperti, che cercheremo di riassumere nelle prossime righe.

Da un certo punto di vista il *pervasive computing* può essere visto come un passo evolutivo a partire da due settori di ricerca sviluppatisi a partire dagli anni '70: i *sistemi distribuiti* e il *mobile computing*. Infatti molti dei problemi tipici di questi sistemi si ritrovano nel *pervasive computing*, insieme ad altri completamente nuovi e caratteristici di quest'ultimo. A tal proposito in figura 2.1 riprendiamo la tassonomia proposta in [65], in cui si raggruppano i problemi del *pervasive computing* in tre grandi categorie:

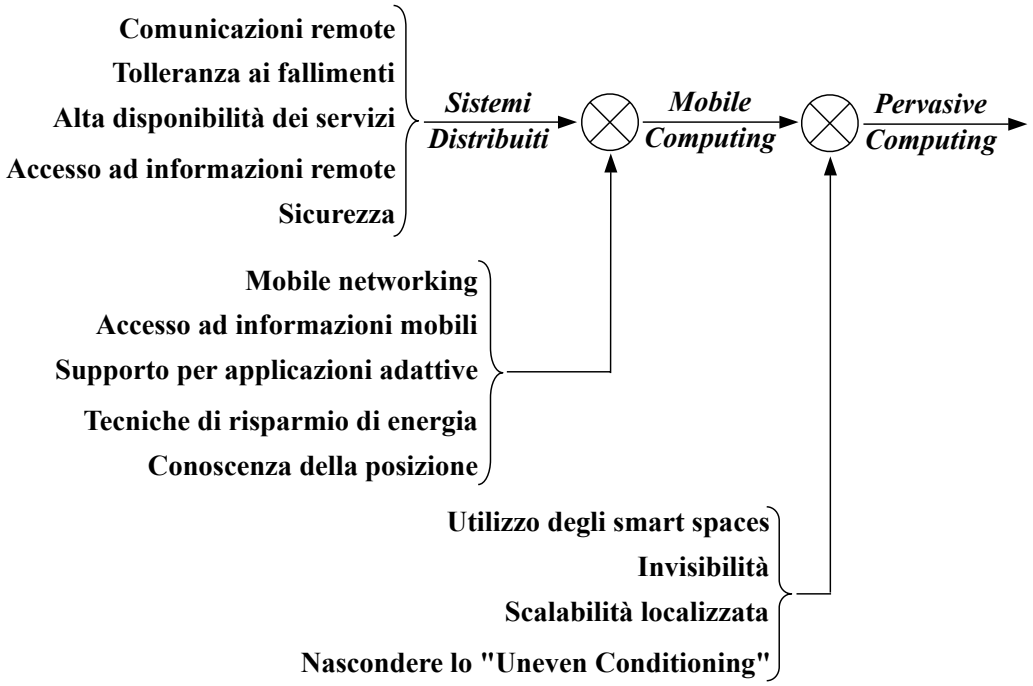
- Derivati dai sistemi distribuiti.
- Derivati dal mobile computing.
- Introdotti dal pervasive computing.

2.2.1 Sistemi Distribuiti

Il campo dei sistemi distribuiti nacque negli anni '70 con l'avvento dei personal computer e delle reti locali. La ricerca nel settore è continuata fino ai primi anni '90, e ha portato a basi concettuali e algoritmiche molto solide, valide per tutti i casi in cui due o più computer connessi tramite rete (sia essa cablata o non, mobile o statica) devono comunicare. Riportiamo qui le ricerche che si applicano anche al *pervasive computing*; per un approfondimento si veda [65].

- **comunicazioni remote**, quindi strutturazione dei protocolli a livelli e suddivisione delle funzionalità tra di essi, tecniche *remote procedure call*, uso dei timeout, etc.
- **tolleranza ai fallimenti**: transazioni atomiche e distribuite, commit a due fasi, etc.
- **alta disponibilità dei servizi**: gestione della replicazione, mirrored execution, etc.

Figura 2.1: I problemi di ricerca introdotti dal pervasive computing, e quelli derivati dai settori già esistenti



- **accesso ad informazioni remote**: database e file system remoti, tecniche di caching, etc.

- **sicurezza**: tecniche di crittografia, autenticazione sicura e gestione della privacy.

2.2.2 Mobile Computing

Il settore del mobile computing è relativamente nuovo rispetto ai sistemi distribuiti: si sviluppa verso la fine degli anni '80 con l'avvento dei primi computer portatili e delle reti wireless. Rappresenta una estensione del precedente, in cui i nodi sono anche di tipo mobile. Fondamentalmente tutti i principi base dei sistemi distribuiti sono riportabili in questo settore, ma devono fare i conti con alcune limitazioni tipiche dei sistemi mobili: *variazioni imprevedibili della qualità della rete*,*risorse locali limitate* e *risparmio energetico*.

Rispetto al precedente, questo è un settore ancora aperto ed in continua evoluzione. I risultati attuali possono essere raggruppati nelle seguenti aree [65]:

- **mobile networking**: protocolli ad hoc e tecniche di ottimizzazione di quelli standard per migliorare la performance delle reti wireless.
- **accesso ad informazioni mobili**: gestione delle disconnessioni dei computer, ottimizzazione della banda, etc.
- **supporto per applicazioni adattive**: gestione adattiva delle risorse, etc.
- **tecniche di risparmio di energia**: applicazioni energy-aware, processori con velocità variabile e gestione della memoria energy-sensitive.
- **conoscenza della posizione**: tecniche per dedurre la posizione geografica e applicazioni con comportamenti dipendenti dalla posizione.

2.2.3 Nuovi problemi

Il *pervasive computing* eredita tutti i problemi sopra descritti, estremizzandoli a causa del gran numero di dispositivi interconnessi e all'alta mobilità degli stessi. Ne vengono poi aggiunti di nuovi, in particolare:

- **Utilizzo degli "Smart Spaces"**: la diffusione di un gran numero di dispositivi embedded all'interno di un'area chiusa può essere sfruttata per "acquisire" dati da essa e modificarne le caratteristiche. Dotando i dispositivi di sensori, infatti, questi possono avere una certa percezione dello spazio; con degli attuatori, invece, modificare le caratteristiche dell'ambiente nell'ottica di renderlo più consono agli utenti presenti. Il risultato finale è uno "spazio intelligente", che si auto-adatta in base alla percezione dell'ambiente e degli utenti in esso presenti.
- **Invisibilità**: uno dei principi base del pervasive computing, già dalla prima idea di Weiser, era la completa "invisibilità" dei dispositivi elettronici da parte dell'utente. Alcune ricerche dimostrano come una buona approssimazione di questo concetto è data dal *minimal user distraction*, ovvero richiedere l'intervento dell'utente il meno possibile e, al tempo stesso, non deludere le sue aspettative; in questo modo l'utente è portato a non interagire direttamente con i dispositivi pervasivi e "dimenticarsi" di essi.

- **Scalabilità localizzata**: un concetto che nasce principalmente in questo settore, dove l'enorme numero di dispositivi presenti rende il problema della scalabilità ancora più pressante. Una delle idee promettenti è quella di studiare un concetto di scalabilità legato alla distanza tra i dispositivi: in un ambiente pervasivo, così come nel mondo reale, le interazioni tra dispositivi sono proporzionali alla loro vicinanza, mentre all'allontanarsi di essi si diradano, in quanto considerate "meno rilevanti".

- **Nascondere lo "Uneven Conditioning"**: con il termine "uneven conditioning" si indica la grande irregolarità (dal punto di vista "pervasivo") degli ambienti in cui l'utente si può trovare. Dobbiamo considerare infatti che ci vorranno decine di anni prima che ogni luogo sia pieno di dispositivi embedded e quindi adatto alla massima espressione del pervasive computing. Fino a quel momento ci saranno grosse differenze di "intelligenza" tra ambienti diversi, che devono in qualche modo essere mascherate all'utente, sempre per il principio dell'invisibilità. Per questo motivo sono necessarie tecniche che permettano di compensare queste differenze, in modo da nasconderle il più possibile all'utente.

Nella ricerca sul pervasive computing due tecniche promettenti per risolvere alcuni dei punti sopra descritti sono *adattività* e *context awareness*.

2.3 Adattività

Il concetto di *adattività* è fondamentale nel pervasive computing, in quanto la forte mobilità dei dispositivi unita al "uneven conditioning" rende imprevedibile la quantità e la qualità di risorse offerte dall'ambiente; in alcuni casi queste potrebbero non essere sufficienti a svolgere le azioni volute. È perciò necessario che l'applicazione, o meglio l'insieme di applicazioni che sfruttano l'ambiente, modifichino il loro comportamento in uno nuovo, adatto per le risorse disponibili ed il più possibile simile a quello richiesto.

Questo adattamento dell'applicazione alle risorse presenti può seguire tre diverse strategie, che portano sia a risultati differenti, sia ad una interazione diversa da parte dell'utente.

1. Modificare il *comportamento dell'applicazione*, in modo da poter essere eseguita sulle risorse presenti; questa tecnica di adattività normalmente riduce la qualità dei risultati, e quindi la percezione di qualità dell'utente[43].

2. Chiedere all'ambiente di garantire un certo livello di risorse con appositi contratti di *Quality of Service*(QoS), necessario per ottenere il risultato voluto. Dal punto di vista dell'utente, questo rappresenta un aumento delle risorse offerte dall'ambiente, magari utilizzando dispositivi altrimenti spenti o in risparmio energetico[60].

3. Chiedere all'utente una *azione correttiva*, un cambiamento nelle azioni che, con buona probabilità, porterà a soddisfare la richiesta di risorse dell'applicazione. Questo terzo caso, concettualmente promettente ma non ancora implementato nella realtà, prevede per esempio la richiesta all'utente di spostarsi in una stanza con più risorse, di accendere nuovi dispositivi, etc.

Tutte e tre le strategie sono importanti nel pervasive computing, e possono essere utilizzate anche insieme, al fine di massimizzare l'esperienza dell'utente. Tra le tre, la strategia 2 ha il pregio di non richiedere assolutamente l'intervento di esso e di essere perciò completamente "trasparente". Non può però essere sempre applicata, in quanto presuppone un ambiente con una infrastruttura adeguata a supportare i meccanismi di QoS che, soprattutto nell'ottica dell'uneven conditioning, non può essere considerata sempre presente. Nel caso le risorse non siano sufficienti, l'applicazione non ha altra scelta che diminuire la qualità dei risultati seguendo la strategia 1, ma al tempo stesso informare l'utente che potrebbe avere risultati migliori se si spostasse in un'altra stanza o se accendesse altri dispositivi (strategia 3).

Ovviamente esistono ancora molte domande aperte in questo settore, come ad esempio:

- Possono esistere tecniche per prevedere il futuro spostamento dell'utente in un ambiente con risorse non sufficienti ed avvertirlo in tempo? Nel caso questi potrebbe portare l'applicazione su un dispositivo più potente, che possa sopperire alle mancanze dell'ambiente futuro. Ad esempio, se l'utente sta andando in una sala riunioni per fare una presentazione col suo PDA, e la sala è sprovvista di un computer collegato al proiettore, potrebbe spostare la i file dal PDA ad un portatile ed usare questo per la presentazione.

- È possibile costruire computer altamente modulari in modo che l'utente possa portare con se solo i moduli necessari per la propria applicazione e "montarli" su un computer già presente nell'ambiente? In parte questo è già possibile oggi, ma l'operazione deve essere resa notevolmente più semplice ed attuabile da un utente medio. Alternativamente, queste espansioni potrebbero essere pensate per il dispositivo trasportato

dall'utente, che quindi lo configurerebbe in base alle applicazioni che deve eseguire.

2.4 Context-Awareness

Un sistema pervasivo, per definizione, deve poter essere "cosciente" dell'ambiente in cui si trova. Questa conoscenza deve essere il più possibile dettagliata, in modo da minimizzare le interazioni con l'utente. Al tempo stesso, un dispositivo più essere spostato, a causa dell'alta mobilità di essi. Perciò la conoscenza non è statica, in quanto il mondo in cui l'oggetto si trova cambia nel tempo. Per questo motivo un dispositivo pervasivo deve essere *context-aware*; deve cioè conoscere il contesto, l'ambiente in cui si trova. La conoscenza del contesto deve essere poi utilizzata dal dispositivo, che deve adattare le sue funzioni, il suo comportamento, all'ambiente attuale.

In questo ambito uno dei punti chiave è l'ottenere le informazioni richieste dall'applicazione per avere un comportamento dipendente dal contesto. Queste informazioni possono essere statiche e già conosciute dell'applicazione, come ad esempio la lista dei contatti telefonici dell'utente o gli appuntamenti quotidiani; altre invece sono ottenute dinamicamente dall'ambiente, come ad esempio la posizione dell'utente nella stanza, le persone vicine, etc.

La storia dei sistemi context-aware inizia nel 1992, con la realizzazione dell'*Active Badge Location System*[81], considerato una delle prime applicazioni context-aware; non a caso è anche uno degli esempi citati da Weiser di pervasive computing[82].
Il sistema era composto da una serie di badge "intelligenti", che permettevano di localizzare i dipendenti all'interno dell'edificio ed inoltrare le chiamate telefoniche al dispositivo a loro più vicino.
Successivamente sono stati introdotti sistemi di visite guidate[1, 29], anche in questo caso "location-aware", che fornivano agli utenti informazioni in base alla loro posizione.

Tutti questi lavori utilizzavano, come informazioni di contesto, la posizione geografica degli utenti; solo successivamente sono state sviluppate e definite applicazioni più generali, che utilizzassero altri tipi di dati di contesto. Parallelamente a ciò sono state date molte definizioni di "contesto"; una delle più accurate secondo [19] è quella di Dey e Abowd:

> "Ogni informazione che può essere utilizzata per caratterizzare lo stato delle entità (siano esse persone, luoghi o oggetti) che sono considerate rilevanti per l'interazione tra un utente e una applicazione, inclusi l'utente e l'applicazione stessi."[39]

Secondo questa definizione le informazioni di contesto sono molteplici; per questo motivo sono spesso suddivise in due sottocategorie:

- **esterne** *o fisiche*: tutti i dati "fisici", raccolti da sensori; ad esempio la temperatura, la posizione, la quantità di luce, etc.
- **interne** *o logiche*: tutti i dati raccolti dall'utente, come ad esempio i suoi obiettivi, lo stato d'animo, il contesto di lavoro, etc.

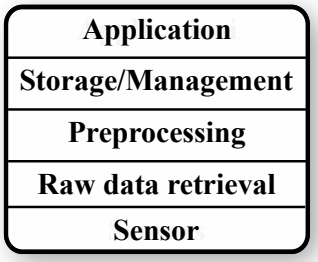
2.4.1 Struttura di un sistema context-aware

In linea di massima un sistema context-aware può essere implementato in vari modi. Vista l'età relativamente giovane del settore, non è stata ancora definita una tecnica "standard" per la realizzazione di applicazioni context-aware; al contrario, l'approccio da utilizzare dipende spesso dalle caratteristiche dell'applicazione che vogliamo sviluppare, come la posizione dei sensori (accessibili direttamente oppure tramite gateway), la quantità di possibili utenti, la tipologia di risorse utilizzabili, etc.

Uno dei problemi importanti in questo tipo di applicazioni è definire il metodo con cui i dati di contesto vengono acquisiti, poiché questa scelta influenza l'architettura dell'intera applicazione. Una delle suddivisioni più utilizzate è quella di Chen[28], che presenta tre modalità per acquisire i dati di contesto:

- **Diretto**: questo approccio prevede l'accesso diretto da parte dell'applicazione ai sensori, per reperirne i dati. Questa tecnica viene utilizzata frequentemente per dispositivi con sensori integrati; non prevede un livello tra i sensori e l'applicazione, che quindi deve essere dotata di driver per accedere ad ogni tipologia di sensore, e richiede all'applicazione di occuparsi di tutti i dettagli legati ad essi. Per questo motivo non è utilizzabile in caso di sensori non integrati.
- **Tramite middleware**: l'approccio con middleware prevede l'accesso ai sensori attraverso una architettura a livelli, dove l'applicazione richiede le informazioni al livello sottostante (il middleware, appunto), che maschera tutti i dettagli sull'accesso ai singoli dispositivi. Rispetto alla precedente questa tecnica è molto più flessibile e semplifica lo sviluppo delle applicazioni, che non devono più occuparsi dei meccanismi di accesso alle singole tipologie di sensori.
- **Con server di contesto**: in questo caso i dati dei sensori vengono tutti raccolti da una singola entità, il "context server", che si occupa poi di gestire l'accesso ai dati da parte di una o più applicazioni.

Figura 2.2: I livelli concettuali di un framework per lo sviluppo di applicazioni context-aware



Rappresenta un ulteriore livello tra i sensori e l'applicazione (il server può a sua volta utilizzare un middleware), e offre maggiori funzionalità alle applicazioni finali. Si può infatti presumere che questo server abbia una discreta potenza computazionale, che può essere utilizzata per scaricare i dispositivi embedded da operazioni intensive. Rappresenta quindi una risorsa aggiuntiva dell'ambiente, utilizzabile ad esempio per i requisiti di QoS.

La maggior parte dei sistemi per sviluppare applicazioni context-aware utilizzano un accesso ai sensori tramite middleware o server di contesto, utilizzando quindi un approccio a "livelli", che permette di semplificare lo sviluppo delle applicazioni. Col tempo i livelli di un sistema context-aware sono stati definiti in modo più preciso, e gli attuali framework prevedono una strutturazione a cinque livelli, come in figura 2.2.

Sensor

Questo livello rappresenta l'insieme dei sensori presenti nel sistema che producono informazioni di contesto. Questa accezione di "sensori" include, quindi, non solo tutti i sensori hardware che rilevano misure fisiche, ma anche i sensori logici, sulle risorse, etc. In questo senso sono spesso definiti tre tipi di sensori[55], con una classificazione che riprende ed estende quella sulle informazioni di contesto.

- *Sensori fisici*: la tipologia più classica di sensori, che si occupa di misurare caratteristiche fisiche dell'ambiente. In tabella 2.1 sono presenti le principali tipologie di sensori fisici.

Informazione	Tipi di sensori
Luce	Fotodiodi, sensori di colore, IR, UV
Immagini	Videocamere e fotocamere di vario tipo
Audio	Microfoni
Movimento	Accelerometri, motion detectors
Posizione	GPS, GSM, etc
Temperatura	Termometri

Tabella 2.1: I tipi di sensori più utilizzati

- *Sensori virtuali*: generano informazioni di contesto in base a dati da applicazioni o servizi. Ad esempio, si può immaginare un sensore virtuale che definisce la posizione di una persona in base alla lista di appuntamenti giornaliera, oppure l'attività corrente dell'utente controllando gli applicativi aperti sul proprio computer.
- *Sensori logici*: questi analizzano i dati ottenuti dai sensori fisici e virtuali per definire informazioni più complesse; per esempio capire se in una stanza non sono presenti persone, oppure determinare la posizione di un impiegato in base al computer attualmente utilizzato e alla posizione di quest'ultimo all'interno dell'azienda.

Raw Data Retrieval

Questo livello si occupa di acquisire i dati dai sensori, attraverso appositi driver per quelli fisici e API per quelli virtuali e logici. Offre al livello superiore un metodo più generale per ottenere i dati, trasparente alla tipologia di sensore utilizzata, in modo da offrire interfacce riusabili anche modificando il livello sottostante. Ad esempio modificare un sistema di localizzazione basato su sensori RFID con uno basato su ricevitori GPS non deve modificare i layer superiori, che continueranno ad utilizzare un metodo *getPosition()* per ottenere la posizione dell'oggetto, indipendentemente dal sensore sottostante.

Preprocessing

Questo livello è in realtà opzionale e non tutti gli ambienti lo offrono. Permette di effettuare una prima elaborazione dei dati ottenuti dai sensori, utile ad esempio se la quantità di dati ottenuti dagli stessi è elevata. Questo livello si occupa di analizzare i dati ed interpretarli per renderli più utili per i livelli

applicativi, a cui vengono quindi offerte informazioni ad un livello maggiore di astrazione. Le operazioni di questo livello possono anche essere non banali e lavorare con dati ottenuti da più sensori; tornando all'esempio della localizzazione, una applicazione potrebbe non essere interessata alle coordinate geografiche dell'oggetto, ma alla posizione all'interno dell'edificio (ad esempio il piano e la stanza).
In altri casi non si è interessati a tutti i valori ottenuti dai sensori ma ad informazioni più elaborate, ad esempio capire se la temperatura di una stanza è variata sensibilmente, oppure se si discosta in modo elevato da quella media. In questi casi sono necessarie operazioni di analisi statistica e dei dati storici ottenuti o tramite sensori o tramite basi di dati.

Queste operazioni di preprocessing sono comunque facoltative, nel senso che possono essere spostate a livelli differenti: se ne può occupare direttamente l'applicazione, oppure si può racchiudere queste informazioni all'interno di sensori logici. Spostare queste operazioni a livello applicativo, soprattutto se è presente un server di contesto, potrebbe però limitare le prestazioni dell'applicazione, specialmente dal punto di vista della rete, in quanto il preprocessing normalmente prende in ingresso una grande quantità di dati per produrne pochi. Per questo motivo trasferire ai dispositivi solo il risultato del preprocessing, e non tutti i dati da analizzare, può permettere un notevole risparmio di banda. Analogamente l'utilizzo del server di contesto per effettuare queste analisi libera il dispositivo mobile da computazioni intensive. È quindi preferibile definire un livello di preprocessing delle informazioni di contesto, per facilitare la creazione di applicazioni più semplici e performanti.

Un altro problema spesso considerato a questo livello è quello della gestione dei conflitti: le informazioni di contesto ricevute potrebbero essere tra di loro contraddittorie, a causa di ritardi temporali o di sensori di bassa qualità. Durante la fase di preprocessing ci si occupa quindi di scartare quelle informazioni che renderebbero il contesto "contraddittorio".

Storage and Management

Questo livello, il più vicino all'applicazione, si occupa di organizzare ed eventualmente immagazzinare i dati ricevuti da quello sottostante. In questo livello sono definite le interfacce con cui l'applicazione può accedere alle informazioni di contesto.

Normalmente le modalità di accesso ai dati si distinguono in:

- *sincrono*: l'applicazione richiede, quando vuole, le informazioni di contesto a questo livello, con chiamate in stile RPC. Questa modalità è utile se l'applicazione, durante la sua vita, vuole conoscere l'attuale

stato dell'ambiente. Ma se invece è interessata ad accorgersi di cambiamenti del contesto, deve periodicamente richiedere tali informazioni e controllare se sono variate dalle precedenti.

- *asincrono*: questa modalità sopperisce ai problemi della precedente: in questo caso l'applicazione si "registra" su specifici eventi (ad esempio l'arrivo di un nuovo utente nella stanza, il cambiamento della temperatura, etc) e viene "avvisata" quanto questo evento si verifica.

Normalmente la tecnica più utilizzata nelle applicazioni context-aware è la seconda: per una applicazione di questo tipo, infatti, la prerogativa principale è adattare il proprio comportamento all'attuale contesto. Questo può essere ottenuto semplicemente modificandosi (se necessario) ad ogni cambio significativo dello stato. L'applicazione ha quindi un comportamento "ad eventi", che si ottiene nativamente con la modalità asincrona.

Application

Il livello finale, in cui si trova l'applicazione context-aware, che utilizza il sottostante per ottenere informazioni dal contesto ed in base a queste modifica il proprio comportamento.

Inoltre, come già detto precedentemente, qui si trovano tutte le funzionalità non espresse nei livelli sottostanti. Ci riferiamo principalmente alla parte di preprocessing, che può non essere presente in certi ambienti di sviluppo o essere comunque troppo semplice per descrivere il comportamento voluto.

2.4.2 Modelli per la rappresentazione del contesto

Un altro punto fondamentale di un sistema context-aware è il modello con cui vengono rappresentate le informazioni di contesto. Fino a questo punto abbiamo parlato dei tipi di informazioni presenti e di come queste arrivano all'applicazione. Non è stato definito, invece, in che forma sono modellate.
I modelli comunemente utilizzati per rappresentare le informazioni sono descritti nella loro interezza in [19]; noi citiamo solo i principali:

- *Modelli chiave-valore*: le informazioni sono rappresentate come coppie, dove la chiave rappresenta univocamente una informazione di contesto. Questa tecnica è molto semplice e poco pesante dal punto di vista computazionale. Purtroppo però non permette di definire relazioni tra differenti informazioni, e risulta perciò piuttosto limitante.

- *Modelli basati su ontologie*: le ontologie permettono di descrivere concetti e relazioni fra essi in modo formale, e si adattano bene a meccanismi di inferenza e *ontology reasoning*; per questo motivo, pur essendo un modello più complicato di quello "chiave-valore", sono la scelta più comune in sistemi context-aware.

2.5 Pervasive Computing e HPC

Nell'idea iniziale di Weiser il pervasive computing era utilizzato in ambienti chiusi, che diventavano "intelligenti" ed aiutavano l'utente nei compiti di tutti i giorni. L'idea di base del Pervasive Computing non si è poi discostata più di tanto, e fino a questo punto anche noi abbiamo portato esempi di questo tipo.

Recentemente si sta però pensando di applicare i risultati e le idee del pervasive computing anche ad altre realtà, tra cui quella dei programmi ad alte prestazioni: in questo ambito da qualche anno si sta studiando una evoluzione delle griglie computazionali, chiamata *Pervasive Grid*[62]; è poi nato un nuovo settore di ricerca, estensione del pervasive computing, in cui si trattano ambienti di dimensioni maggiori e ancora più eterogeneità di dispositivi, con l'obiettivo primario di eseguire nel miglior modo possibile applicazioni che necessitano di molta potenza di calcolo; questo settore viene chiamato *High Performance Pervasive Computing*(HPPC)[58].

La convergenza di questi due settori non è poi così "strana" o "forzata" in quanto entrambi, pur focalizzandosi su problematiche differenti, fondano le proprie radici sul *Distributed Computing*. La loro discendenza comune si fa continuamente sentire, e ha portato i ricercatori a studiare da una parte applicazioni pervasive con problematiche computazionali e dall'altra applicazioni ad alte prestazioni su reti altamente mobili.

2.5.1 Pervasive Grid

Il grid computing si è rivelato negli ultimi anni come il paradigma principale per lo sviluppo di applicazioni distribuite su grandi distanze. L'obiettivo originale delle griglie computazionali[46] era di combinare risorse rese a disposizione da varie organizzazioni in un unico ambiente su cui eseguire applicazioni parallele, nell'ottica di studiare problemi così grandi da essere considerati, fino a quel momento, irrisolvibili. Al giorno d'oggi, dopo quasi un decennio di studi e ricerche, il grid computing è ormai una realtà, e si contano molti strumenti per lo sviluppo di applicazioni su griglie computazionali[4, 17, 45]. La ricerca non è però terminata, e negli ultimi anni la presenza sempre più

importante del pervasive computing ha suggerito la possibilità di un nuovo concetto di griglia, la cosiddetta *pervasive grid*, in cui anche i dispositivi embedded sono uniti nel formare una griglia[52, 63].

Nel pervasive grid confluiscono, estremizzate, le caratteristiche di eterogeneità e volatilità tipiche degli ambienti grid comuni; diventa quindi necessario gestire tutti gli aspetti visti fino ad ora del pervasive computing, tra cui adattività e context-awareness.
Tutti i meccanismi di deployment e resource discovery studiati nell'ambito grid possono essere adattati ad ambienti pervasivi, e offrire una visione "grid-style" alle applicazioni pervasive, che permetta sia di utilizzare tutti gli strumenti già sviluppati per il settore del grid computing, sia di gestire quei calcoli computazionalmente pesanti che un singolo dispositivo non potrebbe eseguire[38, 62, 70].

2.5.2 High Performance Pervasive Computing

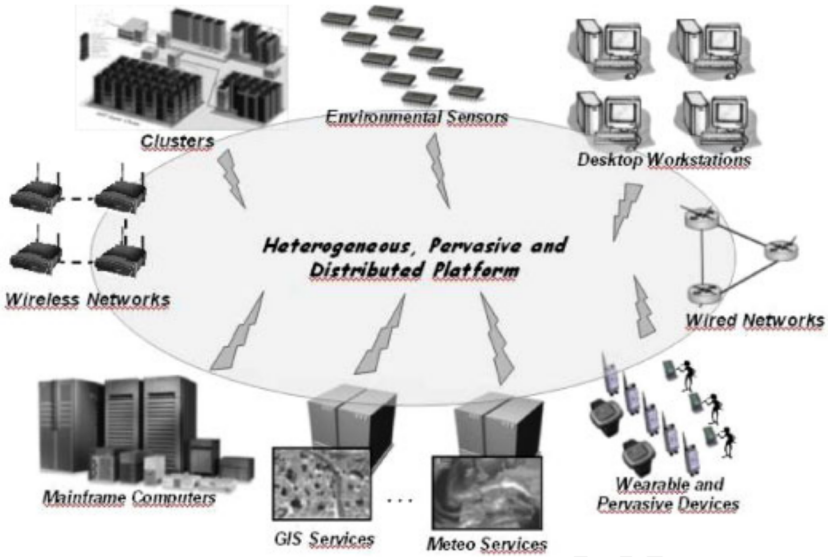
Uno dei maggiori esempi per l'HPPC è la *gestione delle emergenze ambientali*[42, 74], in cui operatori mobili e non (polizia, medici, vigili del fuoco, protezione civile, etc) collaborano per gestire situazioni di pericolo ambientale. Questo tipo di applicazioni sono significative, perché prevedono l'elaborazione di una grande quantità di informazioni, ottenute in parte da sensori dislocati nelle aree a rischio, in parte da altri servizi (Meteo, fotografie satellitari, etc), sia per motivi di *prevenzione*, sia per la *gestione dell'emergenza*.

In generale gestire questo flusso di informazioni ed utilizzarlo per previsioni accurate è una operazione non banale che richiede grandi capacità di calcolo, per eseguire algoritmi di simulazione, data mining, etc. Questo rende necessario l'utilizzo di sistemi paralleli, e di tutte le tecniche e tecnologie tipiche dell'HPC. Allo stesso tempo, però, soprattutto in caso di emergenze, abbiamo a che fare con un grande numero di dispositivi di piccole dimensioni nell'area geografica vicina all'emergenza. In questi casi, proprio a causa dell'emergenza stessa, le risorse normalmente utilizzate per analizzare i dati potrebbero non essere raggiungibili; nell'ottica di aiutare al meglio gli operatori si rende *necessario* l'utilizzo delle poche risorse accessibili sul luogo, per generare delle previsioni "peggiori" ma che possano comunque dare agli operatori un'idea di cosa potrebbe accadere.

In figura 2.3 è riportato uno schema delle differenti tipologie di dispositivi presenti in questo tipo di applicazione[58].

Queste applicazioni hanno quindi tutte le caratteristiche del *pervasive computing*, associate alla necessità (tipica dell'*HPC*) di effettuare calcoli non banali sulla grande quantità di dati raccolta.

Figura 2.3: Esempio delle tipologie di dispositivi utilizzati in una applicazione di gestione delle emergenze



Pur essendo una idea completamente nuova, l'HPPC eredita gli ultimi studi sulle *pervasive grid* come ambienti di esecuzione per applicazioni parallele altamente pervasivi ed eterogenei[62, 63].

Allo stesso modo sono già presenti molti studi su sistemi di adattività per programmi paralleli, orientati principalmente (ma non solo) all'ottimizzazione automatica delle prestazioni dell'applicazione rispetto alle risorse disponibili[10, 41, 79].

Capitolo 3

Modelli di programmazione Context-Aware

Nel capitolo precedente abbiamo mostrato i principali problemi da affrontare nello sviluppo di applicazioni pervasive.
Fino alla fine degli anni '90 realizzare tali sistemi significava trattare tutti questi problemi a livello applicativo, aumentando notevolmente la complessità nello sviluppo di tali applicazioni.

Per questo motivo i ricercatori si sono concentrati sulla definizione di strumenti atti a semplificarne la realizzazione. Questo ha portato alla nascita, negli ultimi anni, di un numero sempre maggiore di sistemi per lo sviluppo di applicazioni context-aware. Vista l'enorme quantità di problemi da risolvere in tali ambienti, gli approcci utilizzati nella definizione e nello sviluppo di questi sono stati di due tipi:

1. studio delle caratteristiche di alcune applicazioni di esempio, e realizzazione del sistema tenendo conto dei requisiti di tali applicazioni;

2. studio di un particolare problema legato alla context awareness (gestione dei sensori, definizione dei dati acquisiti da essi, mobilità, etc) e successiva realizzazione di un framework orientato alla risoluzione di quel particolare problema.

Questi due approcci hanno permesso agli sviluppatori di focalizzare la loro attenzione su alcuni dei punti chiave della context-awareness, riuscendo a definire soluzioni buone ed eleganti per tali problemi. Al tempo stesso, però, questa attenzione ristretta ha portato ad ambienti di sviluppo che si adattano alle applicazioni di esempio, ma non sono abbastanza generali per definire altre tipologie di programmi context-aware.

Analizzando la letteratura ci troviamo perciò davanti ad un gran numero di sistemi di sviluppo di applicazioni pervasive, tutti diversi e focalizzati su differenti aspetti della context-awareness. Nessuno però si candida ad ambiente "definitivo" per lo sviluppo di applicazioni pervasive, ed in particolare nessuno offre le caratteristiche richieste per lo sviluppo di una applicazione HPPC come quella descritta nel capitolo precedente.

In questo capitolo presenteremo alcuni degli ambienti di sviluppo esistenti, descrivendone l'architettura e le principali caratteristiche. Tra tutti i sistemi discussi in letteratura abbiamo scelto quelli che presentano idee innovative o particolari, e che ci hanno fornito importanti spunti per la definizione del modello di ASSISTANT.

Infine studieremo brevemente i problemi che rendono complicata, o impossibile, la realizzazione dell'applicazione di gestione delle emergenze descritta nella sezione 2.5.2 utilizzando i sistemi presentati, sottolineando però le caratteristiche di questi che potrebbero essere utilizzate in un nuovo ambiente di sviluppo specifico per questo tipo di applicazioni, come ASSISTANT.

3.1 Odyssey

Tra il lavori presentati, *Odyssey* è il più vecchio: l'articolo che lo presenta ([61]) risale al 1997. Il lavoro su Odyssey è poi confluito in *Aura*[47], di cui parliamo nella sezione successiva. Nonostante questo consideriamo molto importanti alcuni dei principi chiave di questo modello, passati in secondo piano nella sua evoluzione; per questo motivo lo presentiamo come framework indipendente.

Il problema che Odyssey si propone di risolvere è quello dell'adattività. Come descritto ampiamente nella sezione 2.3, in un ambiente pervasivo le applicazioni possono trovare una quantità di risorse inferiore a quella richiesta per assicurare una corretta esecuzione. In questo caso è necessario modificare il comportamento delle stesse o dell'ambiente, in modo da poterle eseguire con le risorse presenti.

In questo ambito Odyssey implementa un framework per gestire l'adattività tramite la modifica del comportamento delle *applicazioni*. Questa variazione porta ad una riduzione della qualità percepita dall'utente, che in Odyssey viene definita "fidelity". Una diminuzione della fidelity porta ad una applicazione che svolge il suo lavoro con una qualità minore ma che richiede meno risorse. In questo modo una applicazione può adattarsi all'ambiente (o contesto) in cui è eseguita.

3.1.1 Il caso d'uso principale

Odyssey nasce nell'ottica di permettere lo sviluppo di applicazioni adattive e context-aware in esecuzione su dispositivi mobili. Tali applicazioni richiedono normalmente l'utilizzo di una connessione di rete per comunicare con altri dispositivi ed eseguire le operazioni richieste dall'utente.

Lo scenario di esempio presentato con Odyssey è quello di un turista che si muove in città con un dispositivo indossabile (PDA,Smartphone, etc). All'interno della città il dispositivo si troverà a contatto con più reti wireless, variabili nel tempo, e dovrà di volta in volta scegliere quella più adatta alle proprie applicazioni. Verosimilmente l'utente interagirà col dispositivo tramite meccanismi di riconoscimento vocale, e potrà ad esempio ascoltare una web radio, oppure visualizzare un sito internet o ancora controllare la mappa della città per trovare un particolare negozio. Tutte queste informazioni richiedono una connessione ad internet, che in alcuni momenti potrebbe non esistere o essere limitata. In questo caso le applicazioni dovranno in qualche modo adattarsi alla connessione wireless utilizzabile. Un secondo fattore importante sull'utilizzo delle risorse è il consumo energetico: se l'utente prevede di girare la città per tre ore, il dispositivo (e le sue applicazioni) dovrebbe fare di tutto per non esaurire prima la batteria.

Odyssey controlla le risorse del dispositivo: connessione di rete, utilizzo della cpu e batteria residua, ed interagisce con le applicazioni richiedendo ad esse un cambio di "fidelity" se le risorse non sono sufficienti per l'attuale esecuzione, o al contrario sono aumentate e permettono una qualità maggiore. Tutti questi cambiamenti devono avvenire in modo trasparente all'utente che, pur accorgendosi del cambiamento di qualità nel servizio ricevuto, non dovrà in nessun modo richiederlo manualmente.

3.1.2 La fidelity

Punto chiave che ha portato allo sviluppo di Odyssey è il concetto di "fidelity": una applicazione adattiva deve poter modificare la qualità del servizio erogato in base alle risorse su cui viene eseguita. Questo tipo di adattività ha un grosso problema: è legata all'applicazione. Questo significa che normalmente non è possibile definire tecniche standard di variazione di fidelity adatte a tutte le applicazioni e nei pochi casi in cui queste tecniche esistano una variazione della fidelity definita sulla singola applicazione porta a risultati migliori, ovvero un rapporto qualità percepita/risorse utilizzate maggiore.

Uno dei pochi esempi di variazione della fidelity indipendente dall'applicazione è quello della *consistenza*: in caso di risorse limitate possiamo

fornire un risultato "non aggiornato", ad esempio utilizzando una copia presente nella cache dell'applicazione. Questa tecnica è ragionevole in molti casi, ma utilizza così poche informazioni sull'applicazione che si rivela spesso inefficiente (basso rapporto qualità/risorse).

In molti casi possiamo pensare a tecniche più adatte alla singola applicazione, ad esempio:

- Per un *media player* possiamo diminuire la qualità percepita utilizzando una versione del file con compressione maggiore. Questo porta ad una riduzione della dimensione del file al costo di una qualità minore.
- Per una applicazione di *visualizzazione di mappe* possiamo aumentare o diminuire la quantità di dettagli visualizzati: nascondere le strade di dimensione minore, visualizzare solo i nomi principali, etc.
- Per un browser web un esempio di riduzione della fidelity è di scaricare solo le pagine html senza immagini, oppure con versioni a bassa risoluzione di quest'ultime.

I casi sopra elencati forniscono dei dati aggiornati, ma con dettagli minori; per questo motivo sono preferibili all'utilizzo di copie in cache discusso precedentemente.

È importante notare che tutti i tipi di adattività sopra menzionati si basano sul cambiare la qualità di un dato utilizzato dall'applicazione. Questa è una assunzione base degli sviluppatori di Odyssey: la fidelity ed il concetto di adattività sono definiti su dati. Questa ipotesi non risulta comunque restrittiva, se si considera (come spesso accade) l'applicazione finale, quella eseguita sul dispositivo mobile, un mero visualizzatore di tali dati.

La dipendenza dell'adattività dalla singola applicazione diventa però un problema nella definizione di strumenti come Odyssey perché non è possibile disaccoppiare completamente applicazione e adattività, in quanto la seconda è strettamente legata alle caratteristiche della prima.

Questa stretta dipendenza potrebbe portare a pensare che il ruolo di uno strumento di supporto diventi minimale; in realtà questo può occuparsi di monitorare le risorse presenti e notificare all'applicazione la necessità di un adattamento. Si libera perciò lo sviluppatore dalla gestione delle risorse, lasciando però la definizione delle politiche adattive a quest'ultimo. Questo utilizzo del supporto permette inoltre l'esecuzione contemporanea di più applicazioni nell'ambiente: una applicazione ha visione solo di se stessa, e non può calibrare l'utilizzo delle risorse in modo cooperativo con le altre. Spostando questa gestione ad un livello di supporto unico a tutte le applicazioni

possiamo gestire in modo migliore le risorse, definendo politiche adattive generali e non limitate ai singoli programmi; privilegiando ad esempio alcune applicazioni rispetto ad altre ritenute meno importanti dall'utente.

Questo tipo di supporto può essere visto in realtà come un compito del sistema operativo: quest'ultimo infatti gestisce già due importanti risorse come il processore e la memoria. La proposta di Odyssey è proprio quella di estendere il supporto offerto dal sistema operativo, affidandogli la gestione di altre risorse importanti per applicazioni pervasive: connessione di rete e utilizzo delle risorse energetiche.

In realtà, rispetto al supporto minimale descritto, Odyssey vuole spingersi oltre: ad un cambiamento dello stato delle risorse l'applicazione non è notificata del nuovo stato, ma di un dato più raffinato: la *fidelity massima* ottenibile. Con questo l'applicazione può decidere il grado di fidelity adeguato, rispettando il limite imposto.

3.1.3 Architettura di Odyssey

Odyssey è stato sviluppato come estensione di sistemi operativi esistenti, inizialmente NetBSD[61] e successivamente anche Linux[43]. L'idea è quella di inserire all'interno del sistema operativo le routine di gestione delle risorse, lasciando il codice dell'applicazione fuori dal kernel e dal sistema operativo.

Abbiamo già detto come la definizione di fidelity sia dipendente dall'applicazione; perciò con Odyssey l'applicazione viene suddivisa in due parti:

1. una prima che produce un dato, con una certa fidelity;

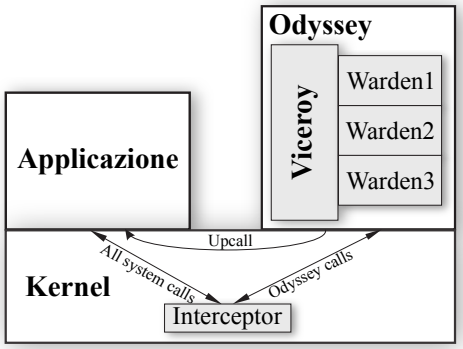
2. una seconda che visualizza il prodotto della prima.

Il sistema operativo (Odyssey) si interpone tra le due entità e ne permette la comunicazione, sia per l'accesso ai dati prodotti, sia per modificare il grado di fidelity richiesto. Odyssey controlla continuamente le risorse utilizzate dal lato produttore, e nel caso queste siano maggiori di quelle allocabili, collabora col lato 1 per calcolare il grado di fidelity massima ottenibile e notifica al lato 2 la necessità un cambiamento; quest'ultimo è libero di richiedere una qualsiasi qualità inferiore a quella massima permessa.

Queste comunicazioni richiedono, ovviamente, una interazione col kernel del sistema operativo, che è però mantenuta al minimo indispensabile. L'architettura finale di Odyssey è rappresentata in figura 3.1; vediamo in dettaglio punti salienti[61].

- La parte **Applicazione** rappresenta il visualizzatore, la parte di applicazione che si occupa di visualizzare il dato prodotto

Figura 3.1: L'architettura di Odyssey



- La parte di applicazione che produce il dato si trova nei **Warden**. Ogni warden ha il compito di produrre un tipo di dato. Possono esistere più warden, legati ad una o più applicazioni in esecuzione. I warden producono dati su richiesta delle applicazioni, in base ad un grado di fidelity fissato.

- I warden sono coordinati da una singola entità, il **Viceroy**, che si occupa di instradare le richieste delle applicazioni verso il warden interessato.

- Le applicazioni accedono ai dati prodotti dai warden attraverso una astrazione di VFS, *Virtual File System*: l'accesso ad un particolare file causa l'invocazione del warden relativo per la sua produzione; in questo modo non sono necessarie nuove API per la comunicazione tra *Application* e *Warden*. Per aumentare la flessibilità è stata comunque inserita una primitiva *tsop*, Type Specific OPeration, che permette di richiedere operazioni particolari ai warden.

- Il kernel si occupa della comunicazione tra i due lati dell'applicazione; in particolare sono definite nuove API per la richiesta, da parte dell'applicazione, del grado di fidelity ed un meccanismo simile a quello dei segnali per notificare una variazione nel grado massimo ottenibile dal produttore.

Vediamo ora in dettaglio i meccanismi di monitoring per le due nuove risorse gestite dal sistema operativo: connessione di rete e batteria.

Controllo sulla connessione di rete [61]

Odyssey controlla continuamente la banda massima ottenibile dalla connessione di rete e la confronta con la quella richiesta dalle varie applicazioni, ottenuta dal viceroy interrogando i singoli warden. Se la banda richiesta è minore di quella fornita, notifica ai warden la banda a loro assegnata e riceve in risposta il grado di fidelity massimo ottenibile. Inoltra questo dato all'applicazione, che decide la nuova fidelity da utilizzare e la richiede al rispettivo warden.

Questa tecnica, in cui la scelta di fidelity è sempre richiesta all'applicazione è importante, in quanto rende quest'ultima libera di definire una politica di adattività propria e non lascia nessuna decisione al supporto.

Controllo sul consumo energetico [43]

Per quanto riguarda il controllo sul consumo energetico gli sviluppatori di Odyssey hanno creato un profiler e, insieme ad un multimetro digitale che analizza il consumo effettivo del computer, sono riusciti a stabilire il consumo dei singoli programmi in esecuzione.

L'ottica finale di Odyssey è quella di garantire la durata minima del dispositivo richiesta dall'utente. Questa durata potrebbe però non essere valida: se tutte le applicazioni, utilizzando la fidelity minima, consumano comunque troppo per garantire la richiesta, il sistema operativo informa l'utilizzatore, che può specificare un nuovo obiettivo.

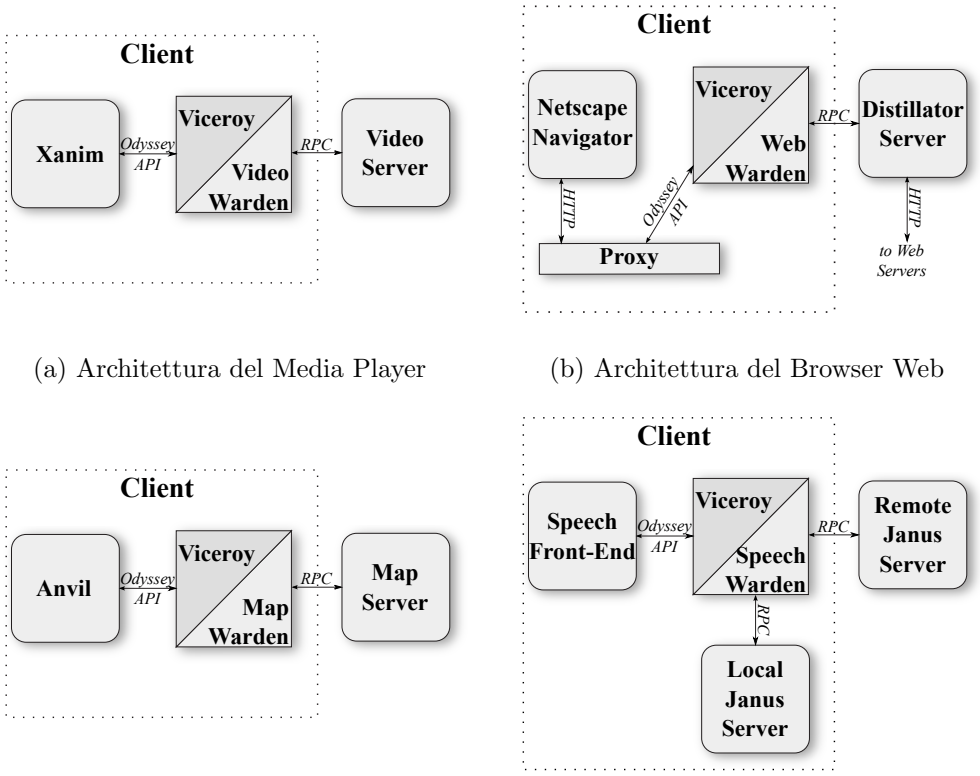
Odyssey calcola continuamente una stima del consumo futuro delle applicazioni in esecuzione, attraverso l'analisi dei dati attuali ottenuti dal profiler, e la confronta con la quantità di energia rimasta. In questo modo determina se le applicazioni devono diminuire la loro fidelity, ma anche se possono permettersi di aumentarla, nell'ottica di rispettare la durata richiesta dall'utente.

Nel caso sia necessario un cambio di fidelity questo viene notificato all'applicazione tramite il meccanismo illustrato precedentemente.

3.1.4 Esempi di applicazioni Odyssey

Per dimostrare l'utilità di Odyssey e l'effettivo vantaggio ottenibile tramite l'utilizzo di applicazioni adattive, gli sviluppatori hanno lavorato su quattro tipologie di applicazioni[43, 61], lavorando su programmi esistenti opportunamente modificati per lavorare sul nuovo sistema operativo.

Per tutte le applicazioni sono stati effettuati test sulla fidelity ottenuta in caso di modifiche alla connettività e sulla capacità di Odyssey di rispettare i requisiti di durata della batteria richiesti. Per questioni di spazio non



(a) Architettura del Media Player

(b) Architettura del Browser Web

(c) Architettura del Visualizzatore di Mappe

(d) Architettura del Riconoscitore Vocale

Figura 3.2: Esempi di applicazioni in Odyssey

riportiamo i risultati dei test, che si sono comunque rivelati molto buoni anche per la durata della batteria, ma ci limitiamo a riportare gli esempi implementati, per dare un'idea dello sviluppo di programmi su Odyssey e degli esempi ben precisi di fidelity legata alle applicazioni.

Video Player In questo caso è stata modificata una applicazione open-source preesistente: *Xanim*. Xanim è un player multimediale per sistemi unix, che permette di visualizzare file locali. L'applicazione è stata divisa in due parti, in modo da realizzarne una versione per file remoti. Il server remoto contiene i file già preparati nelle varie forme di fidelity definite. In questo caso il warden richiede al server remoto un file con la fidelity impostata dall'applicazione, e lo invia ad essa. La struttura è rappresentata in figura 3.2a.

In questo caso sono stati definiti tre livelli di fidelity:

1. Fidelity massima, video a *colori* con frame compressi in JPEG a *qualità 99*.

2. Fidelity media, video sempre a *colori* ma con frame compressi in JPEG a *qualità 50*.

3. Fidelity bassa, video in *bianco e nero*.

Mantenere i file compressi nelle tre versioni causa una occupazione maggiore sul disco del server del 60% rispetto al singolo file di dimensione massima. Con le tecnologie attuali si potrebbe comunque richiedere una codifica *al volo* al server e minimizzare l'utilizzo del disco.

Inoltre l'accesso al video avviene richiedendo i singoli frame. In questo modo il visualizzatore può definire un secondo livello di fidelity, richiedendo un numero di frame al secondo minori rispetto a quelli totali del video. Per finire, un ulteriore grado di fidelity è ottenibile codificando il video a varie risoluzioni, tecnica che si è rivela molto utile soprattutto per il consumo energetico, in quanto diminuisce sensibilmente il carico computazionale del visualizzatore.

Web Browser Nel caso del browser web il grado di fidelity proposto si basa sulla variazione della qualità delle immagini presenti nella pagina:

1. Fidelity massima, *immagine originale* non ulteriormente compressa.

2. Fidelity medio-alta, *immagine compressa* JPEG con *qualià 50*.

3. Fidelity medio-bassa, *immagine compressa* JPEG con *qualià 25*.

4. Fidelity bassa, *immagine compressa* JPEG con *qualià 5*.

Come esempio di browser web è stato utilizzato *Netscape Navigator*. Al tempo il browser non era ancora open-source, perciò le modifiche per rendere l'applicazione compatibile con Odyssey si sono rivelate molto difficili. La comunicazione tra browser e warden è stata possibile attraverso un'ulteriore entità, utilizzata dal browser come proxy web, che si occupa di convertire le richieste HTTP in un formato compatibile col warden e di decidere i livelli di fidelity da utilizzare. Il warden comunica poi con un server remoto, che richiede le pagine da internet e ricomprime le immagini al volo in base alle richieste del warden. L'architettura finale è presentata in figura 3.2b.

Questo esempio dimostra come adattare una applicazione binaria già esistente ad Odyssey può risultare molto difficile. Inoltre questi gradi di fidelity non comportavano modifiche sostanziali nel consumo dell'applicazione, in

quanto il codice del visualizzatore (Netscape Navigator) non era stato modificato, e quindi la sua esecuzione non poteva trarre grosso vantaggio dalla diminuzione di qualità.

Visualizzatore di Mappe In questo caso gli sviluppatori hanno lavorato su un visualizzatore di mappe esistente, *Anvil*, di cui avevano i sorgenti. Il programma richiede una mappa da un server remoto, e la visualizza sul dispositivo. In questo caso l'applicazione (figura 3.2c) può richiedere una mappa con meno dettagli al server oppure ritagliata ad un'area più piccola, per diminuire la dimensione del file scambiato ed aumentare la velocità di disegno sullo schermo del PDA. Vediamo le possibilità di fidelity previste:

1. Fidelity massima, mappa a dimensione intera con tutti i dettagli possibili.
2. Fidelity medio-alta, mappa a dimensione intera ma con la rimozione di alcuni dettagli (strade minori).
3. Fidelity media, mappa a dimensione intera ma con la rimozione di molti dettagli (tutte le strade secondarie).
4. Fidelity medio-bassa, mappa a dimensione ridotta con tutti i dettagli possibili.
5. Fidelity bassa, mappa a dimensione ridotta con la rimozione di alcuni dettagli (strade minori).
6. Fidelity molto bassa, mappa a dimensione ridotta con la rimozione di molti dettagli (tutte le strade secondarie).

Riconoscitore vocale Rispetto alle applicazioni precedenti il riconoscitore vocale lavora solo con dati locali, e non ha bisogno di un collegamento di rete per funzionare. Rispetto ai precedenti, però, non è un semplice visualizzatore di dati ma ha bisogno di effettuare una computazione relativamente intensiva su quelli in ingresso.

I dispositivi wearable (almeno quelli esistenti al tempo) non hanno risorse sufficienti per questo tipo di applicazioni; la soluzione proposta dagli sviluppatori di Odyssey è di affiancare al software di riconoscimento vocale presente sul dispositivo mobile, una copia in esecuzione su un server remoto con adeguate capacità computazionali. In questo caso il dispositivo può chiedere un riconoscimento al server inviando i dati della registrazione. Il focus dell'adattività si sposta quindi nuovamente sulle risorse di rete, che devono essere sufficienti al trasferimento della registrazione verso il server remoto.

In questo caso per diminuire il traffico è possibile effettuare sul client una prima fase di preprocessing della registrazione, che diminuisce di 1/5 la dimensione dei dati da inviare sulla rete, al costo di eseguire una parte della computazione sul client.

Una differente soluzione prevede invece di effettuare il riconoscimento completamente in locale, ma su un vocabolario ristretto, tale da rendere la computazione fattibile anche per il dispositivo mobile. L'architettura ottenuta è in figura 3.2d; le possibilità di fidelity le seguenti:

1. Fidelity massima, esecuzione remota senza nessuna elaborazione da parte del client.

2. Fidelity media, esecuzione remota con preprocessing da parte del client.

3. Fidelity bassa, esecuzione locale con vocabolario ridotto.

Tutte queste possibilità sono valide, soprattutto nell'ottica del risparmio energetico; infatti ridurre l'uso della cpu del client diminuisce il consumo ed aumenta la durata della batteria.

3.2 Aura

Aura[47, 69] si propone come framwork per la creazione di applicazioni pervasive che possano sfruttare in modo ottimale le risorse presenti nell'ambiente, senza dover affrontare i problemi di eterogeneità e variabilità nelle capacità dei dispositivi.

Nel framework di Aura confluiscono i lavori passati del gruppo: *Odyssey*, di cui abbiamo ampiamente parlato sopra, e *Coda*[64, 66], un file system "nomade". Grazie a Coda l'utente più accedere ai propri dati da qualsiasi computer, anche in presenza di collegamenti di rete lenti e non affidabili; implementa perciò una adattività "trasparente" all'applicazione, basata (come per Odyssey) sulla qualità dei dati.

La particolarità di Aura è che i concetti di adattività e context awareness sono spostati ad un livello superiore rispetto a quello comune delle applicazioni. Questo nasce dalla considerazione che in ambienti pervasivi l'eterogeneità di risorse è così alta che una singola applicazione non può essere utilizzata in modo adeguato su tutte le piattaforme. Al contrario, per ogni piattaforma (hardware e software) sono già esistenti applicazioni specifiche che sfruttano appieno le sue caratteristiche. Per fare un esempio, un editor di testo tradizionale ha molte più funzioni di uno per smartphone; quest'ultimo però permette di sfruttare il touchpad del dispositivo mobile.

Se un utente vuole lavorare su un documento probabilmente preferirà utilizzare due applicazioni differenti, che sfruttano tutte le caratteristiche del dispositivo, piuttosto che una unica ma difficile da usare sullo smartphone e con poche funzioni sul desktop.

L'idea di Aura è di definire l'adattività a livello di "Task". Un task è un lavoro che l'utente può voler fare: visualizzare un video, modificare un testo, etc. Per ogni task esisteranno varie applicazioni capaci di eseguirlo: basti pensare a tutti i tipi di editor di testo presenti in commercio o alla moltitudine di media player gratuiti utilizzabili.

Con Aura, in base alle risorse attualmente disponibili all'utente, è possibile decidere quale applicazione utilizzare per un particolare task. A tutti i dettagli sottostanti ci pensa il framework: spostare i file nell'ambiente attuale, convertirli in un formato adatto per l'applicazione corrente, etc.

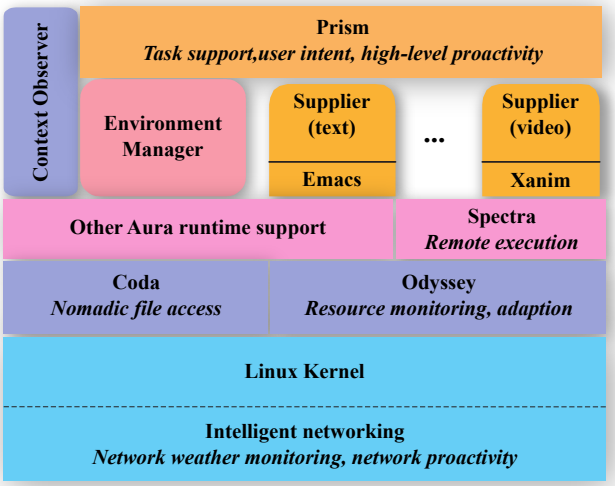
3.2.1 I casi d'uso principali

Vediamo ora due scenari tenuti in considerazione durante lo sviluppo di Aura[47]. Come per Odyssey questi esempi rappresentano una parte significativa dei requisiti considerati durante lo sviluppo del framework.

Immaginiamo un utente in aeroporto, in attesa della chiamata di imbarco per il proprio volo. Ha appena finito di modificare una presentazione e deve inviarla via email ad un collega il prima possibile. La presentazione, però, è molto grande perché piena di immagini, e la rete wireless dell'aeroporto lenta perché in molti stanno navigando su internet. Aura si accorge che, a quella velocità, l'utente non riuscirebbe ad inviare la presentazione completa prima dell'imbarco. Consultando l'elenco dei voli in partenza e la copertura wireless dell'edificio il sistema si accorge che al gate 15 la copertura è buona, la rete libera e non ci sono voli in partenza né in arrivo. Consiglia perciò all'utente di spostarsi al gate 15, per terminare l'invio dell'email prima della chiamata d'imbarco. Analogamente, quando l'invio delle email è quasi terminato, Aura informa l'utente che può nuovamente spostarsi al gate originario in attesa della propria chiamata.

Il secondo scenario vede un utente nel suo ufficio, che sta ultimando le modifiche alla presentazione per un meeting che si terrà tra non molto in un edificio vicino. Non ha ancora terminato il lavoro, ma è già in ritardo; perciò si incammina, portando con se il proprio PDA. Aura si accorge del movimento, e trasferisce la presentazione non ancora terminata sul dispositivo mobile; in questo modo l'utente può completare le ultime modifiche durante il viag-

Figura 3.3: L'architettura di Aura



gio. Attraverso la lista degli appuntamenti si accorge inoltre del meeting, ed inizia a scaricare il software per la presentazione sul terminale della stanza in cui si terrà. Quando l'utente arriva al meeting il sistema ha già provveduto ad accendere il proiettore e a spostare le ultime modifiche sul computer della stanza, in modo che questi riesca ad iniziare la presentazione in orario.

3.2.2 Architettura di Aura

Il concetto chiave di Aura è il *Task*: con esso si definisce una azione ben precisa dell'utente: preparare una presentazione, leggere le email, scrivere un documento, etc. Ad ogni Task sono associati dei programmi (chiamati in seguito supplier) che permettono all'utente di eseguire l'azione, ed i file necessari per completare quel particolare lavoro. All'interno della descrizione del task trovano posto anche le politiche di "conversione" dei file tra differenti supplier.

Aura è un framework molto complesso, e suddiviso in vari livelli. L'architettura è rappresentata in figura 3.3; in essa ritroviamo Coda e Odyssey, che fanno da supporto ad un tool, *Spectra*, di esecuzione remota di applicazioni; questo decide, in base ai dati di contesto, la tecnica di esecuzione remota più adatta nell'ambiente corrente. La parte più interessante sono però i livelli su-

periori: *Environment Manager*, *Supplier*, *Context Observer* e *Task Manager*; vediamoli in dettaglio[69]

- **Task Manager (Prism)**: questo layer, in cooperazione con l'osservatore di contesto, analizza le operazioni dell'utente per capire le sue intenzioni senza richiederle esplicitamente, nell'ottica della *minimal user distraction*; il task manager si accorge, tramite osservazioni sul contesto, di spostamenti dell'utente nello spazio e di cambiamenti nel task eseguito; in base a questo coordina le altre entità per modificare automaticamente l'ambiente senza bisogno di richieste esplicite.

  Un cambio di ambiente, o in generale di task, prevede la ricerca di un nuovo supplier in grado di eseguire il task nell'ambiente corrente, la raccolta dei file necessari e la conversione di questi per il supplier scelto.

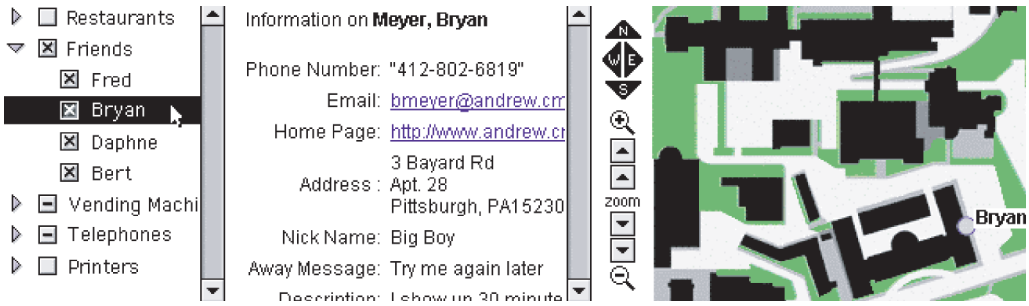
- **Service Suppliers**: sono le applicazioni vere e proprie, che permettono all'utente di eseguire un task. Prerogativa principale di Aura è quella di utilizzare applicazioni preesistenti senza ulteriori modifiche: l'adattività è quindi "trasparente" alle applicazioni, che devono solo essere inserite in un contenitore per fornire al resto del sistema interfacce standard. Tra queste troviamo anche il meccanismo di conversione di file, che si occupa di estrarre le informazioni utilizzabili dall'attuale applicazione e, una volta modificate, reinserirle nel file iniziale in modo da non perdere quelle non trattate da tale programma.

- **Context Observer**: fornisce a Prism tutte le informazioni di contesto acquisite; è un punto chiave della piattaforma, in quanto tante più informazioni offre al task manager, tanto più precisa sarà la previsione di quest'ultimo.

- **Environment Manager**: il componente che si occupa di interagire con l'ambiente: trovare il supplier necessario, distribuire le applicazioni, lanciarne l'esecuzione, etc.

3.2.3 Esempi di applicazioni Aura

Riportiamo ora due applicazioni sviluppate con Aura e descritte in [47]; lo scopo di queste è di supportare la collaborazione all'interno del campus universitario della Carnegie Mellon University.



(a) Screenshot dell'applicazione PHD

(b) Screenshot dell'applicazione Idealink

Figura 3.4: Esempi di applicazioni in Aura

Portable Help Desk PHD è una applicazione *spatial-aware* e *temporal-aware*: conosce l'attuale posizione degli utenti all'interno del campus, ed è collegata con un database degli eventi pubblici o privati in programma.

Portable Help Desk fornisce una mappa della zona in cui si trova l'utente, dove sono rappresentati i colleghi vicini e alcuni servizi, come le stampanti o i distributori automatici. Permette di conoscere l'attuale posizione dei propri amici ed affianca all'interfaccia grafica (in figura 3.4a) un sistema di riconoscimento vocale per interagire con l'applicazione anche in movimento.

Grazie ad Aura PHD offre anche alcuni comportamenti proattivi: ad esempio, se l'utente vuole stampare un documento, PHD può accorgersi se la coda di stampa è troppo lunga e suggerire all'utente una stampante vicina meno affollata, oppure proporre una stampante vicina al luogo di destinazione per un utente in movimento.

Idealink La seconda applicazione presentata è un "ambiente di collaborazione virtuale", e si propone come sostituto elettronico di una meeting room; il cuore di Idealink è una lavagna condivisa in cui tutti gli utenti possono scrivere contemporaneamente, come in figura 3.4b. L'applicazione si integra

con PHD per salvare e caricare le preferenze dell'utente e i propri appuntamenti, e decide automaticamente in quale "meeting virtuale" collegarsi in base ad essi.

Alcuni studi hanno dimostrato che questa applicazione si rivela molto utile, in quanto permette di rendere gli incontri più produttivi ed accorciare i tempi dei meeting.

3.3 CoBrA

CoBrA[27, 28] (Context Broker Architecture) è un framework per applicazioni pervasive sviluppato dall'Università del Maryland a Baltimora, con particolare attenzione verso gli smart spaces. Rispetto ad Aura i ricercatori hanno focalizzato i loro sforzi nella definizione e nel trattamento delle informazioni di contesto. Tra tutti i sistemi analizzati in questo capitolo CoBrA rappresenta perciò quello più evoluto nell'analisi del contesto al fine di inferire nuove informazioni.

L'entità centrale in CoBrA è il *Context Broker* (da cui il nome del framwork), che si occupa di gestire il contesto: acquisire le informazioni dai sensori, ragionare su di esse e fornirle ai dispositivi che le richiedono.

In Cobra il contesto è rappresentato mediante ontologie, che rispetto alle altre tecniche aumenta l'espressività nella definizione delle relazioni tra informazioni e permette inferenze più complesse.
Le ontologie sono espresse con il linguaggio OWL (Web Ontology Language), standard nel campo del Web Semantico. CoBrA offre un insieme di ontologie già pronte, *COBRA-ONT*, che descrivono le informazioni acquisibili in una meeting room intelligente. Lo sviluppatore può utilizzare questo set come base per la propria applicazione, eventualmente espandendolo con altre più complesse legate al particolare dominio applicativo.

3.3.1 Il caso d'uso principale

Riportiamo l'esempio di applicazione riportato in [27], considerato durante lo sviluppo di CoBrA, che tratta una stanza per meeting "intelligente", dotata di proiettore, sensori e vari dispositivi pervasivi.

Un utente entra nella stanza; grazie ad alcuni sensori l'ambiente di accorge della sua entrata, identifica la persona e cerca di capire le sue intenzioni. Si accorge che in quell'orario è in programma una presentazione, e che la persona appena entrata è proprio colui che la deve tenere. Trova tra i file dell'utente

quello per la presentazione, lo invia al proiettore ed avvia il programma per la visualizzazione.

L'utente inizia la presentazione, intanto l'ambiente si accorge che la quantità di luce nella stanza è troppo elevata per il proiettore e spegne alcune lampade.

Finita la presentazione, l'ambiente si accorge della presenza di un PDA e ne identifica il proprietario, colui che ha tenuto la presentazione. Deduce che ha dimenticato il PDA nella stanza, e cerca di localizzarlo all'interno dell'edificio, senza successo. Dalla lista di appuntamenti, però, trova la prenotazione di un volo aereo per il pomeriggio; ipotizza quindi che sia già in viaggio; ricerca il suo numero di cellulare e gli invia un sms per informarlo del PDA perso.

Infine si accorge che nella stanza mancavano alcune persone che avevano segnalato interesse per questa presentazione; per ognuna individua un contatto email ed inoltra una registrazione del meeting.

3.3.2 Architettura di CoBrA

Come già anticipato, il cuore di CoBrA è il *Context Broker*: coordina tutte le entità, che possono cooperare tra di loro attraverso questo componente. L'architettura è quindi come quella rappresentata in figura 3.5, con il context broker al centro del sistema.

Dati i suoi molteplici compiti, questa entità è suddivisa in quattro componenti funzionali:

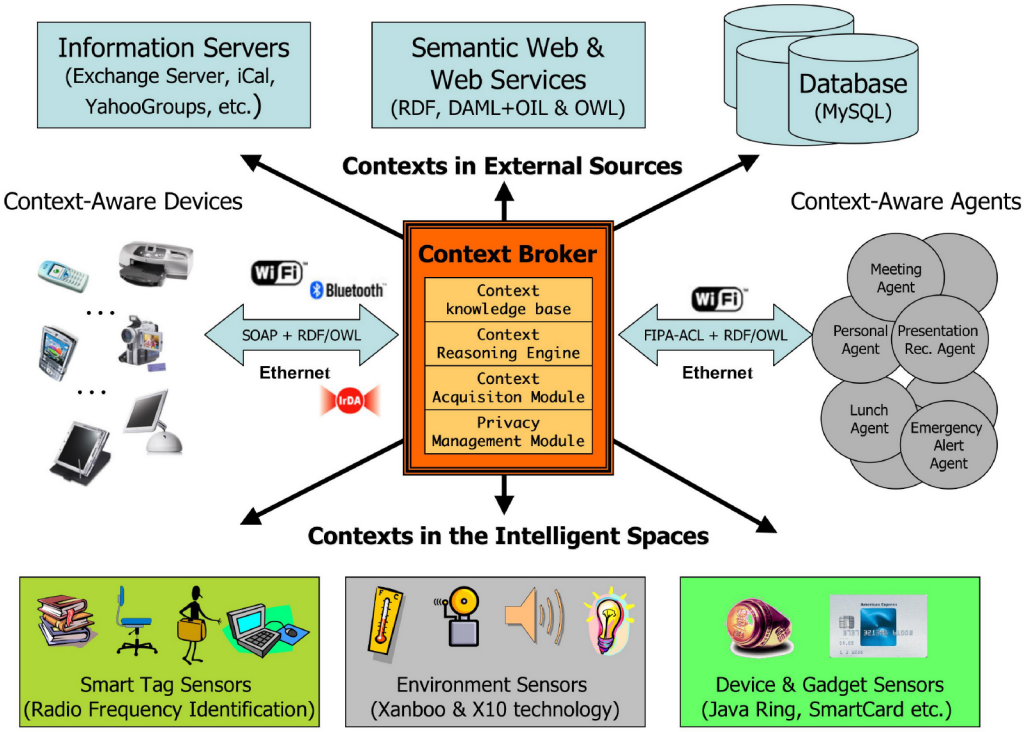
- **Context Knowledge Base**: si occupa di immagazzinare tutta la conoscenza relativa al contesto. In particolare, questa componente gestisce le ontologie che descrivono il contesto e la conoscenza acquisita dai sensori.

- **Context Reasoning Engine**: il motore che analizza le informazioni di contesto per dedurne di nuove. Interpreta quelle presenti nella Knowledge Base, aggrega dati da più fonti, intercetta e rimuove inconsistenze all'interno della KB.

- **Context Acquisition Module**: l'insieme di librerie che si occupano di acquisire informazioni dal contesto: sensori, programmi, etc.

- **Privacy Management Module**: CoBrA presta particolare attenzione alle politiche di sicurezza e gestione della privacy. L'utilizzo di un broker centralizzato favorisce questo tipo di controlli, che sono racchiusi in questo modulo; il suo compito è controllare i permessi associati alle

Figura 3.5: L'architettura di CoBrA



entità che richiedono informazioni al broker ed eventualmente negare l'accesso ai dati.

Il sistema di inferenza della conoscenza, F-OWL, è stato sviluppato appositamente per CoBrA e utilizza come base il motore deduttivo XSB, offrendo caratteristiche migliori rispetto ai comuni sistemi di inferenza per ontologie.

La scelta di design di utilizzare una entità centrale come il context broker ha semplificato molti aspetti, come la creazione di una Knowledge Base, la gestione di privacy e sicurezza, etc. D'altro canto impone grosse limitazioni sulla scalabilità dell'architettura: come sottolineano anche gli stessi creatori, in presenza di spazi molto grandi con un numero elevato di dispositivi (come nell'idea di Weiser) il broker potrebbe diventare un collo di bottiglia e non riuscire a gestire l'intero sistema.

Per questo motivo sono state studiate tecniche per permettere la coesistenza di più broker, che si suddividono lo spazio da gestire e periodicamente aggiornano le proprie Knowledge Base.

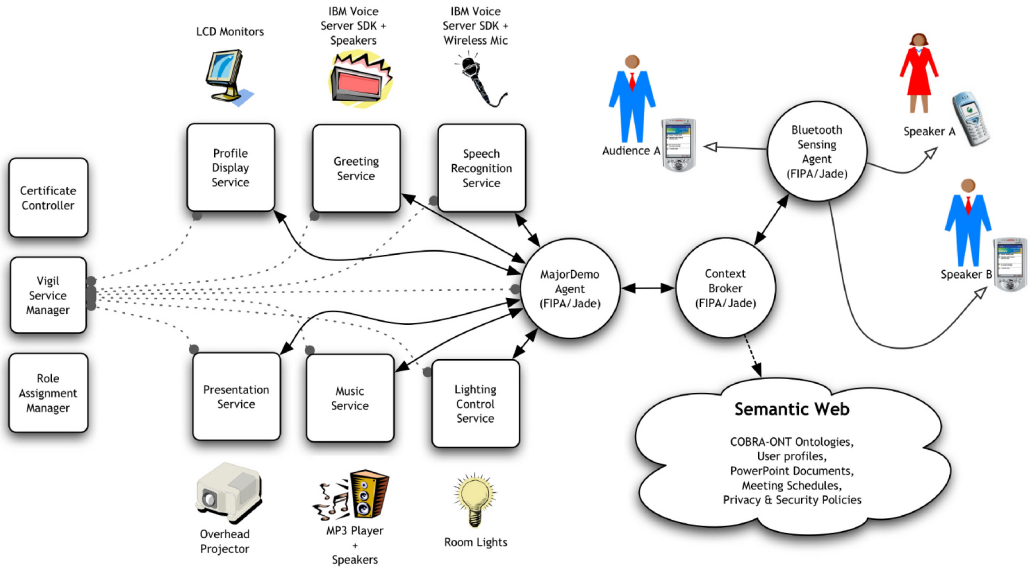


Figura 3.6: Esempio di applicazioni in CoBrA: Interazione tra i componenti di EasyMeeting

Alcuni problemi sorgono comunque in caso di nodi fortemente mobili, che si adattano male ad una architettura così centralizzata.

3.3.3 Esempi di applicazioni CoBrA

Vediamo ora due applicazioni sviluppate in CoBrA e presentate in [27] come esempio di applicazioni context-aware per smart spaces.

EasyMeeting EasyMeeting è una applicazione per gestire smart spaces sviluppata a partire da Vigil[75], una infrastruttura per pervasive computing prodotta sempre nell'Università del Maryland in Baltimora. EasyMeeting porta in Vigil il supporto a context-awareness e protezione della privacy utilizzando CoBrA. L'applicazione fornisce un gruppo di servizi utili per interagire con uno smart space: *riconoscimento vocale*, *visualizzazione di presentazioni*, *controllo della luce*, *riproduzione di musica*, *benvenuto personalizzato*, etc. In figura 3.6 riportiamo lo schema di interazione dei vari componenti. In EasyMeeting il context broker offre un modello di contesto condiviso da tutti i servizi, che si occupa di mantenere informazioni sulla posizione delle persone nella stanza, la programmazione dei meeting e delle presentazioni, gli autori delle presentazioni e altro, utilizzando sensori e informazioni presenti sul

web. Tutte queste informazioni sono utilizzate dal componente MajorDemo per decidere quali servizi abilitare in un particolare momento.

CoBrA Text Messaging Commands CoBrA Text Messaging Commands è una applicazione di controllo tramite SMS integrata col Context Broker. Tramite l'invio di messaggi di testo gli utenti possono interagire col broker, ponendo domande o invocando azioni. Il prototipo corrente di CoBrA permette di ottenere informazioni sui meeting in programma all'eBiquity group dell'UMBC. Le operazioni offerte dall'applicazione sono le seguenti:

- **qMeeting**: il sistema risponde, tramite sms, con la lista di meeting presenti nel programma giornaliero.
- **qSpeaker** [**meeting-id**]: il sistema invia all'utente alcune informazioni utili sul presentatore di un particolare meeting, ad esempio nome, istituzione di provenienza, titolo professionale, ambiti di ricerca etc.
- **qInfo** [**meeting-id**]: il sistema offre all'utente informazioni sul meeting, come luogo ed ora della presentazione, breve descrizione, etc.
- **qFollowup** [**meeting-id**] [**email-address**]: con questa operazione l'utente richiede l'invio dei documenti associati al meeting ad un indirizzo di posta elettronica, per agevolare chi non ha potuto essere presente.

3.4 CORTEX

Vediamo ora CORTEX (CO-operating Real-time senTient objects: architecture and EXperimental evaluation)[24, 80, 84], prodotto di un omonimo progetto europeo che ha visto coinvolte, tra le altre, le università di Lancaster e di Dublino.

Come i precedenti sistemi, anche CORTEX si propone come modello di programmazione per applicazioni pervasive context-aware. In questo caso però non si parla necessariamente di smart spaces, ma anche di applicazioni con mobilità maggiore, come veicoli intelligenti e sistemi di gestione del traffico.

In questi casi particolari non è plausibile la presenza di un punto di centralizzazione, si hanno problematiche stringenti sul tempo di reazione e si deve assicurare un certo grado di sicurezza. Questi requisiti hanno portato allo sviluppo di un modello di programmazione delle applicazioni che differisce sensibilmente da quelli precedenti, e prende il nome di *Sentient Object Programming Model*.

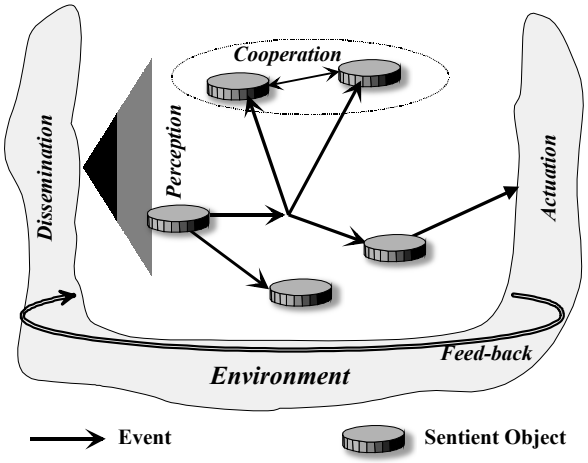


Figura 3.7: Eventi ed interazioni tra oggetti in CORTEX

Una applicazione in CORTEX è descritta da un insieme di entità cooperanti, chiamate *Sentient Object*[24]. Un oggetto "senziente" è a sua volta un piccolo sistema context-aware che può interagire con l'esterno tramite sensori ed attuatori; prevede meccanismi di analisi dei dati acquisiti, ha una propria conoscenza del contesto e possiede anche un motore di inferenza per dedurre nuove informazioni. Infine, un sentient object può essere collegato ad altri per realizzare forme di cooperazione basate su scambio di eventi. In figura 3.7 sono illustrate le modalità che permettono la cooperazione tra sentient object: attraverso uno scambio di eventi o attraverso modifiche all'ambiente che vengono catturate da sensori.

3.4.1 Scenari e casi d'uso principali

Riportiamo due esempi di possibili applicazioni CORTEX, descritti in [80]. Questa volta non si tratta di meeting room, ma di situazioni più complesse e con caratteristiche tipiche dei sistemi real time.

Sistema di aiuto per il soccorso alpino Nell'ambiente del soccorso alpino, i soccorritori sono costantemente impegnati nella ricerca e nel salvataggio di persone bloccate, e in alcuni casi anche ferite, in condizioni estremamente pericolose. In questo ambito anche i dispositivi mobili devono rispettare requisiti stringenti, soprattutto in termini di peso e durata della batteria. Una conoscenza della posizione dei singoli soccorritori può permettere una migliore pianificazione da parte del centro che coordina la ricerca e rappresenta una sicurezza maggiore per queste persone, poste di fronte ad una

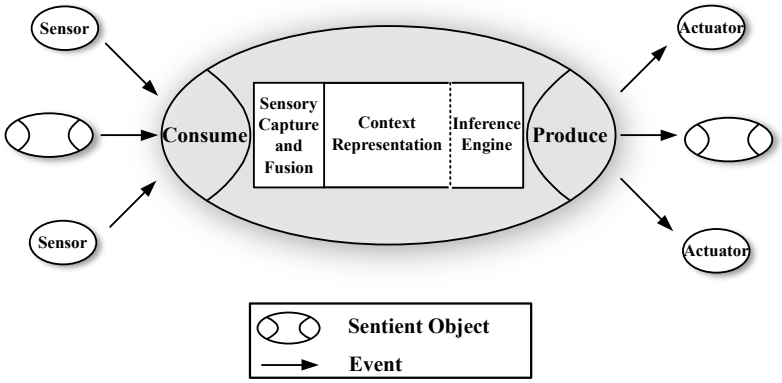


Figura 3.8: Il modello di un sentient object

situazione di pericolo costante. In questo scenario sono necessarie tecniche di cooperazione tra i dispositivi dei soccorritori per sopperire alla mancanza di sistemi di comunicazione affidabili sul luogo della ricerca, e possibilmente tecniche che offrano una certa "intelligenza" ai dispositivi, per aiutare i singoli soccorritori.

Auto intelligenti In questo scenario si ipotizza la presenza di sistemi automatizzati sulle auto per evitare situazioni di grave pericolo. Ad esempio, nel caso di un incidente su una autostrada, la circolazione rimarrà bloccata per molto tempo. In questa situazione sono molto probabili tamponamenti a catena a causa dell'improvviso rallentamento sulla corsia di marcia. Un sistema di comunicazione tra automobili permette, ad esempio, la notifica dell'improvvisa frenata di una macchina a quelle vicine, che potrebbero decidere di frenare autonomamente, eludendo i controlli del guidatore. In altre situazioni, le auto potrebbero notificare condizioni di pericolo, come un tratto di strada allagato, per aiutare l'utente nella guida.

3.4.2 Architettura di CORTEX

In CORTEX, a differenza dei precedenti modelli, non esiste un'infrastruttura centrale che gestisce le applicazioni, ma un insieme di *sentient object* indipendenti che cooperano tra loro. L'ambiente fornisce gli strumenti per programmare questi oggetti e per permettere la loro comunicazione verso l'esterno: sensori, attuatori e altri sentient object.

Le comunicazioni sono modellate come *eventi*, sia per i dati che arrivano dai sensori, sia per le richieste inviate agli attuatori, sia per la comunicazione tra oggetti.

In figura 3.7 abbiamo già visto lo schema di una applicazione, con le interazioni tra oggetti, mentre nella 3.8 riportiamo la struttura di un sentient object. Come si può vedere dall'immagine un oggetto CORTEX contiene le entità base di una applicazione context-aware: un sistema per acquisire dati dai sensori, una rappresentazione del contesto e un motore di inferenza. Vediamo meglio in dettaglio queste componenti:

- **Sensory Capture**: questo modulo si occupa principalmente di prendere i dati provenienti dall'esterno ed effettuare operazioni di *sensor fusion*, per risolvere i problemi legati agli errori commessi dai sensori e derivare informazioni a più alto livello da fonti di dati multiple. Utilizza uno schema probabilistico basato su reti Bayesiane. In questo modo si può, ad esempio, modellare l'incertezza dei dati acquisiti, le dipendenze tra gruppi di sensori e dedurre la veridicità di fatti.

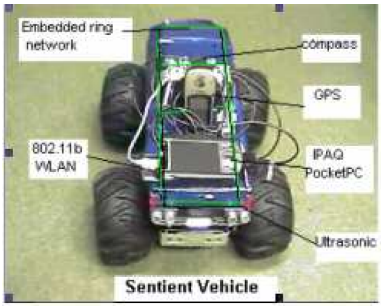
- **Context Hierarchy**: definisce il contesto in cui si trova l'oggetto. Il contesto è modellato da un insieme di fatti, rappresentati gerarchicamente secondo il paradigma *Context-based Reasoning* presentato in [50]. Gestire il contesto gerarchicamente permette di limitare il ragionamento ad un sottoinsieme delle regole e delle azioni possibili, aumentando l'efficienza del sistema di inferenza e decisione.

- **Inference Engine**: si occupa di modificare il comportamento dell'oggetto in base ai dati di contesto; le regole, definibili dall'utente, sono espresse con linguaggio CLIPS (C Language Integrated Production System), che permette di inferire fatti a partire da un insieme di regole; il linguaggio è molto espressivo, permette di definire relazioni di eventi molto complicate e offre la possibilità di integrazione con linguaggi procedurali come C++ e Java.

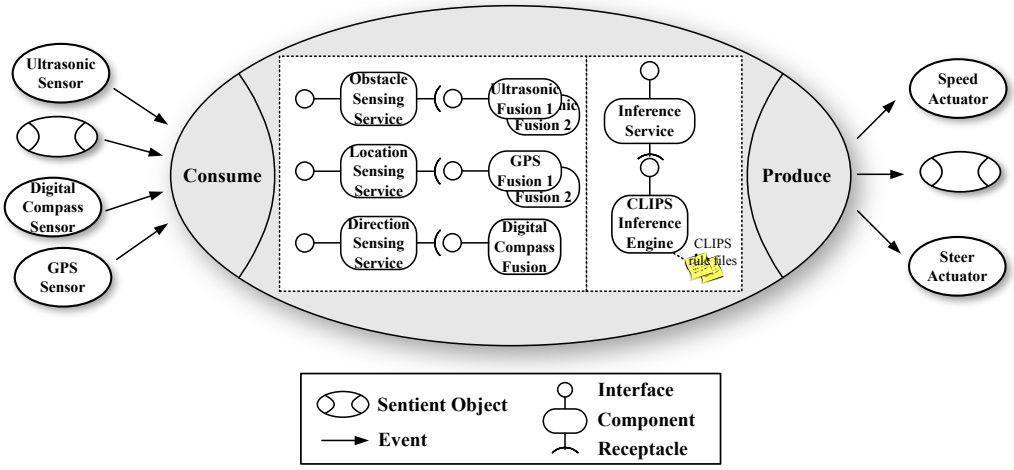
Infine, una importante caratteristica di CORTEX è la dinamicità dei collegamenti dei sentient object. Questi non sono infatti statici, ma permettono un continuo ricambio di produttore/consumatore: la definizione di comunicazioni ad eventi fa si che un oggetto possa ricevere un evento di un certo tipo da qualsiasi entità a lui vicina, sia essa sensore, attuatore o altro oggetto. In questo modo si favorisce la programmazione di applicazioni anche in casi di alta mobilità dei dispositivi.

3.4.3 Un esempio di applicazione CORTEX

Riportiamo ora il risultato della sperimentazione di CORTEX: lo studio di un veicolo intelligente, capace di raggiungere destinazioni prestabilite ed interagire con gli altri veicoli incontrati.



(a) Il test-bed dell'applicazione



(b) Il sentient object che guida un'auto

Figura 3.9: L'esempio di applicazione CORTEX: un sistema di guida automatica

Gli obiettivi dell'auto sono impostati attraverso un "circuito virtuale", composto da una serie di coordinate GPS da raggiungere in sequenza. Ogni veicolo costruisce una rappresentazione dell'ambiente che lo circonda per decidere le mosse da effettuare. La cooperazione tra veicoli è necessaria per evitare scontri, oppure per seguire il percorso di altre auto.

L'applicazione è in sviluppo su un test-bed di auto radiocomandate, modificate per muoversi autonomamente nello spazio. Sono dotate di ricevitore GPS e sensori ad ultrasuoni per individuare oggetti vicini ed utilizzano due schede di rete wireless per comunicare con le altre auto presenti nello spazio.

In figura 3.9a è riportata una foto dell'auto prototipo, mentre nella 3.9b lo schema del sentient object che la governa.

Questa applicazione è dotata di modelli di data-fusion sui dati dei sensori ad ultrasuoni, per dedurre la presenza, e la relativa distanza, di ostacoli vicini;

questa informazione viene utilizzata per definire il contesto attuale, mentre il sistema di inferenza definisce la velocità dell'auto e controlla lo sterzo.

3.5 MB++

Terminiamo l'analisi con *MB++*[56], framework per applicazioni context-aware che cerca di portare caratteristiche tipiche dell'High Performance Computing anche in sistemi pervasivi.

Gli autori sottolineano come, al momento, non esistano modelli di sviluppo per applicazioni pervasive con elevate richieste computazionali. Questo problema accomuna anche tutti i modelli fin qui studiati, dove sia gli approcci "centralizzati" come CoBrA, sia quelli molto distribuiti come CORTEX, non considerano la presenza di piattaforme ad alte prestazioni che offrano le proprie capacità computazionali. MB++ si propone perciò come ambiente di programmazione per quelle applicazioni pervasive che hanno bisogno di effettuare elaborazioni significative sui dati ottenuti dal contesto, offrendo una infrastruttura che permetta ai dispositivi pervasivi di entrare ed uscire dinamicamente dal sistema, e richiedere l'esecuzione di parti dell'applicazione su risorse HPC.

Le applicazioni possono utilizzare le risorse computazionali del server mediante la definizione di "trasformazioni". Una trasformazione è un algoritmo computazionalmente significativo che lavora su uno o più stream di dati, producendo uno stream di risultati. Questa trasformazione può essere dovuta ad operazioni sui dati acquisiti, come algoritmi di data fusion, oppure causata da problemi di interoperabilità tra applicazioni, che lavorano su dati analoghi ma in formato differente (si pensi alla moltitudine di formati audio/video), un po' come in Aura.

In MB++, quindi, una applicazione è definita come composizione di un grafo data-flow di trasformazioni su stream, contenenti dati prodotti da dispositivi pervasivi o da precedenti elaborazioni; il risultato del grafo viene inviato, sempre tramite stream, al visualizzatore in esecuzione su dispositivi mobili. La struttura di una applicazione pervasiva è riportata in figura 3.10

3.5.1 Il caso d'uso principale

Vediamo un esempio, tratto da [56], di applicazione pervasiva che potrebbe trarre vantaggio dall'utilizzo di un ambiente come MB++.

Lo scenario proposto è un sistema di risposta alle emergenze di un'area metropolitana, che ottiene informazioni da telecamere per il controllo del

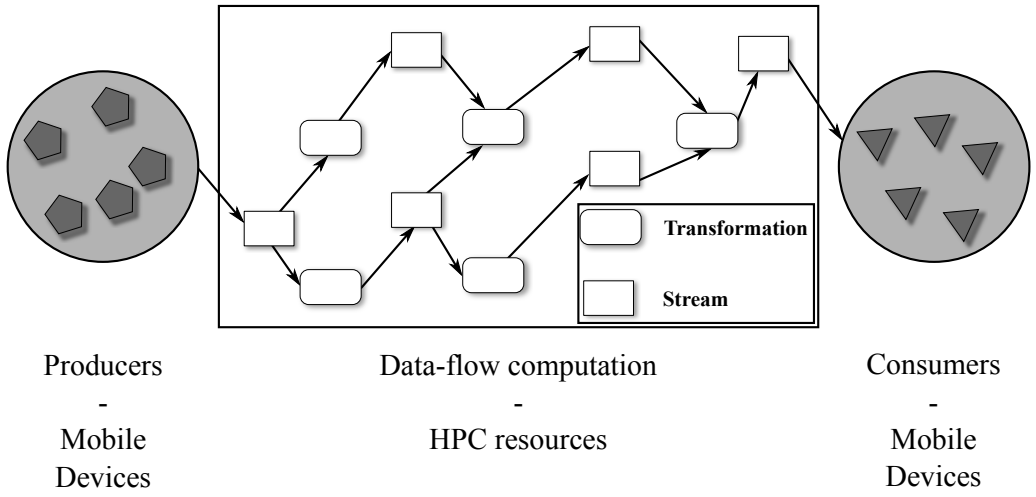


Figura 3.10: La struttura di una applicazione MB++

traffico, dispositivi mobili della polizia locale, allarmi antincendio e contro le intrusioni, etc.

Supponiamo che un ladro entri in un negozio per una rapina, facendo scattare l'allarme. Il comando della polizia è avvisato immediatamente, e il poliziotto che gestisce l'emergenza, attraverso il proprio palmare, richiede i video registrati nella zona dell'infrazione che possono aver inquadrato il criminale. Questo servizio richiede algoritmi specifici, ad esempio di riconoscimento di movimento per identificare la posizione del ladro all'interno dell'edificio; il PDA dell'utente interroga il sistema, e fornisce gli eventuali algoritmi non ancora presenti. Vista la natura della richiesta, il sistema individua una risorsa HPC adeguata ed esegue l'applicazione su di essa. Una volta prodotti i risultati, questi saranno inviati al palmare del poliziotto, che potrà visualizzarli durante lo spostamento verso il luogo del crimine.

3.5.2 Architettura di MB++

Vediamo ora in dettaglio l'architettura proposta in MB++ per la comunicazione tra i dispositivi pervasivi e le risorse HPC. Il sistema (illustrato in figura 3.11 è suddiviso in tre entità principali:

- **Type Server**: il componente che si occupa di gestire le definizioni dei tipi di dati e le trasformazioni definite su di essi. Queste vengono fornite dinamicamente, al momento dell'uso, dalle applicazioni in esecuzione sui dispositivi pervasivi.

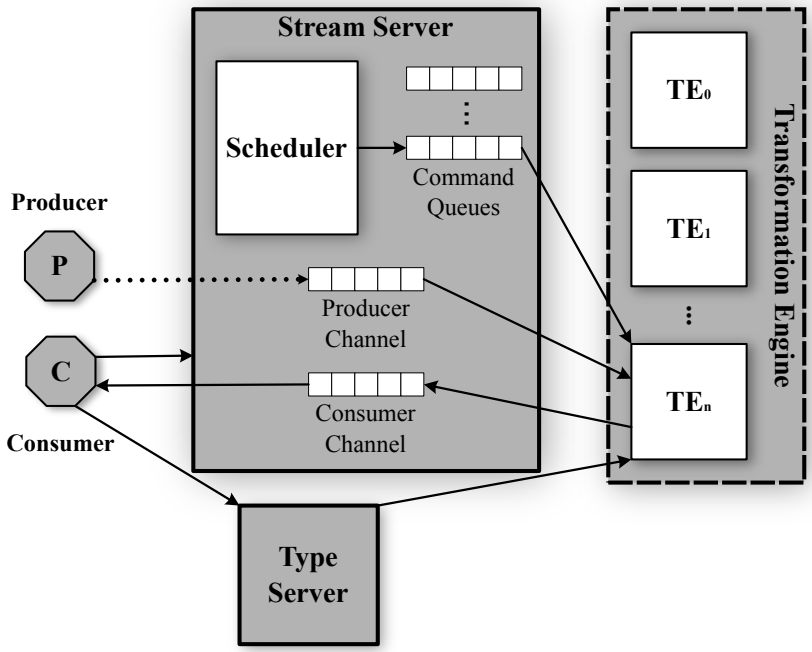


Figura 3.11: Architettura del framework MB++

- **Stream Server**: ha il compito di gestire gli stream; è il componente che contiene le definizioni degli stessi e il grafo dataflow di trasformazioni da eseguire; contiene fisicamente gli stream e, tramite lo scheduler, gestisce le politiche di assegnamento delle risorse ai singoli grafi.

- **Transformation Engine**: l'entità che si occupa dell'esecuzione effettiva delle trasformazioni. Esiste un TE per ogni risorsa HPC presente nel sistema, e la sua esecuzione è guidata dallo stream server, che alloca parti di un grafo data-flow su di esso; il codice da eseguire per la trasformazione viene invece fornito dal Type Server.

MB++, considera scenari con un numero elevato di grafi data-flow attivi contemporaneamente, richiesti da differenti applicazioni pervasive; in questo ambiente si giustifica ulteriormente l'utilizzo di risorse HPC per l'esecuzione parallela di tutte le richieste.

Dal punto di vista del calcolo ad alte prestazioni, la modellazione della computazione come data flow permette di eseguire la catena di trasformazioni in parallelo. L'assegnazione di queste ai TE avviene però staticamente, al momento dell'allocazione del grafo, e quindi non si ottiene un naturale bilanciamento del carico tipico di soluzioni come Muskel[9].

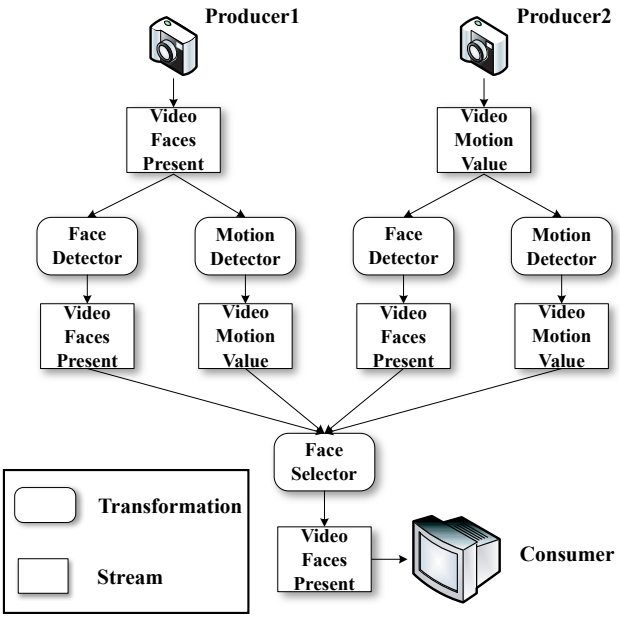


Figura 3.12: Struttura dell'applicazione di riconoscimento di intrusi in MB++

Il focus di MB++ è di fornire un ambiente di calcolo ad alte prestazioni per dispositivi mobili e pervasivi; rispetto agli altri modelli non si concentra però sullo sviluppo di applicazioni context-aware o adattive: mancano completamente modelli per definire il contesto e per le interazioni tra dispositivi pervasivi, se non quelle che passano dallo stream server.

3.5.3 Esempio di applicazione MB++

Riportiamo l'esempio utilizzato in [56] per dimostrare la bontà dell'ambiente. Si tratta di una applicazione di analisi dei dati ricevuti da un insieme di videocamere, alla ricerca di persone in movimento. Rappresenta una evoluzione dei classici sensori di movimento che oltre ad individuare intrusi in un ambiente, ne visualizza anche il volto.

Nel caso proposto abbiamo due fonti che producono video in streaming; questi dati vengono analizzati da due differenti algoritmi, uno per il riconoscimento di volti, l'altro per la rilevazione di movimento. Questi dati vengono inviati ad una terza trasformazione, che analizzando la coppia di valori stabilisce se è presente un intruso nell'ambiente ed invia il volto riconosciuto

al consumatore dello stream, che lo visualizza su un monitor. Lo schema dell'applicazione è illustrato in figura 3.12.

I test di questa applicazione hanno mostrato la scalabilità del sistema, aumentando il numero di grafi data-flow in esecuzione (più istanze della applicazione) fino ad occupare completamente le risorse fornite dai Transformation Engine.

3.6 Conclusioni

Torniamo ora all'esempio di applicazione HPPC riportato nella sezione 2.5.2, ovvero un supporto alla gestione delle emergenze. In questo caso abbiamo bisogno di molte risorse computazionali, da utilizzare non solo durante una emergenza, ma anche nella fase, continua, di previsione e prevenzione.

Tra tutti i modelli analizzati fino ad ora, l'unico che prevede l'utilizzo di risorse HPC è *MB++*. La strutturazione a stream si rivela, inoltre, adatta agli algoritmi di data-mining e di simulazione tipici per la gestione delle emergenze. In questo modello, però, mancano i meccanismi di adattività per trasportare le computazioni sull'insieme di nodi disponibili. Inoltre non vengono forniti strumenti adatti allo sviluppo della parte "pervasiva" dell'applicazione, ovvero dei sensori che acquisiscono i dati ambientali e di tutti i dispositivi mobili utilizzati durante la gestione di una emergenza.

Al contrario i modelli come Aura, CoBrA e CORTEX permettono di sviluppare in modo efficiente le applicazioni da eseguire sulle risorse pervasive, ma non offrono una integrazione con risorse HPC per le elaborazioni intensive. Inoltre nessuno di questi sistemi supporta ambienti con nodi distribuiti in un'area molto estesa, come il letto di un fiume oppure una foresta.

Dobbiamo comunque considerare che i modelli fino ad ora sviluppati si basano sulle idee originarie di Weiser sul pervasive computing ([82]): ambienti intelligenti (soprattutto edifici) con buona parte dei dispositivi pervasivi allocati staticamente, ed utilizzati per eseguire applicazioni "leggere": visualizzatori o editor di file, registratori, etc. Nel nostro caso stiamo parlando di applicazioni che richiedono simulazioni molto complesse, eseguite in ambienti pervasivi creati magari solo nel momento di una emergenza e composti quasi esclusivamente da dispositivi mobili,

Si rende perciò necessario lo studio di un nuovo modello di programmazione, che permetta la *cooperazione* delle risorse pervasive con quelle ad alte prestazioni e l'utilizzo di adeguate connessioni di rete per far comunicare i vari dispositivi in ambienti molto vasti. A questo proposito, sottolineiamo inoltre che tipicamente le risorse HPC non sono dislocate vicino all'area di

emergenza, ma si troveranno a distanze molto elevate. Si rende quindi necessario l'utilizzo di connessioni ad alta velocità per collegare i centri di calcolo all'area della possibile emergenza, e tecnologie per sfruttare al meglio queste reti.

Nonostante questo, gli ambienti visti in questo capitolo presentano caratteristiche degne di nota ed idee riutilizzabili all'interno di questo tipo di ambiente. Ad esempio:

- La modellazione delle informazioni del contesto tramite ontologie, come proposto da CoBrA, permette l'utilizzo di potenti strumenti di inferenza che possono rivelarsi utili per lo sviluppo delle applicazioni da eseguire sui dispositivi mobili utilizzati durante l'emergenza, soprattutto nei casi di bassa connettività con le risorse HPC, per aiutare gli utenti (soccorritori, forze dell'ordine, etc) anche senza il supporto del sistema previsionale e di supporto alle decisioni.

- Tecnologie come quelle utilizzate in *Aura* potrebbero permettere l'utilizzo di applicazioni già sviluppate e la cooperazione con sistemi preesistenti, con un possibile risparmio sui tempi di sviluppo, ma soprattutto la possibilità di realizzare un sistema "estendibile", che possa inglobare nuove tipologie di risorse e di applicazioni, in base all'ambiente in cui si trova.

- Anche le caratteristiche di proattività del Task Manager di *Aura* sono molto interessanti, ed il supporto potrebbe fornire strumenti per aiutare le applicazioni context-aware nella loro definizione.

- Le tecniche di riduzione della fidelity introdotte in *Odyssey* sono sicuramente promettenti, e necessitano uno studio accurato anche nel nostro caso, soprattutto a causa dei problemi di affidabilità e di velocità delle connessioni tra ambiente pervasivo e ambiente HPC. In questa applicazione si rivela necessario un approccio a qualità dei dati - e delle computazioni - variabile, ed il meccanismo della fidelity introdotto in questo modello sembra adattarsi perfettamente al nostro caso.

- L'approccio di *CORTEX* nella realizzazione di più entità al loro interno context-aware ed adattive, che cooperano per realizzare l'intera applicazione, sembra un buon metodo per risolvere i problemi di complessità di queste applicazioni, e al tempo stesso ottenere un comportamento adattivo per l'intero sistema.

- La definizione, sempre di *CORTEX*, di un ambiente non centralizzato si rivela sicuramente vincente nel nostro caso, dove ogni dispositivo

deve essere in grado di funzionare anche con connettività assente, ma al tempo stesso sfruttare al massimo le risorse eventualmente presenti.

- La modellazione riportata in *MB++*, in cui i dispositivi pervasivi agiscono da produttori e consumatori di dati nei confronti delle risorse HPC si rivela molto valida per le operazioni di previsione e anche nei casi di emergenza, se in presenza di un collegamento di rete adeguato. Vogliamo però inserire anche meccanismi aggiuntivi, per poter eseguire le computazioni intensive su tutti i dispositivi presenti nella rete.

L'analisi di questi modelli ha perciò dato una serie di spunti e di idee per la definizione di un ambiente per lo sviluppo di applicazioni pervasive e ad alte prestazioni. Vedremo più avanti come alcune di queste idee costituiscono la base del modello di programmazione di ASSISTANT.

In definitiva in questo capitolo abbiamo visto come il *pervasive computing* sia un settore di ricerca ancora giovane, con tante idee e soluzioni promettenti ma ancora nessun ambiente di sviluppo abbastanza evoluto per la realizzazione di una applicazione pervasiva realmente complessa, tantomeno una applicazione HPPC. È importante capire come il problema degli strumenti presentati non sia di tipo implementativo, ma *architetturale*: i modelli fin qui visti non raggiungono quel livello di espressività richiesto in applicazioni non banali, come quella da noi proposta di gestione delle emergenze.

Da questi modelli emergono perciò le limitazioni di uno studio guidato da esempi, che porta allo sviluppo di strumenti che si adattano male ad applicazioni significativamente differenti da quelle studiate. Nei capitoli successivi vedremo invece come ASSISTANT nasca proprio nell'idea di offrire un modello di programmazione il più generale possibile, partendo da basi solide derivate da un settore, quello dell'HPC, molto più sviluppato.

Capitolo 4

ASSIST: un ambiente di programmazione per applicazioni parallele e distribuite ad alte prestazioni

Con questo capitolo analizziamo il secondo campo di ricerca che converge nell'HPPC, quello delle applicazioni parallele ad alte prestazioni. Come nel capitolo 3 vogliamo illustrare l'attuale stato dell'arte in questo settore, in modo da fornire al lettore le basi per comprendere le idee che hanno portato allo sviluppo di ASSISTANT, e come queste hanno guidato la definizione di tale modello.

Il settore delle applicazioni parallele ad alte prestazioni nasce molto prima rispetto al pervasive computing ed ha visto l'interesse di molte aziende importanti; per questo motivo la ricerca è ormai ad uno stato molto più maturo, ed esistono tanti ambienti di programmazione, profondamente diversi tra loro, per lo sviluppo di applicazioni parallele.

All'interno dell'Università di Pisa si studiano, oramai da anni, modelli per lo sviluppo di applicazioni parallele "ad alto livello", ritenuti da molti una tecnica vincente per governare la sempre più alta complessità delle applicazioni e delle piattaforme hardware e software[16, 31].

Questo interesse ha portato nel tempo alla definizione ed implementazione di vari modelli di programmazione: P^3L[15] (Pisa Parallel Programming Language), *SkIE*[16] (a Skeleton Integrated Environment), *Lithium*[35] e per finire *ASSIST* [78] (A Software development System based on Integrated Skeleton Technology). I primi sistemi sviluppati prevedevano un approccio "a skeleton" classico, in stile Cole[31], che col tempo si è evoluto in un modello di *programmazione parallela strutturata* più generale, per sopperire alla

difficoltà di esprimere molte computazioni parallele significanti (ad esempio data-parallel con stencil) tramite gli skeleton.

In questo capitolo parleremo di ASSIST, da noi considerato lo stato dell'arte per gli ambienti di programmazione parallela, in quanto unisce alle caratteristiche tipiche degli skeleton (facilità di programmazione, possibilità di ottimizzazioni automatiche e modelli dei costi noti) una maggiore generalità che lo rende adatto ad implementare molti algoritmi paralleli difficilmente esprimibili tramite skeleton.

Inoltre la generalità del modello ha permesso di introdurre delle prime forme di *dinamicità* nell'esecuzione[79], al fine di sfruttare nel miglior modo possibile le risorse o di mantenere un livello di prestazioni minimo richiesto; questi meccanismi, nell'ottica della classificazione dei sistemi pervasivi, finiscono sotto la definizione di *adattività*, e possono essere considerati il primo passo per un modello per lo sviluppo di applicazioni pervasive.

Per questo motivo il modello di ASSIST è stato utilizzato come base per la definizione di ASSISTANT, che in un certo senso rappresenta la sua evoluzione.

Nel seguito, il capitolo è strutturato come segue: dopo una introduzione ai modelli di programmazione parallela strutturati (4.1), descriveremo ASSIST: le idee che hanno portato alla sua definizione (sezione 4.2), il modello di programmazione e la sua attuale implementazione (rispettivamente 4.3 e 4.4).

4.1 La Programmazione Parallela Strutturata

Storicamente, le due tecniche principali per realizzare programmi paralleli sono state l'uso di memorie condivise o sistemi a scambio di messaggi. Questi metodi erano strettamente legati all'hardware delle macchine parallele, che potevano supportare uno o l'altro formalismo nativamente. La modalità di programmazione non veniva quindi scelta dal programmatore, ma era definita dalla piattaforma su cui il programma veniva sviluppato. Chiaramente un cambiamento della piattaforma hardware rendeva *necessarie* profonde modifiche al codice, soprattutto nel caso di passaggio da un formalismo all'altro.

Col tempo gli sviluppi hardware hanno portato a piattaforme in grado di emulare reciprocamente i meccanismi senza incorrere in overhead significativi: piattaforme a memoria condivisa possono offrire una astrazione di scambio di messaggi e, analogamente, quelle a scambio di messaggi possono realizzare una memoria condivisa distribuita.

Attualmente si trovano molti modelli di programmazione parallela di questo tipo; i più conosciuti e supportati sono *OpenMP*[33] (Open Multi-Processing), che offre una astrazione a memoria condivisa, e *MPI*[44, 51] (Message Passing Interface), a scambio di messaggi. Abbiamo utilizzato il nome modelli, e non linguaggi, perché tipicamente sono API implementate su più linguaggi (C, C++ e Fortran) attraverso l'uso di librerie e/o direttive per i compilatori. Come abbiamo detto prima sono indipendenti dall'architettura sottostante, ed esistono versioni di OpenMP per cluster e di MPI per macchine a memoria condivisa.

Nonostante questa forma di "portabilità" offerta dalle piattaforme, il passaggio ad architetture differenti comporta, nel migliore dei casi, la necessità di una revisione del codice sorgente per sfruttare al massimo le caratteristiche della nuova piattaforma.

Nella progettazione di una applicazione parallela, infatti, ci troviamo spesso di fronte ad un differente significato di portabilità: se in un programma sequenziale si usa per indicare la possibilità di utilizzare codice scritto per altre piattaforme senza modifiche (almeno non sostanziali), nella versione parallela non ci si accontenta che il programma "funzioni", ma siamo interessati alla "portabilità delle prestazioni", ovvero a sfruttare al meglio le differenti piattaforme.

Oltre ai problemi di portabilità queste librerie rendono i programmi complessi da realizzare, in quanto richiedono al programmatore di gestire esplicitamente i meccanismi di cooperazione delle entità che compongono l'applicazione parallela.

Ci troviamo in una situazione analoga a quando, nella programmazione sequenziale, non esistevano i linguaggi ad alto livello e i programmatori passavano molto tempo a scrivere codice particolare per la piattaforma utilizzata: in quel caso una modifica della piattaforma poteva avere risultati catastrofici. Come nei programmi sequenziali sono stati introdotti linguaggi ad alto livello, che offrono dei livelli di astrazione e lasciano ai tool di sviluppo la creazione di codice ottimizzato per i vari sistemi, così si vorrebbe fare anche per la programmazione parallela, ed offrire strumenti per definire le applicazioni parallele ad "alto livello" che permettano ottimizzazioni automatiche del codice.

La maggiore espressione di questo concetto prende il nome di *programmazione parallela strutturata*[31], e offre astrazioni per costruire le applicazioni tramite la composizione di pattern paralleli di base.

Con questo non vogliamo sminuire l'importanza dei modelli come MPI o OpenMP, ma semplicemente chiarire che, come in tutte le strutturazioni a livelli, e al pari dei linguaggi macchina, questi debbano rappresentare il *supporto* per dei modelli di programmazione di più alto livello.

Tra i vari modelli per la programmazione parallela strutturata, uno dei più interessanti è sicuramente il modello a *skeleton*. Uno skeleton[16, 31, 77] astrae paradigmi di computazione comuni nella programmazione parallela, le cosiddette *forme di parallelismo*, come pipeline e farm (parallelismo su stream) o divide et impera, map e map&reduce (parallelismo sui dati). Nel modello a skeleton il programma viene descritto tramite una definizione della sua *struttura parallela*, ovvero la/le forme di parallelismo che utilizza (lo scheletro dell'applicazione) e la scrittura del codice *sequenziale* dell'applicazione (il contenuto).

Rispetto ai modelli a basso livello, nei quali il codice sequenziale (che chiameremo spesso anche "codice utente") si mischiava alle primitive per la cooperazione, abbiamo ora una netta separazione, che permette di modificare uno dei due senza toccare l'altro.

Una caratteristica molto interessante degli skeleton, derivata dall'uso di forme di parallelismo note, è che si conoscono i pattern di comunicazione tra le entità dell'applicazione. Questo permette di ricavare un modello dei costi che, unito alla conoscenza della piattaforma di esecuzione, offre una stima del comportamento dell'applicazione.

Ma i vantaggi dati da una conoscenza della struttura parallela non si fermano alla caratterizzazione dei modelli di costo. È stato dimostrato infatti che, grazie a queste informazioni, si possono supportare in modo *efficiente* e *trasparente all'applicazione* tolleranza ai fallimenti[20, 23] e adattività[4, 79].

Questo permette al programmatore di non curarsi di aspetti sempre più importanti nel mondo delle applicazioni parallele ad alte prestazioni, dove l'introduzione del *Grid Computing*[46] ha promosso piattaforme eterogenee e fortemente dinamiche. In questi ambienti non sono utilizzabili modelli come MPI o OpenMP, ma ci troviamo di fronte a strumenti specifici come ad esempio Globus[45]; l'elevata dinamicità della piattaforma richiede inoltre applicazioni in grado di sopportare fallimenti dei nodi e riconfigurazioni.

Grazie agli skeleton il programmatore non deve considerare molti degli aspetti inerenti alla gestione della griglia, e può lasciare agli strumenti di programmazione questi compiti. Questa malleabilità degli skeleton ha portato alla definizione di ambienti, come appunto ASSIST, utilizzabili per sviluppare applicazioni parallele indipendenti dalla piattaforma di esecuzione, portabili sia su ambienti grid che su cluster o macchine SMP senza ulteriori modifiche[6, 8].

Esistono anche altri modelli di programmazione parallela strutturata oltre a quello a skeleton; ad esempio i linguaggi concorrenti orientati agli oggetti, come Linda[48], che offrono un livello di astrazione molto più elevato, a scapito di ridotta portabilità delle prestazioni[68].

4.2 ASSIST

ASSIST (A Software development System based on Integrated Skeleton Technology) nasce all'interno del Dipartimento di Informatica dell'Università di Pisa, come proposta di un nuovo ambiente di programmazione orientato allo sviluppo di applicazioni parallele e distribuite con un approccio unificato. L'obiettivo principale è la creazione di un sistema di programmazione ad alto livello per applicazioni complesse e multidisciplinari, che garantisca elevate prestazioni e scalabilità su differenti piattaforme (da singole macchine parallele a cluster eterogenei fino a griglie computazionali) e il riuso di software parallelo preesistente.

Lo sviluppo di ASSIST deriva dall'esperienza maturata nei precedenti ambienti P^3L e *SkIE*, bastati su un modello a skeleton, che hanno mostrato i benefici degli skeleton ma anche le loro maggiori limitazioni:

- gli skeleton sono composti tramite interfacce ben definite, e separano l'implementazione dalla definizione del programma;
- le parti sequenziali degli skeleton possono essere scritte in qualsiasi linguaggio di programmazione, aumentando la produttività degli sviluppatori di applicazioni che non devono riscrivere il codice già prodotto;
- l'adozione di linguaggi standard permette anche di utilizzare i compilatori preesistenti, che garantiscono la creazione di un codice sequenziale ottimizzato;
- con alcune restrizioni, anche del codice binario può essere utilizzato per le parti sequenziali degli skeleton, garantendo quindi anche la portabilità di applicazioni chiuse e non modificabili;
- la portabilità delle performance risulta soddisfacente, soprattutto su piattaforme omogenee, in quanto la conoscenza dei modelli dei costi permette di ricompilare i programmi per utilizzare efficientemente l'ambiente di esecuzione.

In certi casi però l'espressività degli skeleton non basta per sviluppare applicazioni complesse, ed ASSIST si propone di superare alcuni di questi problemi, in particolare:

- oltre alla capacità di esprimere alcuni schemi paralleli tipici, si vuole offrire la possibilità di descrivere strutture più generiche, in modo da favorire applicazioni con pattern di comunicazioni particolari non supportate dagli skeleton classici;

- i modelli a skeleton offrono spesso una semantica funzionale e deterministica, che può limitarne l'uso in applicazioni complesse;
- si vuole migliorare il concetto di composizione di skeleton, e superare le inefficienze indotte in alcune forme che mischiano i concetti di parallelismo su dati e su stream;
- in molte applicazioni è necessaria la presenza di uno *stato condiviso*, o meglio una *Distributed Shared Memory* per lavorare efficientemente con grandi quantità di dati e semplificare la programmazione di problemi dinamici o irregolari;
- infine, progetti precedenti hanno dimostrato che il modello a skeleton non è spesso sufficiente per riusare applicazioni parallele scritte in differenti formalismi, in quanto si riesce ad ottenere un buon grado di riuso solo per le parti interamente sequenziali; si vogliono quindi sviluppare tecniche per aumentare il riuso del codice parallelo preesistente.

Tutte queste considerazioni hanno portato allo sviluppo di ASSIST, che cerca di superare tutte le limitazioni sopra descritte:

1. permette di esprimere programmi paralleli e distribuiti tramite grafi generici, che si sono rivelati abbastanza potenti per modellare la maggior parte delle applicazioni parallele; al tempo stesso, però, supporta il riconoscimento di strutture note, di cui sono conosciute informazioni aggiuntive (ad esempio il modello di costi);
2. i nodi del grafo sono moduli paralleli (parmod) o sequenziali, facilmente sostituibili con nuove versioni senza modificare il resto dell'applicazione;
3. i moduli comunicano tra loro attraverso *stream con tipo*, rappresentanti nel grafo come archi;
4. il codice sequenziale contenuto nei moduli può essere sorgente scritto nei linguaggi più comuni (C, C++, Fortran) oppure, con alcune limitazioni, un binario precompilato;
5. il concetto di skeleton non appare più in modo evidente nel modello, ma è stato sostituito da quello di "modulo parallelo", che può essere considerato una sorta di skeleton generico, programmabile per emulare sia le forme di parallelismo più comuni e normalmente rappresentate dagli skeleton, sia forme differenti (composizione di parallelismo su dati e su stream, non determinismo, etc);

6. i moduli paralleli introducono il concetto di "stato interno" per sopperire alla semantica funzionale degli skeleton, e la possibilità di gestire il non determinismo sugli stream in ingresso;

7. permette di condividere dati tra i moduli tramite "oggetti condivisi" implementati con DSM, per sopperire ai tipici problemi degli stream nel caso di moduli che lavorano con grandi moli di dati.

Ovviamente la maggiore generalità si paga; nel caso particolare di ASSIST la struttura a grafo generico e il modulo parallelo che racchiude differenti tipologie di skeleton potrebbero limitare l'applicabilità dei modelli di costo e di conseguenza le ottimizzazioni possibili. In realtà la computazione mantiene una struttura ben precisa che, seppur più complicata da analizzare rispetto alla semplice composizione di skeleton, si può formalizzare grazie ad elementi di teoria delle code; per quanto riguarda il parmod, con esso si riesce comunque a realizzare solo forme di parallelismo note (esattamente come con gli skeleton) di cui conosciamo tutte le informazioni.

L'evoluzione di ASSIST La bontà e la generalità del modello di ASSIST sono state poi dimostrate nel tempo, grazie a successivi progetti.

Innanzitutto la caratteristica principale derivata dal modello è che i moduli ASSIST possono essere compilati in una versione *parametrica* sul grado di parallelismo in modo trasparente al programmatore; in questo modo il grado di parallelismo viene scelto al momento dell'esecuzione, in base alle caratteristiche della piattaforma scelta o delle prestazioni desiderate.

Un successivo progetto[79] ha permesso l'introduzione di meccanismi *a run time* per la modifica del grado di parallelismo dell'applicazione, estendendo così le potenzialità dell'ambiente; a questo è stato poi affiancato un sistema di gestione automatica del grado di parallelismo che, in base ad un contratto di QoS specificato dal programmatore e variabile nel tempo, decide autonomamente se aggiungere o rimuovere processi all'interno dei moduli paralleli. Questo ha portato ad una prima forma di *Adattività* delle applicazioni ASSIST.

La tipizzazione degli stream ha permesso la realizzazione di meccanismi automatici di conversione dei tipi tra piattaforme differenti, per permettere l'esecuzione di applicazioni parallele su piattaforme eterogenee.

Tutte queste funzionalità sono poi confluite in una versione di ASSIST che supporta griglie computazionali tramite il middleware Globus[6], e sfrutta le tecniche per eterogeneità ed adattività di cui abbiamo parlato sopra per supportare le caratteristiche intrinseche del grid computing.

Infine, nell'ambito del progetto GRID.it[11] i moduli e le applicazioni ASSIST sono state incapsulate in componenti e web-service, al loro interno paralleli, per fornire componenti e servizi ad alte prestazioni.

Tutte le modifiche di cui abbiamo parlato sono state apportate tramite modifiche al supporto e all'implementazione. *Non abbiamo toccato il modello*, che si è rivelato molto flessibile e potente.

Al fine di dimostrare l'effettiva programmabilità di ASSIST sono state poi realizzate molte applicazioni complesse, ad esempio:

- algoritmi di data mining irregolari e difficilmente implementabili tramite skeleton, come il C4.5[78];
- algoritmi di simulazione, come il Barnes-Hut N-Body[7];
- un sistema di Knowledge Discovery parallelo su Database, integrato con altre applicazioni, per il progetto SAIB (System for Internet Banking Applications)[32];
- un algoritmo di "isosurface extraction" applicato alla bioinformatica[59].

ASSIST si è quindi dimostrato un modello di programmazione parallela molto espressivo, e al tempo stesso adatto per delegare la gestione di gran parte dei dettagli al supporto.

4.3 Il modello di programmazione di ASSIST

Questa sezione cerca di approfondire i concetti e la sintassi base del modello di ASSIST. Si tratta comunque di una introduzione, in cui trattiamo principalmente gli aspetti del modello a cui siamo interessati nell'ottica di illustrare successivamente ASSISTANT. Per maggiori dettagli si rimanda a [30, 78].

Le applicazioni ASSIST vengono descritte tramite l'uso di un "linguaggio di coordinamento"[48], chiamato ASSIST-CL, col quale si definisce la struttura parallela: i moduli e le loro interazioni. Ad ogni entità viene poi associato del codice sequenziale, in formato sorgente oppure, in casi particolari, binario precompilato.

4.3.1 Struttura di una applicazione

Gli stream Come anticipato, la struttura di ASSIST è un grafo di moduli collegati tramite stream. Uno *stream* è una sequenza ordinata, di lunghezza potenzialmente illimitata, di valori tipati. Un modulo viene attivato in base

alla presenza di un valore su uno o più degli stream in entrata, seguendo delle politiche definibili dal programmatore. In questo modo si può modellare semplicemente un comportamento su stream di tipo data-flow, ma anche grafi più generali, con cicli o con moduli che lavorano in modo non deterministico sugli stream in ingresso; inoltre non ci sono relazioni tra il numero di elementi degli stream in ingresso e quelli in uscita.

I tipi degli stream sono definibili dal programmatore, con un meccanismo analogo alle *struct* del linguaggio C. ASSIST definisce dei tipi primitivi che riprendono i tipi classici dei linguaggi di programmazione: booleani, interi, numeri in virgola mobile, etc. Per una lista completa dei tipi primitivi e del loro mapping nei tipi dei linguaggi ospite si rimanda a [30]. A questi ne sono affiancabili di nuovi definiti dal programmatore tramite la composizione di tipi primitivi. Gli stream possono, ovviamente, contenere anche array multidimensionali.

I moduli Abbiamo detto che i moduli ASSIST sono di due tipi

1. **Sequenziali**: moduli al loro interno sequenziali, attivati in presenza di un valore su ogni stream di ingresso; producono uno o più valori sugli stream di uscita per ogni attivazione. A causa dell'attivazione in presenza di elementi su tutti gli stream, non permettono di gestire il non determinismo; sono adatti per realizzare stadi di un pipeline o di un grafo data-flow.

2. **Paralleli**: moduli internamente paralleli; in questo caso sono definibili condizioni più complesse sugli stream di ingresso, tra cui non determinismo e selezione in base allo stato corrente. Permette di definire computazioni di tipo farm o data-parallel con stencil; vedremo più avanti, con la definizione, le sue potenzialità.

Un modulo può anche non avere stream di ingresso o di uscita; nel primo caso la modalità di attivazione non può essere definita sugli stream. ASSIST prevede che il modulo venga attivato una sola volta, all'avvio dell'applicazione.

Il grafo di moduli La composizione di moduli e stream avviene in ASSIST-CL tramite il costrutto *generic*, che modella un grafo generico. Al suo interno vengono dichiarati gli stream (archi del grafo) e i moduli (nodi). Nel listato 4.1 si può vedere la sintassi del costrutto, mentre nel 4.2 è modellato un grafo con quattro moduli, illustrato graficamente in figura 4.1. In questo esempio si può anche vedere la definizione di un nuovo tipo per rappresentare delle coordinate cartesiane.

```
generic main() {
  stream <tipoStream_1> <nomeStream_1>;
    ...
  stream <tipoStream_M> <nomeStream_M>;

  <nomeModulo_1> (input_stream<nomeStream_{1,1}>,...,<nomeStream_{1,K_1}>
      output_stream<nomeStream_{1,K_1+1}>,...,<nomeStream_{1,L_1}>);
    ...
  <nomeModulo_N>
      (input_stream<nomeStream_{N,1}>,...,<nomeStream_{N,K_N}>
      output_stream<nomeStream_{N,K_N+1}>,...,<nomeStream_{N,L_N}>);
}
```

Listato 4.1: Sintassi del costrutto *generic* per la dichiarazione del grafo dei moduli in ASSIST-CL

```
typedef struct {
  long x;
  long y;
} T_cart;

generic main() {
  stream T_cart[N] A1;
  stream long A2;
  stream T_cart[N] B;

  genera1   (output_stream A1);
  genera2   (output_stream A2);
  elabora   (input_stream A1, A2  output_stream B);
  stampa    (input_stream B);
}
```

Listato 4.2: Esempio di definizione del grafo dei moduli

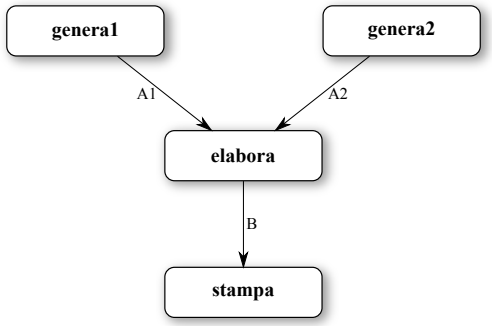


Figura 4.1: Grafo dell'esempio corrispondente al listato 4.2

4.3.2 Il codice utente ed il modulo sequenziale

Nella definizione di ASSIST si è cercato fornire un metodo unico e riusabile per descrivere il codice sequenziale, da utilizzare in tutte le parti dell'applicazione. Questo ha portato alla definizione delle **proc**. Le proc sono parti che possono contenere il codice utente, e sono utilizzate in ASSIST-CL per legare i moduli al loro contenuto. Una proc definisce una *funzione*, con un insieme di parametri di ingresso e di uscita. Il codice della funzione è esprimibile in uno dei linguaggi ospite supportati, e sono definiti dei costrutti per richiedere l'inclusione di file sorgenti (**inc**) e/o oggetto (**obj**) esterni. Nel listato 4.3 è presente la sintassi completa, mentre nel 4.4 mostriamo un esempio.

I tre linguaggi attualmente supportati sono C, C++ e F77 (Fortran 77); dalla sintassi si può vedere anche la keyword **path**, per indicare i percorsi in cui cercare i file sorgenti e binari. Inoltre emerge la distinzione, per i valori in uscita, tra **out** e **output_stream**.

Vista la modalità di esecuzione dei moduli, le funzioni verranno invocate in corrispondenza di una attivazione; per questo, per poter definire moduli che producono un numero di risultati *differente* dal numero di attivazioni è necessario fornire l'astrazione di stream anche all'interno delle funzioni sequenziali. Una variabile di tipo **out** riprende il comportamento tipico dei valori di ritorno: alla fine dell'esecuzione conterrà un valore che verrà poi utilizzato in base alla struttura del modulo. Al contrario il tipo **output_stream** rappresenta uno stream, in cui il codice sequenziale può, durante l'esecuzione, decidere di inviare risultati utilizzando una funzione (**assist_out**) offerta dall'ambiente ASSIST.

È importante sottolineare che il modello è definito in modo da non avere visione del contenuto delle proc; non a caso **assist_out** non rappresenta una keyword di ASSIST-CL, ma una funzione che l'ambiente fornisce per tutti i linguaggi supportati.

Il modulo sequenziale Introdotto il concetto di proc possiamo presentare la sintassi del modulo sequenziale.

In questo tipo di modulo dobbiamo solo descrivere gli stream in ingresso/uscita ed il codice sequenziale, sotto forma di proc, da eseguire al momento dell'attivazione. La sintassi è quindi semplice e molto scarna, l'unica spiegazione degna di nota riguarda l'uso degli stream, e la sequenza di invocazione in presenza di proc multiple. Per avere una semantica corretta è necessario utilizzare tutti gli stream di entrata, e tutti quelli di uscita. Gli stream di ingresso possono essere usati su più proc, mentre quelli di uscita esattamente in una sola. Nel caso di più invocazioni di proc, queste verranno eseguite sequenzialmente, una dopo l'altra, nell'ordine in cui si presentano. Nel listato

```
proc <nomeProc>(in <tipo_1> <nome_1>,...,<tipo_K> <nome_K>
          out <tipo_K+1> <nome_K+1>,...,<tipo_H> <nome_H>
          output_stream <tipo_H+1> <nome_H+1>,...,<tipo_M> <nome_M>)

inc< <sorgente_1>,...,<sorgente_L> >
path< <directory_1>,...,<directory_S> >
obj< <binario_1>,...,<binario_G> >

$<linguaggio>{
  codice sequenziale del linguaggio
}<linguaggio>$
```

Listato 4.3: Sintassi del costrutto *proc* per la definizione di una funzione sequenziale in ASSIST-CL

```
proc Fgenera1(output_stream T_cart A1[N])
inc<"iostream">
$C++{
  T_cart tmp_A[N];
  std::cerr << "STARTING fgen1" << std::endl;
  for (int k=0; k<MAX_ITER1; k++) {
    for (int i=0; i<N; i++) {
      tmp_A[i].x =  i+(k/MAX_ITER1);
      tmp_A[i].y = (i+(k/MAX_ITER1)) * (i+(k/MAX_ITER1));
    }
    assist_out (A1, tmp_A);
  }
}C++$
```

Listato 4.4: Esempio di definizione di una funzione sequenziale

```
<nomeModulo>(input_stream <tipo_1> <nome_1>,...,<tipo_L> <nome_L>
        output_stream <tipo_{L+1}> <nome_{L+1}>,...,<tipo_S> <nome_S>){

  <nomeProc_1>(in <nome_{1,1}>,...,<nome_{1,K_1}>
          out <nome_{1,K_1+1}>,...,<nome_{1,H_1}>
          output_stream <nome_{1,H_1+1}>,...,<nome_{1,M_1}>);
    ...
  <nomeProc_N>(in <nome_{N,1}>,...,<nome_{N,K_N}>
          out <nome_{N,K_N+1}>,...,<nome_{N,H_N}>
          output_stream <nome_{N,H_N+1}>,...,<nome_{N,M_N}>);
}
```

Listato 4.5: Sintassi per la definizione di un modulo sequenziale in ASSIST-CL

```
genera1(output_stream T_cart A1[N]) {
  Fgenera1(output_stream A1);
}
```

Listato 4.6: Esempio di definizione di un modulo sequenziale

4.5 si trova la sintassi della definizione del modulo, mentre nel 4.6 si trova un modulo di esempio che utilizza la proc definita in 4.4.

4.3.3 Il modulo parallelo

Il modulo parallelo (o parmod) rappresenta il fulcro dell'ambiente ASSIST in quanto, solo o tramite composizioni, permette di esprimere le forme di parallelismo più comuni.

I Virtual Processor

Il parmod è essenzialmente composto da un gruppo di *processori virtuali* (VP), entità che al momento dell'attivazione vengono eseguite in modo *indipendente* ed il più possibile parallelo. Il numero di VP eseguiti effettivamente in parallelo dipende, ovviamente, dal grado di parallelismo scelto al momento dell'esecuzione; i singoli VP eseguono codice sequenziale, perciò il grado di parallelismo massimo di un singolo parmod è determinato dal numero di VP dichiarati. I VP, pur essendo indipendenti, possono cooperare tra di loro per mezzo di una *astrazione* di stato condiviso, che vedremo più avanti.

I Virtual Processor di un parmod sono configurabili per realizzare sia forme stream-parallel che data-parallel, in particolare:

- stream-parallel di tipo farm;
- data-parallel di tipo map&reduce;
- data-parallel con stencil.

Inoltre possono essere utilizzati come *modulo sequenziale evoluto*, con la gestione del non determinismo sugli stream di ingresso ed uno stato interno. Con un parmod non possono essere realizzate le forme stream-parallel *pipeline* e *data-flow*, ma questo non comporta una limitazione di espressività del modello, in quanto le due forme sono realizzabili al livello del *grafo* suddividendo gli stadi in moduli differenti; al contrario, in questo modo è possibile realizzare composizioni di forme di parallelismo (tipiche degli skeleton) e realizzare stadi al loro interno *farm* o *data-parallel*.

La Topologia e lo stato interno Esistono differenti modi di "organizzare" i VP di un parmod, in base alla forma di parallelismo che vogliamo realizzare; l'organizzazione, chiamata *topology*, rappresenta la modalità con cui i VP sono indicati all'interno del modulo. Esistono tre tipologie in ASSIST:

- *array multidimensionale*: ogni VP rappresenta un elemento di un array, e viene indicato tramite indice; questa modalità si adatta a forme di tipo data-parallel, in cui ogni ad ogni VP viene assegnata una partizione precisa dei dati in ingresso e, nel caso di stencil, deve comunicare con gli altri;
- *none*: i VP non sono indicabili singolarmente, e vengono scelti in maniera automatica dal sistema; adatta per computazioni in cui i VP lavorano in modo indipendente, come ad esempio computazioni stream-parallel farm;
- *one*: per avere un singolo VP; il modulo ha una semantica sequenziale, ma può sfruttare alcune caratteristiche tipiche del parmod come lo stato interno e il non determinismo sugli stream.

Una caratteristica fondamentale del parmod è la possibilità di definire uno stato interno; questo può essere fondamentalmente di due tipi: *replicato* o *partizionato*. Nel primo caso ogni VP ha la sua copia privata dello stato, alla quale accede esclusivamente; nel secondo, invece, tutti i VP possono accedere *in lettura* all'intera struttura, che però è partizionata tra i processori virtuali; ognuno ha inoltre la possibilità di modificare la propria partizione (*Owner-Compute-Rule*).

```
topology <tipoTopologia> <Parametri> <nomeProcessoriVirtuali>;

attribute <tipo_1> <nome_1> <modalita_1> <parametri_1>;
...
attribute <tipo_N> <nome_N> <modalita_N> <parametri_N>;

stream <tipo_N+1> <nome_N+1>;
...
stream <tipo_M> <nome_M>;

init{
  codice per la inizializzazione degli attributi (C/C++)
}
```

Listato 4.7: Sintassi per la definizione di topologia e stato interno in ASSIST-CL

Nello stato interno sono anche definibili *stream temporanei*, usati per collezionare i dati in uscita dai processori virtuali ed effettuare una fase di post-elaborazione prima dell'uscita dal modulo. Vedremo meglio l'utilizzo nella fase di collezione dei risultati del parmod.

Lo stato di un parmod può essere definito in parte replicato ed in parte partizionato; è importante sottolineare però come con una topologia anonima (none) la partizione dello stato non è possibile, in quanto i VP non sono indicabili singolarmente; in questo caso si può solo definire uno stato replicato.

Ovviamente con l'introduzione dello stato è necessaria anche una modalità di inizializzazione, fornita in ASSIST-CL tramite la sezione **init** del parmod; questa avviene in modo *indipendente per ogni VP*, ed è esprimibile in linguaggio C/C++ (inserito direttamente e non tramite proc). Le variabili dello stato sono invece definite con la keyword **attribute**. Nel listato 4.7 trovate la sintassi per definire topologia e stato interno, mentre nel 4.8 è riportato un esempio per la topologia array con stato sia partizionato che replicato.

Il codice sequenziale dei VP La definizione del codice sequenziale contenuto nei VP risiede nella sezione **virtual_processors**. Il comportamento è definibile in modo separato per differenti condizioni di attivazione (che vedremo più avanti) e, nel caso di topologia array, anche per VP singoli o gruppi. Grazie a questa caratteristica si possono modellare semplicemente computazioni particolari, ad esempio data-parallel con stencil in cui i VP ai bordi hanno un comportamento differente.

```
topology array [i:N][j:N] VP;

attribute long S[N][N] scatter S[*i][*j] onto VP[i][j];
attribute long L replicated;
stream long out_matrix;
init{
  L = 0;
}
```

Listato 4.8: Esempio di definizione di topologia e stato interno

```
virtual_processors {
  elaborazione( in guard1 out out_matrix){
    VP i=1..N, j=1..N {
      for(h=0;h<N;h++){
        F(in L, S[i][h],S[h][i] out S[i][j]);
      }
      assist_out(out_matrix, S[i][j]);
    }
  }
}
```

Listato 4.9: Esempio di definizione del codice eseguito dai VP

Il codice utente deve essere racchiuso in apposite **proc**, seguendo la sintassi vista prima; come nel modulo sequenziale si possono usare più proc in cascata, ma nel parmod è anche possibile utilizzare costrutti aggiuntivi: eseguire operazioni tra interi e definire cicli *for* e *while* per modellare algoritmi con iterazioni. Per una sintassi precisa e completa di questa parte si rimanda a [30]; qui riportiamo solamente un esempio, nel listato 4.9

Nell'esempio (un data-parallel con stencil) si definisce la stessa computazione per ogni VP, che consiste nell'eseguire N volte una funzione che prende come parametri due elementi di una matrice presente nello stato interno, ed aggiorna l'elemento (i,j) della matrice stessa, su cui il VP ha possibilità di scrittura. Infine, al termine del loop, l'elemento modificato viene inviato sullo stream out_matrix per essere poi collezionato.

Distribuzione e raccolta dei dati

Parliamo ora di due parti importanti del parmod, ovvero le modalità di attivazione e di distribuzione degli elementi ricevuti sugli stream di ingresso e la collezione dei dati prodotti dai VP per formare gli stream di uscita. Queste parti sono gestite da due costrutti particolari del parmod: **input_section** per

gestire gli stream di input e l'attivazione, e **output_section**, per la collezione dei risultati e la preparazione degli stream di output.

Input Section Questa parte si occupa delle operazioni sugli stream in ingresso. Permette di gestire il non determinismo e condizioni particolari di attivazione, riprendendo il comportamento dei comandi con guardia del linguaggio concorrente ECSP[18].

Nella sezione di input vengono definite una serie di *guardie*. Ogni guardia è composta da una tripla <*priorità,condizione booleana,lista di stream*>, e viene definita *attiva* se e solo se la condizione booleana è verificata e su ogni stream della lista è presente un valore.

Se una guardia della sezione di input è attiva, il modulo stesso viene attivato, ed assume il comportamento associato a quella guardia; nel caso più guardie siano attive contemporaneamente, viene scelta quella a priorità più alta, e a parità di priorità in modo non deterministico.

Attivato il modulo, i virtual processor assumono il comportamento legato alla guardia; per fare questo devono però essere distribuiti i valori presenti sugli stream ai VP. Con ASSIST è possibile specificare politiche di distribuzione differenti per ogni stream, scelte tra:

- *on demand*: l'intero valore è inviato ad uno dei VP attualmente "liberi", scelto non deterministicamente;
- *scheduled*: l'intero valore è inviato ad uno solo dei VP, indicato in fase di distribuzione;
- *scatter*: il dato viene partizionato tra i vari VP;
- *multicast*: il dato viene inviato interamente a tutti i VP appartenenti ad un particolare gruppo;
- *broadcast*: il dato viene inviato a tutti i VP.

Le distribuzioni *scheduled* e *scatter* richiedono una topologia di tipo **array** per poter essere utilizzate, mentre le altre sono disponibili su tutte le topologie. I dati possono essere inviati ai VP direttamente come stream, oppure attraverso l'uso delle variabili di stato.

Oltre alla distribuzione dei valori, all'attivazione di un modulo è possibile effettuare altre operazioni, ad esempio modificare il valore di quelle variabili che determinano la condizione booleana delle guardie o la terminazione del modulo.

Output Section In molti casi i dati prodotti dai VP non possono essere inviati direttamente sugli stream di output, e in generale necessitano di una fase di "collezione", in cui si ottengono i valori dai singoli VP per generare un unico dato da inviare sullo stream. Questo è particolarmente vero nel caso dei data-parallel, in cui ogni VP produce una parte del risultato.

La sezione di output permette di collezionare i dati dai VP, *eventualmente* effettuare anche una fase di postprocessing, e aggiornare le variabili di stato utilizzate dalla sezione di input per l'attivazione delle guardie e la terminazione del modulo.

ASSIST definisce due tipologie di collezione dei dati:

- **from any**, per elaborare separatamente gli output dei VP;
- **from all**, per raccogliere gli output di tutti i VP e successivamente effettuare una elaborazione.

Il primo caso permette di trattare facilmente ed *efficientemente* i casi in cui i singoli VP producano un output per il modulo; il secondo si adatta invece al caso di cui abbiamo parlato sopra. Nel caso di collezione *from all* si ottiene una struttura iterabile tramite il costrutto **AST_FOR_EACH**, che restituisce i singoli risultati già ordinati per VP, indipendentemente dall'ordine col quale sono stati prodotti.

Come per la sezione **init** anche in questo caso si può usare del codice C/C++ per determinare il comportamento di questa parte di parmod, e in questo modo definire anche operazioni di post-processing.

Nel listato 4.10 è riportata la sintassi dei due costrutti. Nel 4.11 si trova invece un esempio, da cui si può vedere l'utilizzo del costrutto per iterare su tutti gli elementi ricevuti dai VP.

La struttura completa del parmod

Vediamo ora di definire in modo più completo la struttura di un parmod. Come abbiamo detto è composto essenzialmente dai Processori Virtuali, ma con la necessità di una fase di distribuzione e una di collezione. Grazie agli studi sulle forme di parallelismo sappiamo che modellare queste due fasi come entità separate dalla computazione permette di ottenere prestazioni *migliori* e allo stesso tempo *predicibili*. In questo modo infatti stiamo definendo un *pipeline*, dove i tre stadi sono:

- **Input Section**, che si occupa di ricevere i dati dagli stream e distribuirli ai Virtual Processor;
- **Virtual Processor**, che effettuano il calcolo vero e proprio;

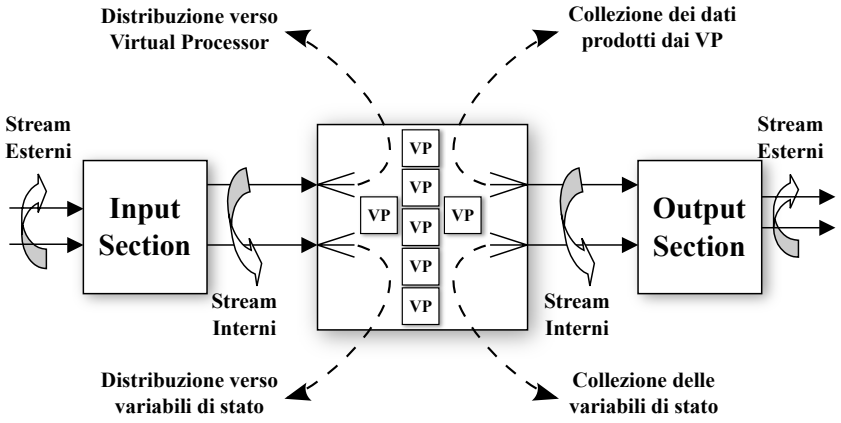
```
input_section{
  <nomeGuardia_1>: on <priorità_1>,<condizione_1>,
     <stream_1,1>&&...&&<stream_1,K_1> {
    distribution <stream_1,1> <distribuzione_1,1> <parametri_1,1>;
      ...
    distribution <stream_1,K_1> <distribuzione_1,K_1> <parametri_1,K_1>;
    operation { Codice per le variabili di condizione (C/C++)
    }<variabili di stato modificate>
  }
    ...
  <nomeGuardia_N>: on <priorità_N>,<condizione_N>,
     <stream_N,1>&&...&&<stream_N,K_N> {
    distribution <stream_N,1> <distribuzione_N,1> <parametri_N,1>;
      ...
    distribution <stream_N,K_N> <distribuzione_N,K_N><parametri_N,K_N>;
    operation { Codice per le variabili di condizione (C/C++)
    }<variabili di stato modificate>
  }
}while(<condizione di fermata>)

output_section{
  collects <nomeStream_1> from <tipoCollezione_1> <parametri_1>{
    Codice per post-processing e modifica dell variabili (C/C++)
  }<variabili di stato modificate>;
  ...
  collects <nomeStream_M> from <tipoCollezione_M> <parametri_M>{
    Codice per post-processing e modifica dell variabili (C/C++)
  }<variabili di stato modificate>;
}
```

Listato 4.10: Sintassi per la definizione di input e output section in ASSIST-CL

```
input_section{
  guard1: on , , A && X {
    distribution A[*i0][*j0] scatter to S[i0][j0];
    distribution X broadcast to L;
  }
}while(true)
output_section{
  collects out_matrix from ALL Pv[i][j] {
    int elem; int ris[N][N];
    AST_FOR_EACH(elem) {
      ris[i][j]=elem;
    }
    assist_out(result, ris);
  }<>;
}
```

Listato 4.11: Esempio di definizione di input e output section

Figura 4.2: Rappresentazione grafica di un modulo parallelo



- **Output Section**, che colleziona i risultati ed effettua una fase di post-processing.

Questa suddivisione è importante per mascherare la latenza delle comunicazioni e del post-processing; inoltre permette ulteriori ottimizzazioni su pipeline di parmod, in cui sezioni di input e di output possono essere fuse per diminuire ulteriormente la latenza. Questa struttura a *pipeline* è illustrata in figura 4.2.

A questo punto possiamo definire la sintassi completa del parmod, che trovate nel listato 4.12. Nel 4.13 riportiamo invece il parmod completo derivato dagli esempi precedenti.

4.3.4 Oggetti esterni e DSM

I moduli possono utilizzare oggetti esterni, per cooperare con altri moduli ASSIST (dell'applicazione stessa o di altre) o applicazioni esterne. In ASSIST esistono due tipi di oggetti esterni.

- *Variabili condivise*, variabili definite con i tipi di dati di ASSIST-CL: estensione del concetto di stato interno del modulo, le variabili condivise rappresentano lo stato dell'intera applicazione. Sono dichiarate utilizzando la keyword *shared*, all'esterno delle definizioni dei moduli e del grafo. Eventuali strategie per la cooperazione dei moduli tramite variabili condivise spettano al programmatore, che può ad esempio usare gli stream per mandare messaggi di sincronizzazione.

```
parmod <nomeModulo>(
    input_stream <tipo_1> <nome_1>,...,<tipo_K> <nome_K>
    output_stream <tipo_K+1> <nome_K+1>,...,<tipo_M> <nome_M>){
  definizione topologia
  definizione attributi (stato interno)
  init{
    inizializzazione stato interno
  }
  input_section{
    definizione guardie e distribuzione dei dati
  }while(condizione di terminazione)
  virtual_processors {
    definizione del comportamento dei VP
  }
  output_section{
    raccolta dei dati ed invio sugli stream di uscita
  }<variabili di stato modificate>;
}
```

Listato 4.12: Sintassi per la definizione di un modulo parallelo in ASSIST-CL

- *Oggetti contenuti in Distributed Shared Memory*, accessibili all'interno dei programmi ASSIST attraverso librerie proprie delle DSM specifiche; rispetto alle variabili condivise, possono ospitare tipi diversi da quelli definiti da ASSIST ed essere condivisi anche attraverso differenti applicazioni, ma è cura del programmatore utilizzare le librerie in modo corretto; teoricamente tutte le DSM possono essere utilizzate nei moduli, importando le adeguate librerie; attualmente ASSIST integra il supporto per DVSA, Shared Tree, SHOB e Reference, anche se normalmente si consiglia di utilizzare la Reference[30].

Dal punto di vista modellistico le due tipologie si differenziano in modo sostanziale: nel primo caso parliamo di entità *del modello*, che fanno parte della struttura del programma ASSIST e che sono gestite in modo automatico dal suo supporto. Nel secondo invece abbiamo librerie esterne, utilizzate dentro al codice utente e non rappresentate nel modello. Queste non sono gestibili dal supporto, e tutti gli aspetti sono delegati al programmatore. Per fare un esempio, le tecniche di conversione di tipi per ambienti eterogenei non possono essere garantite con le DSM esterne.

```
parmod matrixCalc(input_stream long S[N][N], long L
    output_stream long result[N][N]){
  topology array [i:N][j:N] VP;

  attribute long S[N][N] scatter S[*i][*j] onto VP[i][j];
  attribute long L replicated;
  stream long out_matrix;
  init{
    L = 0;
  }

  do input_section{
    guard1: on , , A && X {
      distribution A[*i0][*j0] scatter to S[i0][j0];
      distribution X broadcast to L;
    }
  }while(true)

  virtual_processors {
    elaborazione( in guard1 out out_matrix){
      VP i=1..N,j=1..N {
        for(h=0;h<N;h++){
          F(in L, S[i][h],S[h][i] out S[i][j]);
        }
        assist_out(out_matrix, S[i][j]);
      }
    }
  }

  output_section{
    collects out_matrix from ALL Pv[i][j] {
      int elem; int ris[N][N];
      AST_FOR_EACH(elem) {
        ris[i][j]=elem;
      }
      assist_out(result, ris);
    }<>;
  }
}
```

Listato 4.13: Esempio di definizione di un modulo parallelo

4.4 L'implementazione di ASSIST

In questa sezione descriveremo brevemente l'attuale implementazione di ASSIST. Non ci soffermeremo troppo su questa parte, in quanto non siamo interessati all'implementazione ma al modello; per maggiori informazioni rimandiamo agli articoli [3, 7] e alla tesi [25].

L'ambiente ASSIST è composto dal compilatore, una serie di librerie e programmi per creare i moduli ed il GEA (Grid Execution Agent).

Una volta prodotto il file sorgente ASSIST si può utilizzare il compilatore *astCC*, fornito dall'ambiente, che si occupa di interpretare il linguaggio di coordinamento, compilare moduli e proc (invocando i compilatori necessari), e creare un file di configurazione (*ast.out.xml*) che descrive l'applicazione ed i suoi requisiti. Una applicazione è composta da una serie di programmi eseguibili da distribuire, insieme ad eventuali librerie e programmi di supporto, sul gruppo di macchine sulle quali eseguirla; questo rende la procedura piuttosto complicata, specialmente per un utente che non conosce a fondo l'ambiente ASSIST.

Nel tempo si sono susseguiti diversi sistemi di deployment e caricamento (chiamati GEA), atti a rispecchiare le modifiche apportate al compilatore. Attualmente la versione di GEA considerata stabile è il "Loader" [57], ma sono presenti studi per una nuova implementazione più flessibile[36]. Il Loader è composto da un server, che gestisce la distribuzione e l'esecuzione dei programmi ASSIST, e un client, per richiedere operazioni al server; i comandi principali consistono nel *caricamento* di un modulo o di una intera applicazione e l'*esecuzione* di una applicazione. Al momento dell'esecuzione il Loader si occupa di distribuire i file eseguibili sui nodi a disposizione (specificati in un file di configurazione xml e accessibili tramite ssh, oppure rintracciabili tramite i servizi *GIS* - Grid Information Services del kit di sviluppo Globus[45]) e di eseguire i vari moduli dell'applicazione, con il grado di parallelismo specificato nel file di descrizione dell'applicazione. Per una descrizione più approfondita del GEA si consiglia [57].

4.4.1 Compilatore ed esecutore di ASSIST

Il compilatore, astCC, è stato progettato per essere modulare ed estensibile; per questo motivo è diviso in tre livelli:

- **Front End**, che si occupa di interpretare il linguaggio ASSIST-CL, estrarre il codice sequenziale dei moduli e le dipendenze da librerie esterne, ed infine generare una rappresentazione interna al compilatore dell'applicazione;

- **Middle End**, nel quale i moduli vengono traslati in "*task code*", una implementazione in un linguaggio di programmazione (C++) dei moduli; su questo livello intermedio possono essere eseguite ottimizzazioni del codice, basate sulla conoscenza della struttura parallela dell'applicazione;
- **Back End**, che si occupa di compilare il *task code* prodotto, comprese le parti di codice sequenziale espresse in uno dei linguaggi di programmazione supportati, e riunire i file oggetto prodotti in singoli eseguibili. Una volta compilata l'applicazione, questo livello si occupa di creare dei file che la descrivono (il grafo dei moduli e tutte le informazioni necessarie al GEA).

Gran parte dell'implementazione dei moduli, però, non risiede effettivamente nel task code, ma in una libreria, *assistLib*, che contiene, ad esempio, le routines di comunicazione e sincronizzazione tra moduli e processi; modifiche a questa libreria permettono di usare differenti tecniche di comunicazione; attualmente si può scegliere di usare un'implementazione basata su ACE[1] (per l'esecuzione su cluster) o sul Globus Toolkit[2] (per l'esecuzione su griglia). Questa libreria, nell'ottica di inserire un overhead minimo, fa molto uso di *template* e codice *inline*.

In figura 4.3 sono illustrate le fasi di compilazione di un programma ASSIST.

Al contrario dei programmi ASSIST e del compilatore, il GEA è realizzato in Java. Questo per una maggiore modularità e facilità di programmazione di un tool che non influisce sulle prestazioni dell'applicazione; anche il GEA nasce con la portabilità in mente; utilizza infatti un sistema a plugin per permettere l'uso di differenti sistemi di caricamento. Attualmente sono presenti il plugin per Globus e quello per accedere alle macchine via SSH.

4.4.2 Compilazione di un programma ASSIST

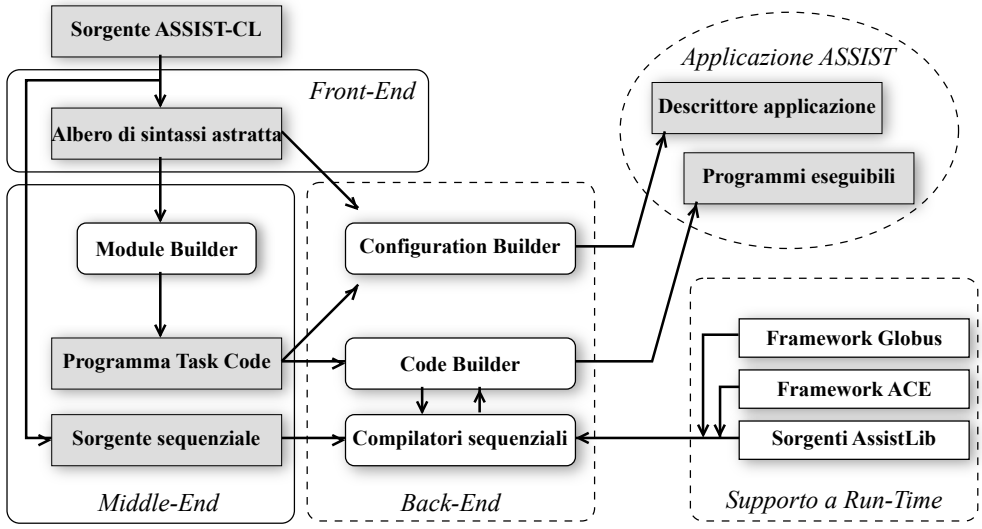
Vediamo ora brevemente come un programma ASSIST venga compilato in un insieme di processi cooperanti a scambio di messaggi. A seconda della libreria di comunicazione avremo poi una compilazione per cluster o per grid.

L'applicazione prodotta è *parametrica* sul grado di parallelismo: all'avvio si può decidere quanti processi allocare sulle macchine disponibili. A differenza dei modelli come MPI il prodotto non è un singolo processo SPMD, ma

[1] ADAPTIVE Communication Environment, un framework che implementa i principali pattern utilizzati in software con comunicazioni concorrenti[67].

[2] Un insieme di tool per realizzare applicazioni su grid[45].

Figura 4.3: Passi di compilazione di un programma ASSIST



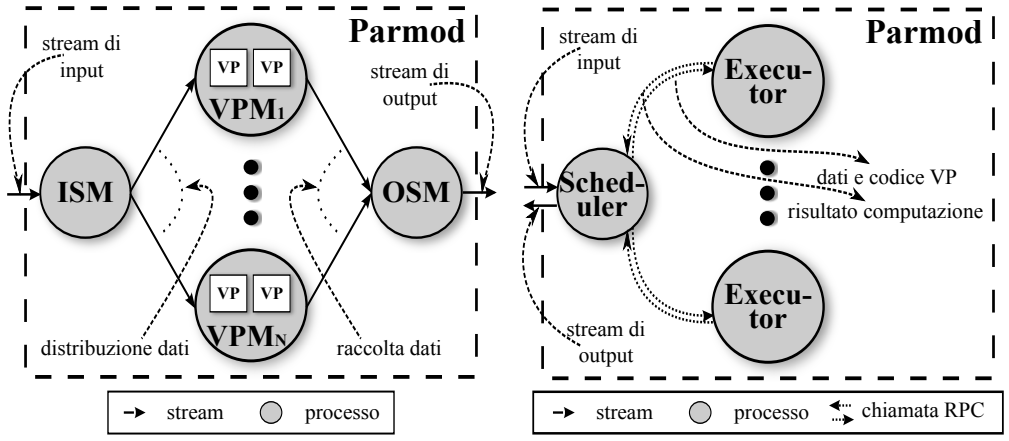
un gruppo di processi che implementano i moduli. I moduli sequenziali, visto l'assenza di parallelismo interno, vengono realizzati da un processo, mentre per i parmod si vuole mantenere la semantica *pipeline* introdotta prima, che porta alla definizione di tre processi: *Input Section Manager*, *Virtual Process Manager* e *Output Section Manager*.

Tra questi siamo ovviamente interessati ad allocare più o meno risorse al VPM, per poter eseguire in parallelo più processori virtuali. La definizione di VP ci permette di realizzare un Virtual Process Manager parametrico, di tipo SPMD, che all'avvio si coordina con tutte le sue istanze per decidere *come allocare* i *Processori Virtuali* sulle risorse fisiche.

Anche qui il comportamento varia in base alla topologia: in caso di processori anonimi, e quindi non numerati, si decide all'avvio il numero di processori *virtuali* presenti, facendolo coincidere con il numero di processi; se invece i VP sono indicizzati il loro numero è fissato dal programmatore e non modificabile; i processori virtuali vengono perciò partizionati sui processi esistenti.

In figura 4.4a si mostra la struttura dei processi cooperanti che *implementano* il parmod. La struttura è molto simile a quella del modello, perché questo tipo di implementazione offre generalmente le prestazioni migliori.

Detto questo esistono molte possibili implementazioni, ad esempio di tipo master-slave con ISM e OSM coincidenti in un unico processo che comunica con i VPM in stile RPC, oppure una sua evoluzione basata sul modello *macro*



(a) Parmod implementato tramite partizionamento dei Virtual Processors

(b) Parmod implementato con un modello macro data flow

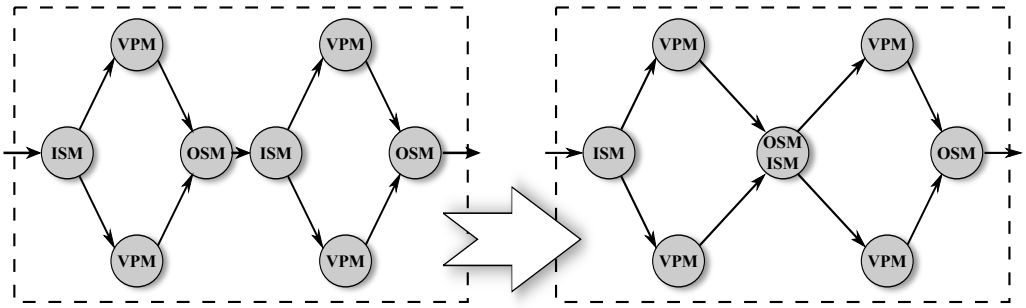
Figura 4.4: Differenti implementazioni di un parmod

data-flow, con uno scheduler e degli esecutori generici[34].

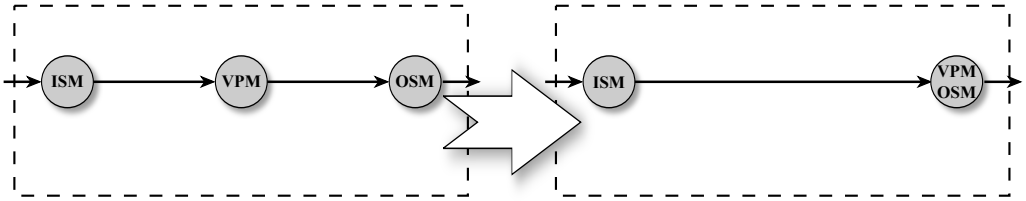
Nel secondo caso la struttura fisica è sempre di tipo master-slave, dove i worker non sono più i VPM visti prima ma degli esecutori generici di Virtual Processor, coordinati dallo scheduler. In questo modo si riesce a bilanciare quei programmi in cui il tempo di calcolo dei singoli VP è molto variabile, ed un partizionamento statico come quello definito dai VPM degrada le prestazioni. La Struttura di questa implementazione è riportata in figura 4.4b.

Sul modello di processi definito si possono poi applicare delle ottimizzazioni per ridurre la latenza, accorpando due o più processi. Esistono dei casi, infatti, in cui questa trasformazione porta *sempre* a latenze migliori e tempi di servizio uguali. Studiando il modello dei costi di un pipeline si può ad esempio vedere che ISM e OSM sono computazionalmente "leggeri"[3], con un tempo di servizio molto basso. Su un pipeline di parmod si possono perciò unire, come illustrato in figura 4.5a; analogamente nel caso di parmod con topologia *one* il singolo VPM può essere unito all'OSM, che non fa altro che ricevere un elemento da uno stream interno ed inviarlo su uno esterno (figura 4.5b).

[3]a parte casi particolari con codice di post-processing molto costoso



(a) Ottimizzazione su pipeline di parmod



(b) Ottimizzazione su parmod con topologia one

Figura 4.5: Esempi di ottimizzazioni in ASSIST

4.4.3 Adattività in ASSIST

In un successivo momento l'implementazione è stata estesa per permettere la modifica del grado di parallelismo dei singoli parmod durante l'esecuzione. In questo modo il programmatore può adattare il programma in un momento successivo al caricamento, in base alle attuali risorse a disposizione.

Per implementare questa caratteristica sono stati definiti dei punti in cui l'intero parmod raggiungeva uno stato consistente, chiamati *reconf_safe point*. In quei momenti può avvenire una riconfigurazione del parmod, ovvero una variazione del numero di processi VPM utilizzati. Anche in questo caso la procedura di riconfigurazione dipende strettamente dalla topologia del parmod:

- nel caso di topologia *none*, essendo i processori indipendenti, basta aggiungere o rimuovere un nuovo VP al sistema, notificare input e output section della sua presenza e, nel caso della rimozione, aspettare la terminazione dell'ultima attivazione prima di rimuovere il processo;

- nel caso di topologia *array* la cosa si fa più complicata, perché i VP sono già stati assegnati ai VPM presenti, ed è quindi necessario un nuovo partizionamento di essi.

Una seconda fonte di problemi è la presenza di uno *stato interno* ai VP, che deve essere trasportato nel caso del ripartizionamento ed inizializzato nel caso di aggiunta di VP.

Grazie alla presenza di questa dinamicità è stato possibile implementare una forma di adattività in ASSIST, dove il parmod si riconfigura automaticamente per mantenere un tempo di servizio richiesto dall'utente mediante un contratto di QoS. In questo caso nel parmod è stata inserita una nuova entità, il *manager*, che si occupa di analizzare il comportamento del modulo e richiedere eventuali riconfigurazioni. Inoltre i vari processi sono stati modificati per catturare statistiche in tempo reale sul comportamento del modulo: tempo di servizio, latenza, etc.

4.5 Conclusioni

In questo capitolo abbiamo presentato i risultati delle ultime ricerche nell'ambito della programmazione parallela strutturata, e l'ambiente di sviluppo progettato all'interno del Dipartimento di Informatica dell'Università di Pisa.

La possibilità di descrivere il programma in un modo tale da permettere lo sviluppo di strumenti di supporto *efficienti* per automatizzare molti degli aspetti fondamentali di una applicazione parallela è, dal nostro punto di vista, molto importante per lo sviluppo di applicazioni parallele ad alte prestazioni, soprattutto in ambienti particolarmente dinamici come le griglie computazionali.

Con ASSIST è stato introdotto un modello per la programmazione strutturata abbastanza formale da permettere lo sviluppo di un supporto efficiente con tali caratteristiche e, al tempo stesso, molto più espressivo dei modelli a skeleton puri.

ASSIST rappresenta quindi la soluzione ideale per sviluppare applicazioni parallele negli ambienti di *grid computing*, ma anche in altri ancora più dinamici come quelli pervasivi. Per questo sarà la base della definizione del modello di programmazione di ASSISTANT.

Parte II

ASSISTANT: un modello di programmazione HPPC

Capitolo 5

Un nuovo modello per applicazioni pervasive ad alte prestazioni

Nell'ottica di realizzare applicazioni HPPC, che uniscano le caratteristiche del pervasive computing alla richiesta di alte prestazioni, abbiamo inizialmente studiato gli strumenti di sviluppo attuali.

Per fare questo ci siamo concentrati sui due settori separatamente: nel capitolo 3 sono stati discussi i principali modelli esistenti per applicazioni pervasive, mentre nel 4 abbiamo riportato quello che per noi rappresenta lo stato dell'arte nei modelli per applicazioni parallele ad alte prestazioni.

Questo studio ci ha portato alla conclusione che nessuno dei modelli presenti si adatta perfettamente alle nostre esigenze; abbiamo quindi deciso di svilupparne uno nuovo, che tenesse conto nativamente sia degli aspetti di pervasive computing sia di quelli delle applicazioni parallele. Da questo è nato l'ambiente ASSISTANT: ASSIST with Adaptivity and coNText awareness.

ASSISTANT sfrutta l'esperienza maturata dal gruppo di ricerca nello sviluppo di modelli per applicazioni parallele, e riprende il modello di base di ASSIST, estendendolo per fornire i meccanismi di adattività e context awareness necessari per gestire le caratteristiche degli ambienti pervasivi.

In questo capitolo descriveremo le caratteristiche principali del modello, introducendo per la prima volta ASSISTANT. Innanzitutto riprendiamo brevemente le caratteristiche interessanti e le mancanze dei modelli visti nei capitoli precedenti (Sezione 5.1), per poi motivare la nascita di un nuovo modello e le caratteristiche cercate in esso (5.2). Parliamo poi dei concetti base di ASSISTANT: la suddivisione del modello a strati (Sezione 5.3) e la gestione delle problematiche di adattività (5.4), context-awareness (5.5) e tolleranza ai guasti (5.6).

5.1 I modelli studiati

Riprendiamo velocemente le caratteristiche dei modelli studiati, nell'ottica di capire quanto si prestano a realizzare una applicazione HPPC e se le idee di questi possano essere in qualche modo utilizzate come base per ASSISTANT.

5.1.1 Modelli per pervasive computing

Gli strumenti per pervasive computing attuali non sono abbastanza generali, e si adattano a poche tipologie di ambienti pervasivi. Ognuno di essi presenta comunque delle caratteristiche interessanti, che non ci fanno preferire *completamente* un modello agli altri, ma piuttosto protendere verso la definizione di un ambiente che in qualche modo riprenda tutte le caratteristiche interessanti viste in questi sistemi.

Quasi tutti i modelli hanno studiato ambienti *statici*, come stanze o edifici. Pur considerando la volatilità di alcuni dispositivi intrinsecamente mobili, assumono la presenza fissa di altre risorse. Questo ha semplificato notevolmente la definizione di questi sistemi, poiché ha permesso l'utilizzo di meccanismi di supporto centralizzati, allocabili sui nodi statici, per coordinare tutte le altre risorse del sistema, comprese quelle mobili.

Nel nostro caso vogliamo invece poter definire un supporto *completamente* decentralizzato, che possa essere distribuito interamente tra nodi mobili e tollerare disconnessioni improvvise, come per i sistemi Peer to Peer.

Allo stesso modo vorremmo offrire meccanismi di tolleranza a guasti e disconnessioni al livello applicativo. I modelli attuali generalmente non considerano questo problema, mentre noi vorremmo permettere alle applicazioni di poter continuare a lavorare anche in questi casi.

Se escludiamo MB++, inoltre, questi modelli non considerano minimamente applicazioni con requisiti computazionali elevati, caratteristica *fondamentale* per l'HPPC. Anche MB++ affronta questa necessità in modo troppo blando, in quanto si focalizza sul supportare un ambiente di esecuzione remota con caratteristiche prestazionali elevate, ma non necessariamente per applicazioni parallele. Permette comunque di descrivere computazioni di tipo data-flow, sulle quali offre un minimo di parallelismo allocando, se possibile, gli stadi su nodi differenti. Nonostante questo non garantisce di sfruttare al meglio le macchine e non offre garanzie sull'effettivo grado di parallelismo utilizzato. Dobbiamo inoltre considerare che i grafi data-flow modellano solo un piccolo sottoinsieme degli algoritmi paralleli. Per finire, questo modello non si pone nell'ottica di utilizzare le risorse pervasive presenti per suppor-

tare applicazioni parallele, e senza la connettività verso una piattaforma ad alte prestazioni remota non è di alcuna utilità.

Vorremmo perciò provare ad utilizzare un approccio completamente differente per il modello, che non nasca dai requisiti di pervasive computing, ma da quelli delle applicazioni ad alte prestazioni.

5.1.2 Modelli per applicazioni ad alte prestazioni

Per le applicazioni ad alte prestazioni abbiamo visto in dettaglio ASSIST (sviluppato qui a Pisa dal nostro gruppo di ricerca), che presenta un modello molto potente in grado di descrivere la maggior parte degli algoritmi paralleli, ma al tempo stesso ad alto livello.

Con ASSIST si possono gestire nativamente e in modo trasparente all'utente molte caratteristiche normalmente lasciate al programmatore, come la realizzazione di applicazioni eterogenee, la gestione di nodi fortemente dinamici tramite tolleranza a guasti e disconnessioni, e forme di adattività dell'applicazione parallela.

Tutte queste caratteristiche sono state studiate ed introdotte nell'ambito del grid computing, in cui diventa necessario trattare dinamicità ed eterogeneità delle risorse.

Una certa somiglianza tra grid e ambienti pervasivi è stata notata da tempo sia dagli studiosi di pervasive computing([38, 70]), sia dagli esperti di griglie ([62, 63]). Il concetto di *pervasive grid* è ormai utilizzato per denotare griglie composte da quei dispositivi mobili tipici del pervasive computing.

Per questi motivi ASSIST ci sembra un buon punto di inizio per la definizione di un modello per applicazioni ad alte prestazioni pervasive. Ovviamente la versione di ASSIST attuale, sviluppata solo per applicazioni parallele, non contiene molti dei concetti base richiesti come la gestione del contesto, il supporto a dispositivi mobili e meccanismi di adattività. Abbiamo visto come alcuni di questi requisiti (come la tolleranza ai guasti e alle disconnessioni improvvise) possano essere delegati al livello di supporto *senza* modificare il modello. Altre caratteristiche, invece, (ci riferiamo a context-awareness e adattività) non possono essere trattate in modo automatico e devono "apparire" all'interno del modello.

5.2 La nascita di ASSISTANT

Le considerazioni sui modelli visti ci hanno portato a decidere di adattare ASSIST ad ambienti pervasivi, inserendo meccanismi per gestire le problematiche di adattività e context awareness. Abbiamo perciò deciso di chia-

mare il nuovo modello ASSISTANT: *ASSIST with Adaptivity and coNText awareness*. Il nostro gruppo di ricerca sta lavorando su ASSISTANT da più di un anno, nell'ottica del progetto FIRB In.Sy.Eme (Integrated Systems for Emergencies); l'obiettivo attuale è quello di poter descrivere interamente una applicazione di gestione delle emergenze.

Rispetto ai progetti precedenti, questa volta si sono rese necessarie delle modifiche al *modello*. A livello implementativo la versione attuale di ASSIST è nata per ambienti di esecuzione statici, e nonostante le modifiche per il supporto delle griglie, non è facilmente adattabile ad ambienti fortemente dinamici; inoltre non è portabile (se non con grandi sforzi) su molti dispositivi tipici del pervasive computing come cellulari, pda e navigatori.

La necessità di modifiche importanti al modello e di una completa riprogettazione dell'implementazione ci portano a considerare ASSISTANT come un nuovo strumento, che riprende le caratteristiche di base di ASSIST, ma con un target di applicazioni molto differente; rappresenta quindi qualcosa di più di una semplice evoluzione.

5.2.1 Le caratteristiche richieste

Il nostro obiettivo principale è di mantenere le caratteristiche fondamentali di ASSIST per le applicazioni parallele, e fornire un ambiente utilizzabile *anche* per sviluppare applicazioni parallele su ambienti di esecuzione classici: sistemi a parallelismo massiccio, cluster di workstation, etc. Nelle applicazioni HPPC che immaginiamo, infatti, una parte di esse richiede calcoli molto pesanti *normalmente* eseguiti su piattaforme tipiche dell'HPC, e vorremmo ottenere le massime prestazioni possibili da questo tipo di macchine.

Ma in mancanza di tali risorse, l'applicazione può contare solo sull'insieme di dispositivi mobili attualmente presenti. Ci troviamo perciò di fronte ad un ambiente di esecuzione molto variabile che passa da cluster a palmari, con ovvie differenze prestazionali e architetturali.

Adattività La realizzazione di un ambiente per esprimere computazioni adatte a tutti questi sistemi è una sfida notevole, che richiede anche l'interazione col programmatore: un programma sviluppato per essere eseguito su cluster difficilmente potrà essere "trasportato" su un gruppo di PDA interconnessi con reti wireless. L'utilizzo di un modello ad alto livello, infatti, ci permette di compilare senza problemi il programma per le due piattaforme. Possiamo ottenere un modello dei costi, determinare l'implementazione più efficiente per la particolare piattaforma, etc. Ma le risorse hardware disponibili sui PDA non saranno sicuramente sufficienti a garantire una corretta esecuzione: anche ammettendo che la memoria di questi dispositivi possa

contenere i dati della computazione (un server può gestire qualche gigabyte di dati, il pda poche decine di megabyte) i tempi esecuzione sui dispositivi mobili saranno spesso talmente alti da risultare inutili.

L'applicazione parallela deve quindi *adattarsi* all'ambiente di esecuzione, esattamente come tutte le applicazioni pervasive.

Per garantire delle forme evolute di adattività l'interazione con l'applicazione è necessaria. I meccanismi di adattività di ASSIST, basati sulla modifica del grado di parallelismo, permettono di utilizzare un numero di risorse adeguato all'ambiente, ma spesso non sono sufficienti ad ottenere applicazioni in grado di essere eseguite su di esse.

Abbiamo visto nel capitolo 3, grazie ad Odyssey, che l'adattività intesa come "modifica del comportamento dell'applicazione" necessita una interazione con quest'ultima, e non può essere realizzata *solo* dal livello del supporto. La nostra idea è di permettere al programmatore di definire differenti "versioni" dell'applicazione, con requisiti computazionali differenti, adatte ai vari ambienti possibili.

L'approccio da noi proposto è quindi analogo a quello di Odyssey: un cambio del comportamento provocherà una qualità nei risultati differente, riprendendo esattamente il concetto di "fidelity" di Odyssey. A differenza del loro lavoro, però, non è necessariamente il visualizzatore a richiedere una fidelity particolare, ma il produttore del dato che, automaticamente, definisce la fidelity in base alle risorse a sua disposizione. Ci saranno comunque meccanismi per permettere la richiesta di un cambio di fidelity dalle altre entità dell'applicazione.

Context Awareness Altro punto fondamentale che richiede una modifica del modello è la gestione del contesto. Una applicazione deve poter interagire con esso, per acquisire informazioni anche complesse tramite meccanismi di inferenza e richiedere modifiche tramite degli attuatori.

Abbiamo quindi la necessità di modellare il contesto in modo che una applicazione parallela possa interagire con esso. In questo caso consideriamo interessanti le idee di CORTEX, dove il contesto emerge tramite un meccanismo di "eventi" che vengono notificati alle entità attive, che nel loro caso sono i sentient object, mentre nel nostro saranno i moduli dell'applicazione. Questi eventi possono essere informazioni "primitive", acquisite dai sensori (fisici o virtuali), oppure derivate (ad esempio tramite meccanismi di inferenza).

Non abbiamo ancora definito completamente la parte di gestione del contesto, ma pensiamo ad una soluzione completamente decentralizzata, simile a quella utilizzata in CORTEX, in cui i dati del contesto possono essere ricevuti dai moduli dell'applicazione che ne hanno bisogno.

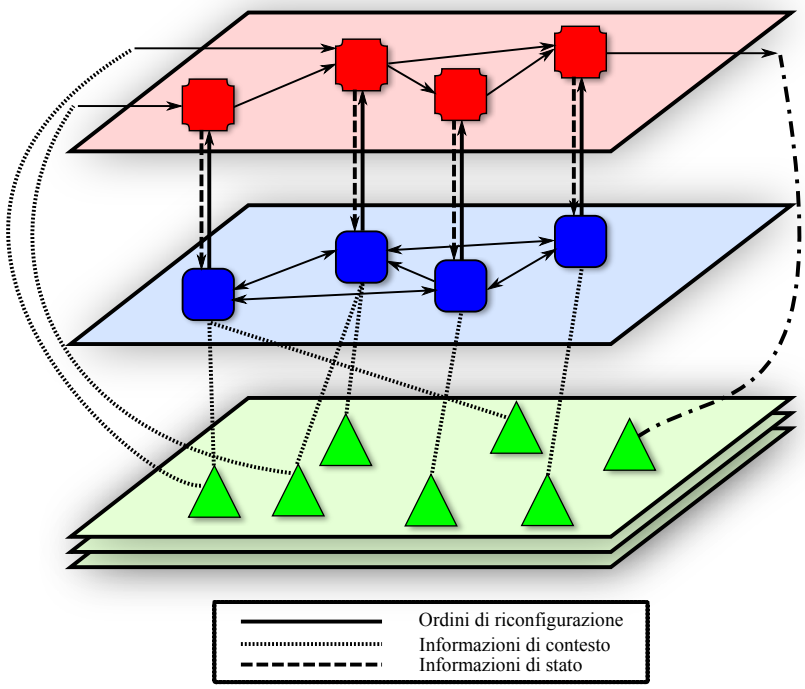


Figura 5.1: Suddivisione a strati di una applicazione ASSISTANT

5.3 Struttura delle applicazioni ASSISTANT

La struttura di una applicazione riprende quella del modello di ASSIST, formata da un insieme di moduli che cooperano attraverso l'invio di dati su stream. I singoli moduli mantengono tutte le caratteristiche viste nel capitolo 4: possono essere al loro interno sequenziali o paralleli, e sono espressi con gli stessi costrutti. Vista la relativa indipendenza tra moduli, abbiamo deciso di inserire i concetti di adattività e context awareness all'interno di ogni modulo, e non al livello dell'intera applicazione.

Per semplificare la modellazione abbiamo separato la logica dell'applicazione in "strati" cooperanti, come illustrato in figura 5.1.

- **RED**: rappresenta lo strato "applicativo", modella i moduli computazionali le interazioni tra essi.

- **BLUE**: lo strato di gestione del comportamento dell'applicazione, permette di introdurre adattività e context-awareness nel modello.

- **GREEN**: modella le interfacce di contesto, le cui informazioni sono utilizzate dallo strato blu per prendere decisioni.

5.3.1 RED

In questo strato ritroviamo la concezione di applicazione tipica di ASSIST: un grafo generico di moduli eventualmente paralleli collegati da stream, che modellano l'aspetto "computazionale" di una applicazione pervasiva. Definire adattività a livello dei singoli moduli richiede in essi la predisposizione ad eseguire differenti computazioni, adatte a differenti risorse e contesti. Le varie implementazioni di un modulo sono perciò definite a questo strato, e l'interazione con il BLUE permette di cambiarle in base all'ambiente attuale. Questa variazione avviene sui singoli moduli in modo indipendente: non vogliamo che la riconfigurazione di un modulo richieda modifiche a quelli a lui collegati, per non incorrere in effetti "a catena".

5.3.2 BLUE

Vediamo ora lo strato che ci permette di definire i comportamenti adattivi di una applicazione; questo non era presente in ASSIST e, insieme al green, incapsula le principali novità di ASSISTANT.

A questo strato l'applicazione è vista come un insieme di entità, i *Manager*, che implementano le politiche adattive dell'applicazione. Interagiscono con il green, dal quale ricevono eventi per acquisire conoscenza del contesto, e in base a queste informazioni e allo stato dei moduli possono richiedere al red un cambio di comportamento. Abbiamo detto che i differenti comportamenti (implementazioni) sono già disponibili nel red, che però non decide autonomamente se e quando utilizzarli.

Durante lo sviluppo di ASSISTANT abbiamo pensato ad uno schema "partizionato" per i manager. Concettualmente potremmo avere un unico manager che ha una visione di tutte le informazioni del contesto e comanda tutti i moduli del red, ma abbiamo pensato ad una suddivisione che rispecchiasse in modo più naturale il concetto di adattività dei singoli moduli. Per questo, come nella figura, abbiamo esattamente un manager per ogni modulo. I manager, a questo punto, hanno bisogno solo di una parte delle informazioni di contesto, quelle richieste per il singolo modulo. Per questo motivo, spesso useremo il termine Manager per indicare *il manager del singolo modulo.*

Nella figura sono presenti anche connessioni tra manager. Riteniamo quest'ultime fondamentali, nell'ottica di poter definire anche delle politiche adattive per l'intera applicazione; in questo modo un manager può richiedere modifiche dei comportamenti *anche* sugli altri moduli.

I manager hanno inoltre la possibilità di richiedere una modifica dell'allocazione dei moduli sulle risorse, necessaria per gestire un ambiente fortemente dinamico come quello del pervasive computing.

Infine assumiamo i manager come entità robuste ed affidabili, sempre presenti all'interno dell'applicazione. Per questo potrebbero essere necessarie tecniche di replicazione che ne garantiscano la presenza anche in caso di disconnessioni o guasti.

Le informazioni del contesto e delle risorse arrivano ai manager sotto forma di *eventi*. In questo modo possiamo unificare la gestione di adattività e context-awareness: in entrambi i casi ci aspettiamo infatti una modifica del comportamento dell'applicazione; possiamo perciò dire che, in base al tipo di evento ricevuto dal manager, questi chiederà una modifica dell'applicazione, ottenendo:

- **un comportamento adattivo** se l'evento era relativo alle risorse attualmente disponibili;
- **un comportamento context-aware** se l'evento era relativo ad informazioni del contesto.

Ovviamente, perché questa tecnica sia funzionale e abbastanza generale per descrivere molte tipologie di applicazioni pervasive, è necessario avere una astrazione di *evento* molto potente; prevediamo la possibilità di definizione di eventi da parte del programmatore, l'utilizzo di tecniche di inferenza e di algoritmi paralleli per derivare nuovi eventi complessi.

L'utilizzo di eventi rende i moduli concettualmente *reattivi*, perché modificano il loro comportamento al verificarsi di una determinata condizione. Questo modello, per quanto potente, non permette la definizione nativa di comportamenti proattivi (come quelli di Aura), in cui il sistema cerca di prevedere le necessità future e su queste previsioni basa il proprio comportamento.

Siamo comunque convinti, anche se questo aspetto è da studiare e sperimentare, che questo modello permetta di *emulare* comportamenti proattivi, tramite la definizione di eventi complessi che possono essere generati, a partire dagli eventi ottenuti dal contesto, con meccanismi di previsione analoghi a quelli di Aura; in questo modo, a differenza di Aura, i meccanismi previsionali possono essere specificati dal programmatore, offrendo anche maggiore flessibilità.

5.3.3 GREEN

Lo strato verde modella "il contesto", le modalità con cui le informazioni catturate dall'ambiente (ad esempio tramite sensori) vengono gestite ed elaborate per formare gli eventi utilizzati dall'applicazione. Modella anche il flusso di informazioni opposto, ovvero quello dall'applicazione verso gli attuatori presenti nell'ambiente. È quindi definito in funzione del blue, ma può essere utilizzato anche dal red.

Si noti come, infatti, in alcuni casi le informazioni di contesto potrebbero essere utilizzate *anche* a livello applicativo, non per gestire la context-awareness, ma come dati da elaborare. Nell'applicazione di gestione delle emergenze, ad esempio, i dati raccolti da un insieme di sensori sono utilizzati per eseguire gli algoritmi previsionali. Allo stesso modo, questo permette al programmatore di definire eventi "complessi", derivati da una post-elaborazione dei dati acquisiti dalle interfacce di contesto. Questo può essere ottenuto semplicemente utilizzando dei moduli (comuni parmod) che trattano i dati ricevuti dal contesto come stream, li elaborano e producono nuovi eventi. Vedremo successivamente come introdurre tutte queste possibilità all'interno del modello di ASSISTANT.

In questa ottica, gli eventi non vengono prodotti *solo* dallo strato green, ma anche dal red e (abbiamo visto prima) dallo stesso blue per la coordinazione di manager. Abbiamo perciò differenti fonti di generazione di eventi. Tra queste, il green rappresenta sicuramente una parte importante: tutti e soli i dispositivi che non sono "programmati" direttamente dall'utente. Rappresenta infatti la modalità con cui le entità esterne all'applicazione, e programmate da altri, interagiscono con essa.

Nella nostra ottica le interfacce nel green non sono scritte dal programmatore, ma piuttosto dai produttori di sensori o dagli sviluppatori del modello.

Spendiamo un'ultima parola per chiarire il significato di questi strati. Innanzitutto, non sono da confondersi con i livelli. La rappresentazione grafica in questo senso può essere fuorviante, in quanto questi tre si trovano tutti *al livello applicativo*, e modellano differenti aspetti della stessa applicazione. Non sono implementati uno sopra l'altro, ma *cooperano* tra loro. In questo senso, il BLUE utilizza i dati ottenuti dal GREEN, ma quest'ultimo **non è** il supporto del primo, piuttosto saranno implementati tutti dal **supporto di ASSISTANT.**

5.4 Adattività in ASSISTANT

Vediamo ora di spiegare in modo dettagliato come si è scelto di gestire il concetto di adattività all'interno di ASSISTANT.

Abbiamo già parlato del fatto che le modifiche del comportamento possono essere anche non banali e sono relative alle singole applicazioni; richiedono perciò un'interazione con il programmatore. A questo proposito sono stati inseriti meccanismi per

1. definire un insieme di comportamenti possibili per il modulo.

Nel caso di applicazioni parallele strutturate sappiamo che è possibile definire in modo automatico, senza bisogno dell'intervento del programmatore, differenti versioni del solito modulo; queste possono differire per *implementazione della forma parallela* oppure per *grado di parallelismo*.

Non vorremmo perdere queste caratteristiche, ma ci rendiamo conto che non sono sufficienti per ambienti fortemente dinamici. A queste aggiungiamo quindi la possibilità di definire comportamenti completamente diversi per il modulo, che possano realizzare la stessa funzionalità in modo differente, ed essere eseguiti in ambienti molto diversi tra loro.

Queste due modalità concorrono nel fornire l'insieme di comportamenti possibili del modulo, che può quindi avere in modo nativo e senza sforzo ulteriore una forma base di adattività, estendibile dal programmatore con un meccanismo molto flessibile.

L'esistenza di differenti comportamenti non basta però per ottenere un'applicazione adattiva; dobbiamo anche fornire strumenti per

2. decidere quando e come modificare il comportamento di un modulo.

Le informazioni dell'ambiente (dati di contesto e sulle risorse) sono utilizzati per definire degli eventi; al verificarsi di un evento il modulo può cambiare comportamento.

Il compito del programmatore è definire le operazioni da eseguire al verificarsi di alcuni eventi. Ogni evento porta con se un valore, che potrebbe essere booleano nel caso di eventi base (ad esempio "la connessione di rete è presente") oppure rappresentare qualcosa di più complesso, come "l'attuale banda massima è x", oppure "vorrei una fidelity y". In ogni caso il modulo può controllare il valore ricevuto e decidere se riconfigurarsi in base ad esso; inoltre può utilizzare il suo stato interno per prendere decisioni ancora più complesse.

In teoria la strutturazione parallela del modulo ci dà abbastanza informazioni per conoscere tutte le comunicazioni interne ed esterne al modulo, mentre da un'analisi statica del codice si dovrebbe poter ottenere una buona approssimazione del numero e della tipologia di istruzioni eseguite dalle parti sequenziali. Grazie a queste informazioni dovrebbe essere possibile *anche* inserire meccanismi *automatici* di gestione dell'adattività, che scelgano il migliore comportamento attuale per i moduli, conoscendo l'ambiente di esecuzione.

Nell'ottica della programmazione parallela il termine di paragone sul quale decidere è sempre stata la performance (tempo di servizio o latenza) del modulo; stiamo però studiando altre funzioni obiettivo per ottimizzare, ad esempio, il consumo della batteria per i dispositivi mobili (esattamente come hanno fatto i ricercatori di Odyssey). Se questi studi si verificheranno validi, potremmo inserire anche meccanismi di adattività *autonomi* all'interno del modulo ASSISTANT.

Attualmente, però, deleghiamo la definizione di questi comportamenti al programmatore, che conosce l'applicazione e (almeno in questo momento) può definire politiche migliori di quelle automatiche che conosciamo.

In ASSISTANT la modifica del comportamento di un modulo viene chiamata "**riconfigurazione**". Abbiamo deciso di distinguere le riconfigurazioni in due differenti tipologie, derivate dai due tipi di adattività definiti precedentemente in quanto queste presentano caratteristiche differenti e, al fine di garantire prestazioni migliori, dovrebbero essere trattate in modo differente.

5.4.1 Riconfigurazioni non funzionali

Questa modalità di riconfigurazione rappresenta una estensione di quella introdotta in ASSIST con [79]; con ASSISTANT ci proponiamo di superare le difficoltà incontrate da tale approccio, e soprattutto offrire al programmatore la possibilità di definire comportamenti adattivi personalizzati e non solamente quelli statici presentati in ASSIST.

Con questo termine ci riferiamo ad una modifica del modulo che non tocca la sua struttura parallela né il codice utente. Esattamente come in ASSIST, infatti, possiamo modificare la richiesta di risorse di un modulo parallelo cambiandone il grado di parallelismo, oppure migliorare le prestazioni spostandone alcune parti su nodi differenti.

Questo tipo di modifica, ereditata dal precedente ambiente di sviluppo, viene gestita *interamente* dal supporto e non necessita di lavoro addizionale da parte del programmatore durante la definizione dell'applicazione;

possiamo perciò dire che si tratta di una modifica che si ripercuote solo sull'implementazione del modulo, e non sui livelli superiori.

Modifica del grado di parallelismo

Per chiarire le idee riportiamo un esempio, ragionando su un programma ASSIST. Parliamo di un modulo per la moltiplicazione matrice-vettore, parallelizzato nella forma data-parallel *map*; il codice di un parmod per tale operazione è illustrato nel listato 5.1. In questo caso abbiamo definito un partizionamento per righe della matrice (il metodo più comune per questo algoritmo) che ci permette di raggiungere un grado di parallelismo massimo pari al numero di queste. Non abbiamo stencil, ma solo una fase di scatter ed una di gather; il partizionamento inoltre rende la versione parallela abbastanza bilanciata. A parte casi particolari, questa applicazione dovrebbe avere una buona scalabilità, e garantirci un aumento di prestazioni lineare sul numero di nodi utilizzato.

Assumiamo che per motivi legati all'applicazione questo modulo debba garantire un tempo di servizio minore di una costante, TS. In questo caso il programmatore può chiedere di aumentare il numero di nodi se il tempo di servizio attuale è maggiore di TS, e diminuire il grado di parallelismo se bastano meno nodi per rimanere sotto la soglia di TS. Questa seconda operazione ha senso nell'ottica di liberare nuove risorse per altre applicazioni (o moduli) nel caso di esecuzione su un cluster, e di risparmio energetico nel caso di esecuzione su un insieme di nodi mobili. In figura 5.2 mostriamo il cambiamento che avverrebbe con questo tipo di riconfigurazione, marcando le modifiche in rosso. Come si può vedere dall'immagine, il programma rimane *identico*, ma cambia la sua implementazione, che utilizza più nodi dopo la riconfigurazione.

Ovviamente questo ragionamento è molto "leggero" e non tiene conto delle caratteristiche dell'architettura su cui viene eseguito; ci è servito solo per mostrare l'utilità delle riconfigurazioni non funzionali. In realtà uno studio più accurato ci porterebbe a stabilire che il tempo impiegato per distribuire i dati aumenta con l'aggiunta di nodi (abbiamo più comunicazioni per la scatterizzazione di A ed inviamo più copie di B) e potrebbe perciò influire negativamente sulle prestazioni.

Oltretutto fino a questo momento abbiamo pensato all'implementazione classica di ASSIST, che però non è l'unica e potrebbe variare in futuro. Questi problemi si presentano in modo maggiore su parallelizzazioni con stencil, dove le comunicazioni tra i nodi dipendono fortemente dal tipo di ottimizzazioni che il supporto è riuscito ad introdurre[37].

```
parmod matrixCalc(input_stream double A[N][N], double B[N]
    output_stream double result[N]){
  topology array [i:N] Pv; stream double out_matrix;
  do input_section{
    guard1: on , , A && B {
      distribution A[*i0][] scatter to Pv[i0];
      distribution B broadcast to Pv;
    }
  }while(true)
  virtual_processors {
    elaborazione( in guard1 out out_matrix){
      VP i=1..N { Fmul(in A[i][],B[] out out_matrix); }
    }
  }
  output_section{
    collects out_matrix from ALL Pv[i] {
      int elem; double ris[N];
      AST_FOR_EACH(elem) {ris[i]=elem;}
      assist_out(result, ris);
    }<>;
  }
}
proc Fmul(in double A[N], double B[N] out double C)
$c++{ double r=0;
  for (int k=0; k<N; ++k) {r += A[k]*B[k];}
  C = r; }c++$
```

Listato 5.1: Modulo ASSIST per la moltiplicazione matrice-vettore parallela

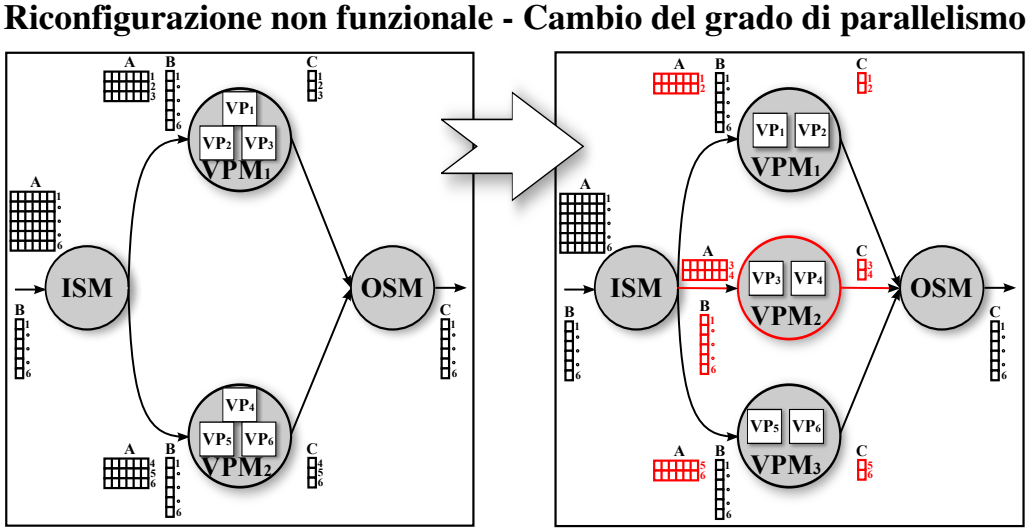


Figura 5.2: Modifica del grado di parallelismo del modulo sopra descritto: un esempio di riconfigurazione non funzionale.

Per questi motivi il programmatore ASSISTANT non può essere a conoscenza dei modelli dei costo della propria applicazione, in quanto sono strettamente dipendenti dall'architettura di esecuzione e dall'implementazione fornita dal supporto.

Infine, se anche i modelli di costo fossero fissati, il programmatore difficilmente potrebbe conoscerli a fondo, e rischierebbe di utilizzarli nel modo sbagliato.

In questo scenario le riconfigurazioni non funzionali perderebbero gran parte del proprio significato, in quanto difficilmente si potrebbe decidere se conviene riconfigurarsi ed in che modo.

In ASSISTANT vogliamo inserire meccanismi per offrire nativamente al programmatore questi modelli di costo, per permettergli di valutare se la riconfigurazione può rivelarsi valida oppure no. In questa ottica pensiamo di offrire un insieme di funzioni, che forniscano stime del comportamento del modulo al variare del grado di parallelismo. Come detto prima, per ora conosciamo solo stime legate alla performance, ma contiamo di introdurne altre col tempo.

Modifica dell'allocazione del modulo sui nodi disponibili

Proponiamo ora un secondo esempio importante di riconfigurazione funzionale, sempre a partire dall'esempio del listato 5.1. Pensiamo ad una esecuzione su un ambiente molto dinamico, dove nuovi nodi entrano regolarmente nel sistema. In questo caso, anche senza modificare il grado di parallelismo, potremmo essere interessati a "spostare" parte della computazione su uno dei nuovi nodi del sistema. Questo per motivi prestazionali (il nuovo nodo è più veloce) o anche per problematiche di tolleranza alle disconnessioni (ad esempio, la probabilità che un nodo si disconnetta potrebbe essere proporzionale al suo tempo di rimanenza nella rete).

Un altro esempio importante si può trovare durante l'esecuzione su dispositivi a batteria (portatili,pda,etc): se uno dei nodi utilizzati raggiunge un livello di carica critico, si dovrebbe spostare la computazione su uno libero, anche se con prestazioni inferiori. Abbiamo riportato questo esempio nella figura 5.3, dove nella configurazione iniziale il sistema aveva scelto due laptop per la parte computazionalmente intensiva del modulo (la sezione *virtual_processors*), e solo le parti di distribuzione e collezione venivano eseguite su terminali più lenti (i PDA). Quando uno dei portatili raggiunge un livello di batteria critico, l'allocazione delle risorse viene modificata, spostando i processi allocati nel laptop su un PDA precedentemente in standby.

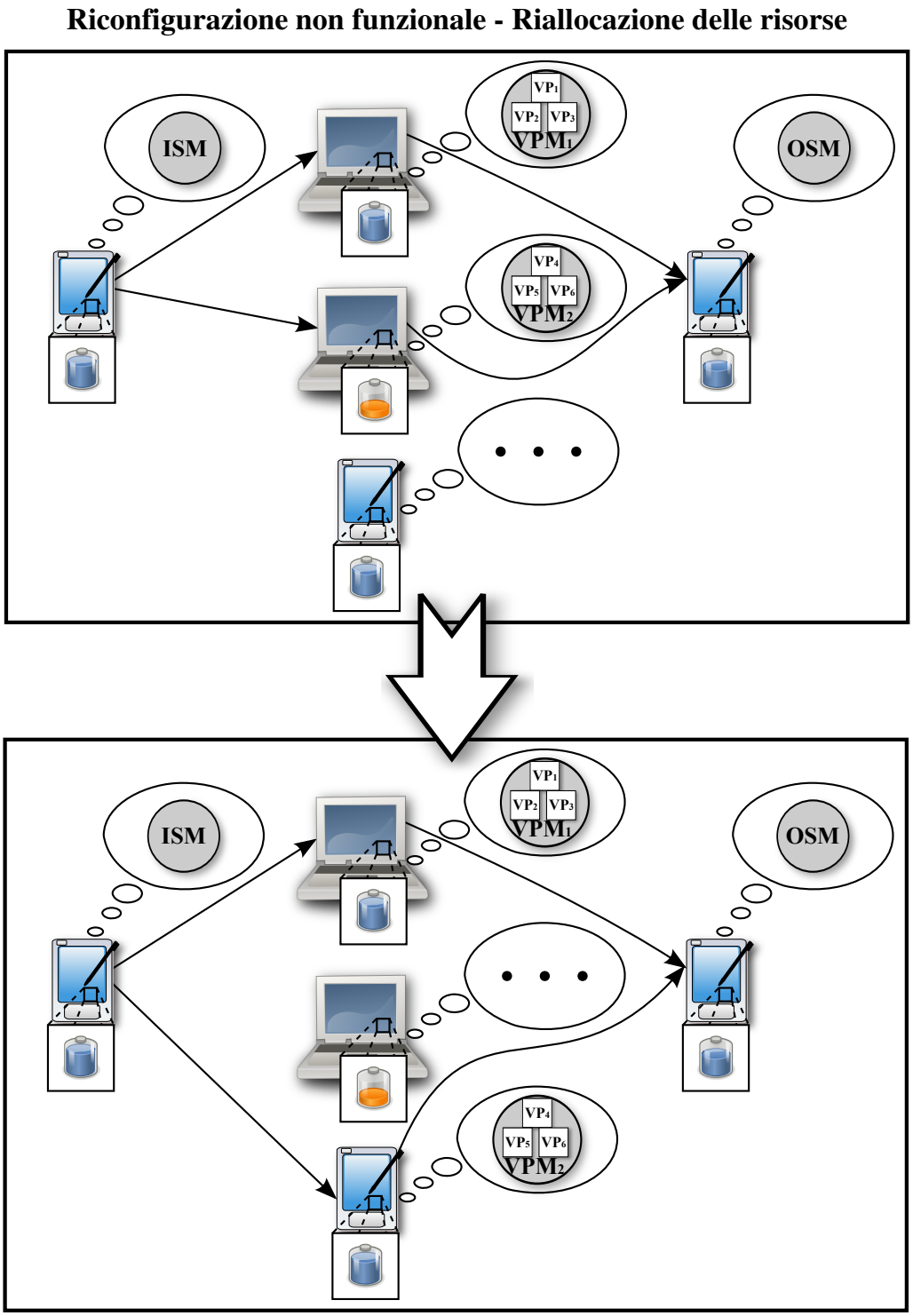


Figura 5.3: Un secondo esempio di riconfigurazione non funzionale: modifica dell'allocazione delle risorse in caso di livello di batteria critico.

Questo secondo esempio ci pone un problema non ancora considerato, ovvero la conoscenza dei dispositivi presenti nell'ambiente. Nell'esempio di prima, infatti, il programmatore si limitava a chiedere la modifica del grado di parallelismo al sistema; qui, invece, dovremmo avere conoscenza di tutti i nodi utilizzati dal modulo stesso e di eventuali nodi "liberi". Questo complicherebbe notevolmente la modellazione di una richiesta di riconfigurazione, richiederebbe nuovamente al programmatore di conoscere dettagli dell'implementazione e sarebbe comunque difficile da gestire.

Per questo motivo, per il momento non pensiamo di inserire questa tipologia in ASSISTANT, ma questo non vuol dire che il problema verrà ignorato per sempre: a livello di supporto inseriremo i meccanismi per permettere anche questo tipo di riconfigurazione[1], e quando riusciremo a definire meccanismi automatici per la decisione delle riconfigurazioni non funzionali, in questi inseriremo anche la logica per gestire casi come quello dell'esempio.

Un lettore attento noterà che, in fondo, anche nella modifica del grado di parallelismo emergono problemi di allocazione su nuovi nodi; in quel caso abbiamo deciso di lasciar scegliere al supporto quali rimuovere e/o inserire, anticipando quindi alcuni meccanismi di scelta dei nodi che in futuro verranno formalizzati in modo migliore.

5.4.2 Riconfigurazioni funzionali

In questo tipo di riconfigurazione si può modificare completamente il comportamento del modulo. Concettualmente questa operazione corrisponde a "sostituire" il modulo con uno differente; ovviamente, per garantire una corretta esecuzione dell'intera applicazione, i due moduli in gioco devono essere in qualche modo "compatibili". Questi concetti sono stati studiati in modo molto accurato per la *programmazione a componenti*, ed hanno portato alla definizione di interfacce ben definite per l'interazione tra essi. In questi modelli un componente può essere sostituito con uno nuovo, a patto che i due mantengano la stessa semantica e le stesse interfacce.

In ASSISTANT possiamo trovare una certa analogia tra moduli e componenti: entrambi sono entità autonome, con interfacce ben definite, componibili per realizzare applicazioni complesse. Abbiamo perciò deciso di utilizzare un concetto analogo a quello dei componenti: due moduli sono "compatibili" se mantengono le stesse interfacce (gli stream di ingresso e di uscita) e la stessa semantica. Una riconfigurazione funzionale corrisponde alla sostituzione di un modulo con uno ad esso compatibile. In figura 5.4 abbiamo illustrato

[1] Che poi rappresenta un caso particolare della modifica del grado di parallelismo in cui il nuovo grado è uguale al vecchio.

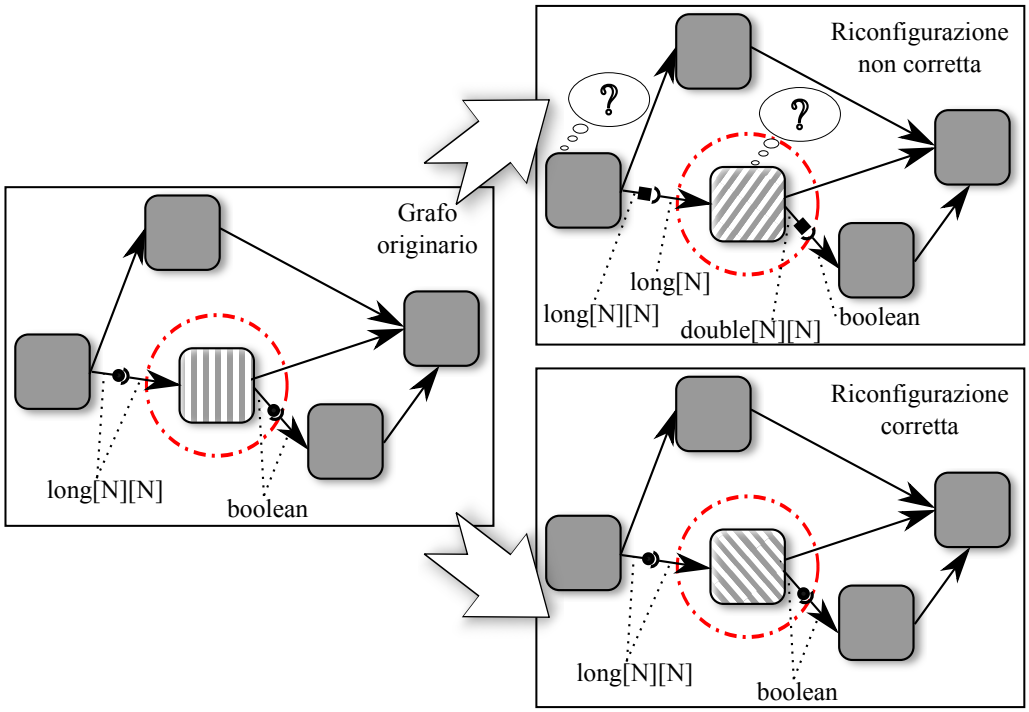


Figura 5.4: Esempi di correttezza di una riconfigurazione funzionale

due modifiche: una corretta, che rispetta le interfacce tra moduli, ed una non corretta, dove i moduli non riuscirebbero più a comunicare.

In ASSISTANT il modello di programmazione si occupa di mantenere invariate le interfacce esterne: lo sviluppatore, infatti, descrive differenti "implementazioni del parmod", chiamate nel linguaggio **operation**, che possono differire tra loro sia come struttura parallela, sia come codice utente; ma gli stream di ingresso/uscita sono descritti a livello di parmod e non di operation, garantendo così nativamente la compatibilità sintattica tra operation dello stesso modulo.

Modifica del codice sequenziale

Vediamo un caso in cui si possa essere interessati *solo* alla modifica del codice sequenziale eseguito, senza voler modificare la struttura parallela del modulo. Riprendiamo l'esempio della moltiplicazione matrice-vettore visto prima; questa volta proponiamo una versione di tipo *farm*, riportata nel listato 5.2.

Anche per un problema così semplice (che non offre differenti algoritmi per risolverlo) possiamo pensare ad una versione ottimizzata per architet-

```
parmod matrixCalc(input_stream double A[N][N], double B[N]
    output_stream double result[N]){
  topology none Pv;
  do input_section{
    guard1: on , , A && B {
      distribution A on_demand to Pv;
      distribution B on_demand to Pv;
    }
  } while(true)
  virtual_processors {
    elaborazione( in guard1 out result){
      VP { Fmul(in A,B out result); }
    }
  }
  output_section{ collects result from ANY Pv; }
}
proc Fmul(in double A[N][N], double B[N] out double C[N])
$c++{
  for (int i=0;i<N;i++){ double r=0;
    for (int k=0; k<N; k++) { r += A[i][k] * B[k]; }
    C[i] = r; }
}c++$
```

Listato 5.2: Modulo ASSIST per la moltiplicazione matrice-vettore parallela in versione farm

ture particolari. Consideriamo, ad esempio, i dispositivi mobili tipo PDA, SmartPhone, Navigatori, etc. Questi dispositivi sono, dal punto di vista computazionale, molto simili tra loro: utilizzano gli stessi sistemi operativi, processori di tipo ARM e sono dotati di connettività wireless. I processori di cui sono dotati, pur avendo un rapporto consumo/prestazioni molto elevato, non sono pensati per elaborazioni pesanti; ad esempio il supporto nativo per calcoli in virgola mobile è delegato ad un coprocessore esterno[13], non disponibile su tutti i modelli, ed in sua assenza il calcolo deve essere eseguito a software.

In questo scenario ci immaginiamo un guadagno prestazionale significativo utilizzando dati con minore precisione, ed eseguiamo l'intero calcolo su float invece che su double. Questa modifica intacca solo il codice utente, e non modifica la struttura parallela. In questo caso basta modificare la *proc* invocata dai VP, sostituendola con quella riportata nel listato 5.3. Il risultato è quindi un modulo ASSIST più veloce su alcune particolari architetture, al costo di una precisione minore. Nell'ottica dei sistemi adattivi stiamo perciò modificando la fidelity del dato prodotto, ma questa volta al livello

dell'applicazione, modificando l'algoritmo.

La riconfigurazione appena descritta trova la sua applicazione naturale all'interno di un ambiente fortemente pervasivo. Inizialmente il modulo potrebbe essere in esecuzione su un laptop di ultima generazione. Da qualche anno, ormai, questi dispositivi sono dotati di processori dual core, perciò anche se allocato su un singolo nodo, il modulo utilizzerebbe un grado di parallelismo 2, per sfruttare meglio la macchina. Il laptop, per qualche motivo, potrebbe non essere più disponibile: la batteria si è scaricata, oppure il proprietario lo ha spostato, o ancora si trova in una zona con interferenze che rende inutilizzabile la rete wireless. Il modulo verrà spostato sulle risorse rimaste, che potrebbero essere solo alcuni pda o smartphone. In questo caso oltre allo spostamento del modulo si richiederà il passaggio alla versione ottimizzata, che verrà eseguita su dispositivi mobili. Questa riconfigurazione è mostrata nella figura 5.5.

Anche questo caso necessita, ovviamente, di alcuni commenti. Innanzitutto si rivela necessario poter definire, nel modello, il tipo di risorse pensate per ogni versione del modulo: il primo era per computer standard, il secondo per PDA o SmartPhone non troppo evoluti.

Nel codice utente si rivela necessario effettuare manualmente la conversione tra *double* e *float* per eseguire l'algoritmo con precisione inferiore, in quanto il tipo di dati trattato al livello di ASSIST rimane il *double*. Probabilmente questa cosa potrebbe essere migliorata, ottimizzata ed eseguita automaticamente dal livello di supporto: non scordiamoci, infatti, che il modulo è studiato per essere eseguiti in ambienti pervasivi, e le rappresentazioni dei dati sono dipendenti dall'architettura. Nell'esempio abbiamo il modulo che produce il dato è un laptop, quindi verosimilmente con architettura x86. Il trasferimento dei dati verso una *ARM* potrebbe richiedere comunque una manipolazione dei dati per renderli compatibili. Se i dati sono già manipolati dal supporto, possiamo approfittarne per chiedere una ulteriore conversione tra tipi compatibili (come int e long o float e double) automatica, eseguita dal supporto. In questo caso il codice della proc verrebbe quello del listato 5.4, aumentando la leggibilità e le prestazioni.

Infine, un ultimo commento importante riguarda il possibile utilizzo *contemporaneo* di differenti codici sequenziali all'interno del modulo. In un ambiente eterogeneo, infatti, nell'ottica di utilizzare **tutte** le possibili risorse, potremmo trovarci di fronte alla scelta di quale ottimizzare. L'algoritmo visto prima che trasforma i dati durante la computazione diventa più lento sui processori moderni, in quanto la conversione dei dati non permette l'utilizzo di istruzioni vettoriali, mentre il costo delle moltiplicazioni sulle due

```
proc FmulARM(in double A[N][N], double B[N] out double C[N])
$c++{
  for (int i=0;i<N;i++){
    float r=0;
    for (int k=0; k<N; k++) {
      r += (float)A[i][k] * (float)B[k];
    }
    C[i] = (double)r;
  }
}c++$
```

Listato 5.3: Proc modificata per una esecuzione più veloce su processori senza FPU

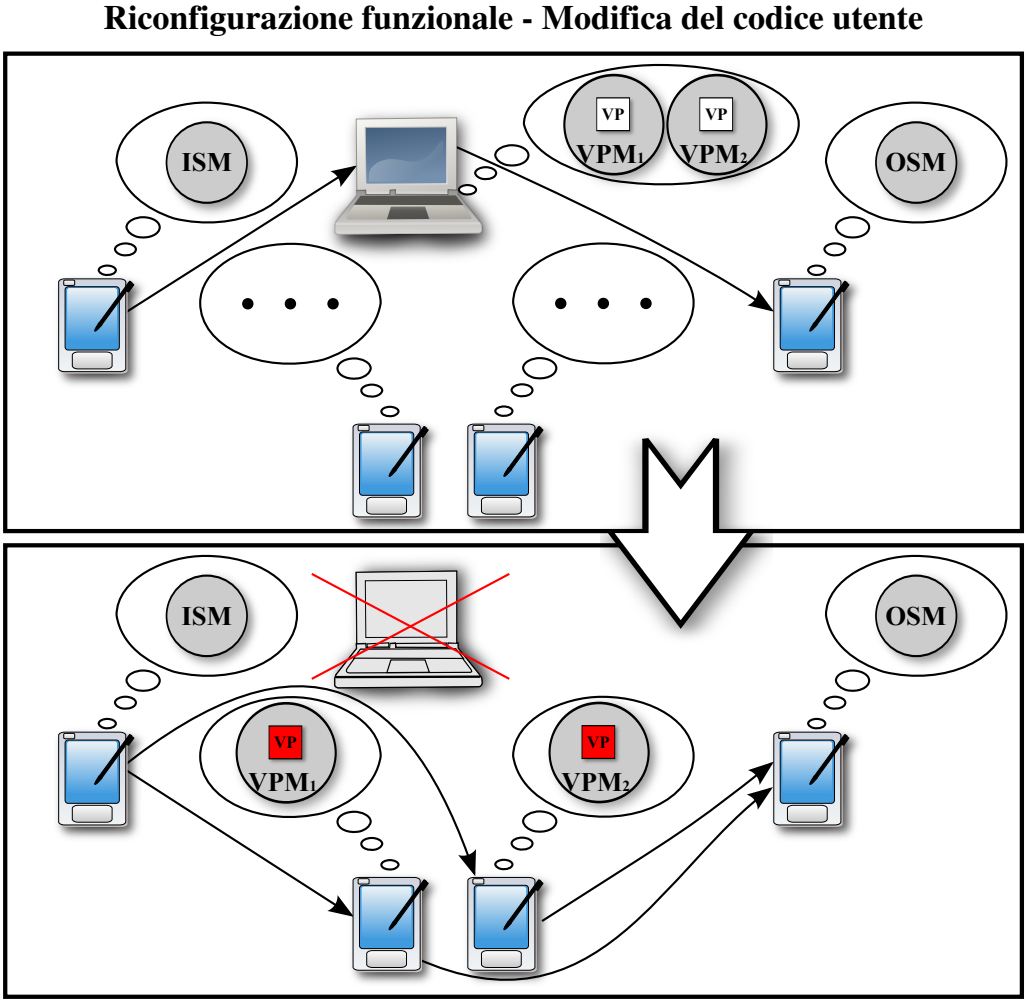


Figura 5.5: Esempio di riconfigurazione funzionale tramite modifica del solo codice sequenziale

```
proc FmulARM_float(in float A[N][N], float B[N] out float C[N])
$c++{
  for (int i=0;i<N;i++){
    float r=0;
    for (int k=0; k<N; k++) {
      r += A[i][k] * B[k];
    }
    C[i] = r;
  }
}c++$
```

Listato 5.4: Proc che utilizza tipi differenti (ma compatibili) con quelli del modulo ASSIST

precisioni è pressoché identico. Potrebbe essere interessante studiare casi in cui, su ogni nodo, viene eseguita una versione del modulo differente. In un farm questa cosa è realizzabile senza problemi, in quanto i singoli worker non comunicano. Ovviamente, su un data-parallel di qualsiasi tipo questo non è possibile, in quanto tutti i nodi cooperano per produrre una unica soluzione, e devono perciò adottare il solito algoritmo.

Esempi più complessi Quello della moltiplicazione matrice-vettore è solo un esempio semplice, utilizzato per far capire i concetti senza dover spiegare un problema complesso. Ci sono molti casi in cui queste riconfigurazioni sono più complesse, e possono utilizzare un algoritmo sensibilmente differente. Nell'ottica della risoluzione di sistemi lineari, per esempio, esistono molte tecniche differenti, a partire da metodi diretti o iterativi. Per ogni classe esistono poi differenti algoritmi, tutti con caratteristiche prestazionali e di precisione differente.

Applicabilità alle forme data-parallel Questa tipologia di riconfigurazione si presta principalmente per parallelizzazioni di tipo farm. La caratteristica importante di questo tipo di parallelizzazione, infatti, è che tutti i dettagli dell'algoritmo sono racchiusi nel codice sequenziale e (a parte i dati in entrata/uscita) non emergono nella struttura parallela. Da questo punto di vista, i casi data-parallel sono differenti: nel caso di presenza di stencil, questi sono strettamente legati all'algoritmo. Una modifica di quest'ultimo modificherebbe quasi sicuramente anche lo stencil, le comunicazioni tra nodi e quindi la struttura parallela. Nella map invece, nonostante l'assenza di comunicazioni tra nodi, emerge lo stato utilizzato dall'algoritmo e la modalità di distribuzione del dato da elaborare.

In questi casi, perciò, difficilmente potranno essere introdotte modifiche riguardanti *solamente* il codice sequenziale, ma con alta probabilità queste interesseranno anche la struttura parallela, ed apparterranno perciò alla tipologia descritta successivamente.

Modifica della struttura parallela

Parliamo ora del caso più importante di riconfigurazione funzionale, dove il modulo cambia completamente comportamento, modificando anche la struttura parallela. Per non affaticare il lettore non introduciamo nuovi esempi, ma sfruttiamo quelli già discussi: abbiamo infatti due versioni del modulo fin ora utilizzato: uno *farm* (listato 5.2), l'altro *map* (listato 5.1). Delle due versioni non ne esiste una "migliore in assoluto", in quanto in base alle caratteristiche dei nodi di esecuzione una si comporta meglio dell'altra, e viceversa.

In condizioni ottimali le due forme garantiscono un tempo di servizio analogo, ma la forma *map* offre anche una diminuzione della latenza proporzionale al numero dei nodi, e l'occupazione di memoria richiesta sui singoli nodi risulta nettamente inferiore; per questi motivi saremmo portati a preferire la forma data parallel.

In realtà le prestazioni di questa versione vengono fortemente ridotte in caso di sbilanciamento del carico; in ambienti molto eterogenei rischia perciò di essere penalizzata rispetto ad una versione *farm* che si bilancia automaticamente.

In presenza di un cluster, o anche di un semplice multicore ad alte prestazioni, possiamo utilizzare senza problemi la versione *map*, che ci garantisce ottime prestazioni in questo ambiente omogeneo. Se questo non fosse disponibile (ad esempio perché necessario per un modulo più pesante oppure per una applicazione differente con priorità maggiore), passando ad un gruppo di dispositivi molto eterogenei, conviene cambiare il modulo in *farm*. Infine, potremmo essere obbligati a tornare alla versione *Map* anche su macchine eterogenee, a causa della richiesta eccessiva di risorse di quella stream parallel. Queste due trasformazioni sono illustrate in figura 5.6.

Inizialmente il programma è eseguito su una macchina quad core di ultima generazione; successivamente questo computer viene riservato per una applicazione con priorità maggiore e perciò non è più disponibile per il nostro modulo. Le macchine rimaste vengono utilizzate, ma a causa della loro eterogeneità passiamo ad una versione di tipo *farm*. Si scelgono due laptop in quanto questi offrono prestazioni sufficienti per la computazione. Infine, uno dei due portatili esce dal sistema, e siamo obbligati ad utilizzare i dispositivi

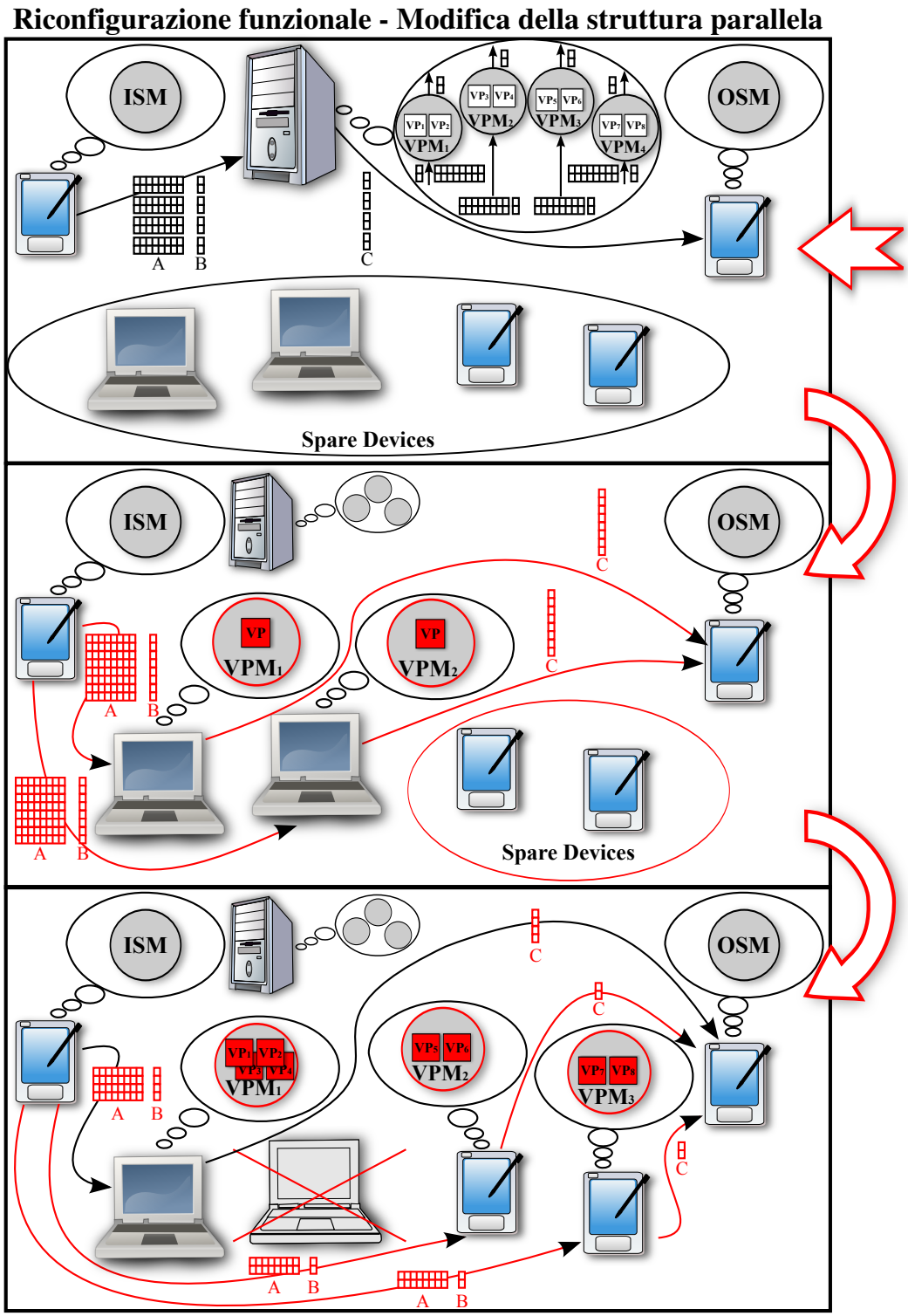


Figura 5.6: Esempi di riconfigurazione funzionale tramite modifica della forma parallela. Le modifiche in rosso

mobili presenti. A questo punto il farm non può più essere eseguito a causa di limiti della memoria, e perciò torniamo alla versione data-parallel.

Riportiamo alcuni brevi commenti anche per questo tipo di riconfigurazione, quella in assoluto più potente introdotta in ASSISTANT.

Con questa possiamo modificare ogni particolare del modulo, e definire perciò comportamenti anche profondamente differenti. Nel nostro caso, per questioni di spazio, ci siamo limitati a descrivere due forme parallele per lo stesso algoritmo. Già con queste siamo riusciti a trattare profondi cambiamenti nell'ambiente di esecuzione, in modo pressoché trasparente al resto dell'applicazione.

Un lettore attento avrà notato che in questo ultimo esempio oltre alla modifica della struttura parallela abbiamo anche modificato il numero di nodi utilizzati durante l'esecuzione. Questo comportamento è stato già illustrato precedentemente come una "riconfigurazione non funzionale". È chiaro che sia spesso necessario anche nel caso di questo tipo di riconfigurazioni, in quanto con buona probabilità differenti algoritmi avranno bisogno di una quantità di nodi differente. Potremmo considerare questa modifica come parte della riconfigurazione funzionale, oppure un passo ulteriore di ottimizzazione effettuato *dopo* di essa. Noi preferiamo considerarlo come una trasformazione unica, in quanto questo ci porterebbe ad un cambio di comportamento molto più veloce ed è comunque concettualmente aderente all'idea di un cambio dell'implementazione del modulo.

5.4.3 Differenze tra riconfigurazioni funzionali e non funzionali

Cerchiamo ora di capire per quale motivo abbiamo deciso di considerare le riconfigurazioni non funzionali come "diverse" da quelle funzionali. Le motivazioni di questa scelta nascono, ancora una volta, sia per motivi di ottimizzazione, sia per motivi "storici". Come abbiamo già detto, la prima è ereditata dal modello di ASSIST, nel quale è stata trattata come unica tecnica di adattività. Inoltre, non richiede l'interazione del programmatore, in quanto viene creata automaticamente dal supporto di ASSISTANT. Nonostante questo, niente vieterebbe di considerarla semplicemente come caso specifico di una più vasta gamma di riconfigurazioni.

In realtà il conservare la forma di parallelismo e il codice utente ci permette di inserire una ottimizzazione molto importante, che nell'altro caso, in generale, non è possibile.

Ricordiamo che i moduli ASSISTANT hanno una semantica "a stream", ovvero eseguono la stessa operazione ad ogni attivazione del modulo. Le riconfigurazioni, di qualsiasi tipo esse siano, devono mantenere una semantica corretta e ben precisa del modulo. A questo proposito, anche in applicazioni puramente sequenziali, è necessario poter definire in modo preciso cosa succede ai dati nel caso avvenga una riconfigurazione. Nel caso parallelo la situazione è ancora peggiore, in quanto in realtà stiamo modificando il comportamento di tanti programmi che cooperano insieme: garantire una forma di correttezza non è necessariamente banale né da ottenere, né da dimostrare.

Non possiamo "riconfigurare" un modulo in qualsiasi momento della sua esecuzione, in quanto questo porterebbe a risultati impredicibili ed inconsistenti. Al contrario, vogliamo garantire una semantica ben precisa nel caso di una riconfigurazione, che ci garantisca un risultato "valido", ovvero producibile da uno dei due comportamenti in gioco.

Per fare questo, un modo semplice (ma spesso anche l'unico, se non si hanno informazioni aggiuntive) è quello di assicurare che ogni attivazione del modulo viene trattata da un *singolo* comportamento. Questo vuol dire che la computazione in corso durante la riconfigurazione verrà eseguita *interamente* da una singola versione del modulo. Non è necessario che sia quella precedente alla riconfigurazione, in quanto si hanno due possibilità:

1. **continuare** l'elaborazione corrente con il vecchio comportamento, ed applicare il nuovo *solo* alle successive attivazioni del modulo;

2. **annullare** l'elaborazione corrente, e **rieseguirla** col nuovo comportamento.

Ovviamente entrambe le soluzioni hanno un senso, ed esistono casi un cui potrebbe essere meglio utilizzare la prima, altri in cui questa non è più praticabile, e siamo in qualche modo "costretti" ad utilizzare la seconda.

Tutto questo, però, nel caso "generale" di una qualsiasi riconfigurazione. Con le riconfigurazioni *non funzionali* il modello ad alto livello di ASSISTANT ci permette di fare qualcosa in più: ci offre una terza scelta che è *sempre* migliore delle due precedentemente discusse.

Infatti, la modellazione ad alto livello ci permette di definire dei punti, all'interno della computazione, in cui lo stato dell'intero modulo è "consistente". In questi momenti (che saranno poi utilizzati anche per ottenere tolleranza ai fallimenti) lo stato del modulo è ben definito e può essere utilizzato come punto di partenza di una nuova computazione.

In una prima approssimazione, raggiungere uno stato consistente per l'intero modulo richiede dei momenti di "sincronizzazione", dove le singole entità

del modulo si fermano, in attesa delle altre. Continuare la propria elaborazione porterebbe a modifiche dello stato interno rendendolo non più consistente. Questa limitazione può essere facilmente superata tramite tecniche di *checkpoint non coordinato*, dove queste entità salvano, in momenti ben precisi, il proprio stato interno mediante tecniche di checkpointing. Gli studi in [20] dimostrano come sia possibile utilizzare tali tecniche per formare checkpoint "validi" dello stato dell'intera applicazione. In questo modo possiamo definire uno stato consistente come l'ultimo checkpoint globale raggiunto, ed eliminare la necessità di sincronizzazione. La definizione di checkpoint consistenti permette di implementare in modo ottimizzato sia tolleranza a guasti/fallimenti, sia i meccanismi di riconfigurazione.

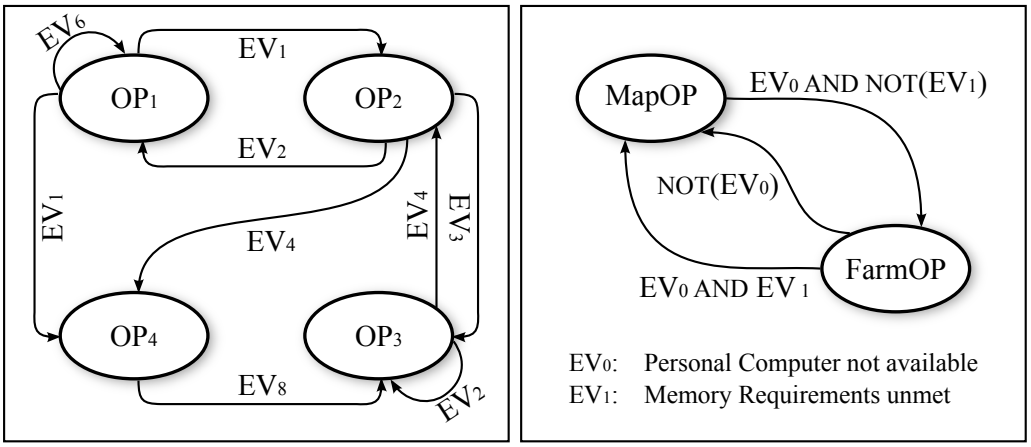
Mantenere la stessa struttura parallela e lo stesso codice utente in ASSISTANT significa (nel caso di topologia array) ridistribuire i Virtual Processor su un insieme differente di nodi. In questo caso ci basta ridistribuire lo stato sui nuovi nodi, ed il modulo riconfigurato potrà continuare l'esecuzione dal punto raggiunto con il vecchio comportamento.

Per la topologia *none* il discorso è differente: ogni Virtual Processor è indipendente dagli altri; qui, per ogni singolo VP possiamo applicare una delle due scelte elencate prima: mantenere il comportamento vecchio, oppure annullarli. Ma nel caso di riconfigurazione non funzionale il comportamento del singolo VP è rimasto invariato, perciò abbiamo un passaggio "indolore" da una versione all'altra.

Le tecniche utilizzate nel caso di riconfigurazione non funzionale sono comunque molto interessanti, e stiamo studiando come applicarle anche nel caso più generale, ovviamente non più in modo *automatico*, ma aiutati dallo sviluppatore, che può fornirci preziose informazioni per l'applicabilità di queste tecniche sulla singola applicazione.

5.4.4 Descrivere quando effettuare le riconfigurazioni

Fino a questo momento non abbiamo trattato una questione fondamentale del modello di ASSISTANT: come descrivere quando effettuare le riconfigurazioni. È ormai abbastanza chiaro che i differenti comportamenti del modulo sono forniti dal programmatore mediante la definizione di differenti *operation*, che riprendono la sintassi dei *parmod* ASSIST: tutti i listati visti fino a questo momento possono essere trasformati in operation senza particolari modifiche. Non abbiamo ancora parlato a fondo, invece, dei meccanismi per modellare il "manager", necessari per descrivere come e quando decidere di effettuare una riconfigurazione.



(a) Esempio di un grafo di riconfigurazione

(b) Grafo di riconfigurazione che modella il comportamento illustrato in figura 5.6

Figura 5.7: Event-Operation Graph

Parlando dello strato blue dell'applicazione abbiamo già spiegato che in ASSISTANT le riconfigurazioni sono scaturite dagli eventi. Le decisioni, però, non dipendono esclusivamente dall'evento ricevuto, ma anche dallo stato interno del modulo stesso: primo fra tutti, il comportamento attualmente in esecuzione. Questo ci porta alla definizione di una riconfigurazione in base alla ricezione di un evento e dell'attuale *operation* attiva. Possiamo quindi modellare l'insieme di possibili riconfigurazioni tramite un grafo, in cui i nodi rappresentano *operation* e gli archi *eventi*, esattamente come in figura 5.7a. Gli archi che tornano sulla stessa operation rappresentano le *riconfigurazioni non funzionali*, quelli tra operation diverse le *riconfigurazioni funzionali*. Gli eventi possono essere semplici eventi esterni oppure combinazioni di essi.

Dal grafo si può notare la dipendenza tra riconfigurazioni e operation corrente: infatti la ricezione dello stesso evento da due operation diverse può portare a riconfigurazioni differenti.

Inoltre dalla stessa operation possiamo avere più archi con lo stesso evento. Questo perché, come abbiamo già detto, una riconfigurazione è scaturita dalla ricezione di uno o più eventi, ma determinata anche dallo stato interno del modulo. In base a questo si potrebbe definire, su uno stesso evento, due possibili riconfigurazioni.

Per fare un esempio concreto, di un grafo di riconfigurazioni (chiamato in ASSISTANT **event-operation graph**) riportiamo il grafo (figura 5.7b) che rappresenta il comportamento del modulo descritto nella figura 5.6 per

presentare le riconfigurazioni funzionali con cambio di forma parallela. In questo caso abbiamo definito due eventi:

1. **EV_0**: Un evento ricevuto quando il computer multicore diventa non disponibile per il modulo, perché non più raggiungibile o perché allocato per altre applicazioni.

2. **EV_1**: Un evento ricevuto quando la piattaforma corrente non rispetta i requisiti di memoria imposti dalla versione farm (la quantità di ram di almeno uno dei dispositivi non è sufficiente).

Il grafo mostra anche l'utilizzo di combinazioni di eventi mediante il costrutto "AND", e del "NOT" per definire la non presenza di un particolare evento.

Nell'esempio, rispetto a quanto detto discusso nella sezione precedente, abbiamo inserito anche la logica per ritornare all'esecuzione sul quad-core nel caso questo torni disponibile. Inoltre le riconfigurazioni sono più precise: si passa dal Map al Farm se il Personal Computer non è disponibile e, nello stesso momento, i dispositivi utilizzati rispettano i requisiti di memoria. Al contrario si torna al Map se il PC non è ancora disponibile e i dispositivi non hanno abbastanza memoria oppure se il PC torna disponibile.

Questi due archi potrebbero essere in realtà collassati in un arco con evento EV_0 OR EV_1. Ma questo grafo viene fornito esplicitamente dal programmatore, che difficilmente si accorgerà di questa possibilità e verosimilmente li terrà divisi. Comunque al livello del modello o delle prestazioni questo non influisce minimamente.

5.4.5 Analogie con i meccanismi di adattività dei sistemi context-aware

Facendo una analogia con i sistemi visti nel capitolo 3 la prima tecnica riprende il concetto di Coda e Aura, in cui a livello applicativo non emerge l'adattività; al contrario, la seconda vede un approccio più simile ad Odyssey, dove il programmatore deve descrivere le implementazioni dei *Warden* per i vari livelli di fidelity.

Le nostre tecniche di adattività si discostano comunque da queste nominate, in quanto si definisce l'adattività in termini di qualità delle computazioni e non, come in Odyssey e Coda, in funzione dei dati. In ogni caso le due cose sono strettamente correlate: un algoritmo più veloce produce risultati (dati) con qualità (e quindi fidelity) minore, e al tempo stesso un dato con fidelity bassa avrà richiesto computazioni "leggere".

5.5 Context Awareness in ASSISTANT

Fino a questo momento lo sviluppo di ASSISTANT si è focalizzato principalmente sulle tecniche per l'adattività, mentre l'aspetto della context-awareness è stato un po' trascurato e rimandato a studi futuri.

Nonostante questo siamo fermamente convinti che esista una forte analogia tra adattività e context-awareness, e che le *riconfigurazioni funzionali* possano permettere di esprimere *anche* comportamenti context-aware per i moduli.

Dal nostro punto di vista la struttura base per realizzare applicazioni context-aware è già presente in ASSISTANT. Ovviamente va studiata meglio, al fine di verificare le nostre ipotesi ed eventualmente fornire costrutti appositi (ma si tratterebbe comunque di "zucchero sintattico") per facilitare la descrizione di tali comportamenti.

Un aspetto di sicuro interesse è l'elaborazione e l'analisi dei dati nell'ottica di dedurre nuova conoscenza. Molto probabilmente anche in ASSISTANT sarà necessario introdurre tecniche di inferenza, analoghe a quelle presenti in tutti i modelli context-aware, per generare nuove informazioni a partire da quelle acquisite dal contesto. Nel nostro caso la conoscenza dei singoli moduli è limitata ad una parte dell'ambiente, quello di cui ricevono gli eventi, esattamente come in CORTEX.

In un primo momento abbiamo pensato di definire un sistema di inferenza della conoscenza da inserire in ogni modulo. In questo modo ogni singola componente dell'applicazione, a partire dagli eventi che riceve dal contesto, potrebbe dedurre nuovi fatti (e quindi generare nuovi eventi). Questa nuova conoscenza sarebbe comunque rilegata al singolo modulo, dotato di un processo di inferenza indipendente dagli altri. La prima modellazione, proposta in [58] e riportata in figura 5.8 vede nel manager una parte dedicata alla gestione dei dati di contesto, molto simile a quella di CORTEX: i dati vengono acquisiti dal contesto, modellati tramite ontologie ed elaborati con tecniche di inferenza, allo scopo di dedurre nuove informazioni. Questa parte però, pur essendo considerata nella struttura del Manager, non è stata mai modellata nei programmi ASSISTANT.

Ultimamente stiamo pensando di modificare questa struttura, lasciando ai manager il solo compito di decidere le riconfigurazioni e guidare lo strato RED in queste operazioni.

La necessità di modellare i dati provenienti dalle interfacce di contesto sia come dati su stream che come eventi, ci ha portato ad una considerazione importante riguardo all'essenza di questi ultimi. Un "evento"

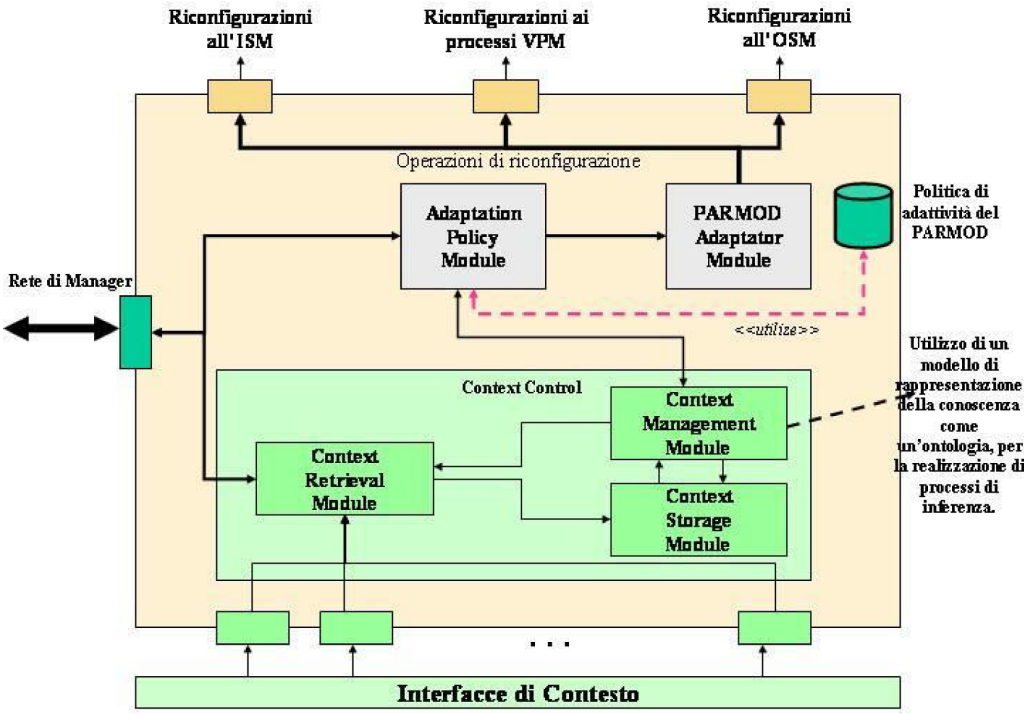


Figura 5.8: La prima versione di Manager, con componenti per la gestione del contesto e un motore di inferenza

viene rappresentato da un valore, inviato al verificarsi della condizione che rappresenta.

Possiamo perciò trovare una certa analogia tra *connessioni per eventi* e *stream*: entrambe modellano una comunicazione tra due entità, in cui il dato scambiato è sempre dello stesso tipo.

Perciò la nostra proposta è di modellare anche gli eventi come dati inviati su uno stream; in questo modo possiamo trattare in uniformemente i due concetti: il contenuto di uno stream può essere sia un dato per l'applicazione che un evento, in base a come viene trattato dal parmod.

In questo modo possiamo modellare in modo nativo quei casi in cui i dati prodotti da interfacce di contesto non rappresentano eventi, ma valori utilizzati come dati di input per l'applicazione. Analogamente un modulo può generare dati che vengono trattati come "eventi" dai parmod che li ricevono.

Le potenzialità di questo approccio sono molteplici; vediamone alcune:

- Possiamo modellare in modo nativo e trasparente la produzione di eventi e di dati dalle interfacce di contesto.
- Analogamente anche lo scambio di informazioni tra manager viene gestito in modo uniforme alle restanti informazioni.
- Lo sviluppatore ha la possibilità di creare eventi "derivati" utilizzando un parmod programmato a dovere che, ricevendo eventi dalle interfacce di contesto, ne genera di più complessi; in questo modo possiamo creare sistemi di inferenza come quelli utilizzati dai modelli context-aware classici, ma anche algoritmi più complessi ed eventualmente paralleli, grazie alla potenza del parmod.

La generazione di eventi complessi, derivati da inferenze, verrebbe così spostata nello strato RED e trattata semplicemente come una parte computazionale dell'applicazione, al pari dei restanti moduli ASSISTANT.

Ovviamente in futuro possiamo studiare modalità per fornire dei costrutti al livello di linguaggio per definire queste entità, ad esempio utilizzando linguaggi logici stile CLIPS, Prolog, etc. Potremmo inserire queste capacità all'interno del parmod, oppure offrire una nuova tipologia di modulo (un po' come il modulo sequenziale in ASSIST) che espone i costrutti per utilizzare un motore di inferenza fornito nativamente da ASSISTANT.

5.6 Tolleranza ai guasti e alle disconnessioni

Uno degli aspetti che consideriamo fondamentali in questo tipo di applicazioni è la tolleranza a guasti e disconnessioni, che in un ambiente di esecuzione

normale sono da considerarsi eccezioni (ed essere trattate come tali), mentre in ambienti pervasivi diventano situazioni comuni. Il nostro proposito è di utilizzare il lavoro svolto in [20]; questo ci permette di inserire tecniche efficienti di tolleranza a guasti, attraverso l'uso di meccanismi di checkpointing e rollback-recovery, in applicazioni composte da grafi di forme di parallelismo note: esattamente la struttura di applicazione ASSISTANT.

Per quanto riguarda i manager, invece, questi richiedono un trattamento particolare, in quanto esterni alla struttura parallela dell'applicazione; a differenza delle altre parti di un modulo, questi devono essere sempre presenti durante l'intera computazione, anche in caso di fallimenti. Proponiamo perciò tecniche di replicazione, probabilmente di tipo *Active*, dove ogni copia esegue esattamente le stesse operazioni.

Le tecniche di tolleranza ai guasti tramite checkpointing si possono anche applicare per realizzare in modo efficiente le modalità di riconfigurazione di cui abbiamo parlato nella sezione sull'adattività. Infatti il primo requisito per le tecniche di checkpointing su applicazioni parallele è quello di trovare dei punti durante l'esecuzione in cui lo stato dell'intera applicazione (vista come insieme di processi) sia "consistente".

Questo requisito è fondamentale *anche* per la modifica del comportamento dell'applicazione. I *reconfiguration point* di ASSIST, utilizzati per modificare il grado di parallelismo dei moduli paralleli, sono esattamente punti in cui lo stato dell'intero modulo è consistente. Il checkpoint stesso può essere utilizzato *anche* durante una riconfigurazione, per trasferire sui processi appena avviati la parte di stato interno di cui hanno bisogno.

L'utilizzo di tecniche di checkpointing si rivela quindi una ottima soluzione per risolvere contemporaneamente i problemi di adattività e di fault tolerance.

5.7 Conclusioni

In questo capitolo abbiamo proposto una prima introduzione alle caratteristiche di ASSISTANT. Abbiamo preso spunto dai lavori presentati nel capitolo 3 per estendere il modello di ASSIST, in modo da poter realizzare anche applicazioni adattive e context-aware, capaci quindi di essere eseguite in ambienti pervasivi ed in *pervasive grid*.

Abbiamo discusso una la modellazione a tre strati, che rappresenta tre differenti aspetti di una applicazione ASSISTANT e ci guida nella sua definizione.

Abbiamo mostrato la tecnica da noi proposta per gestire in modo coerente sia l'adattività che la context awareness, tramite la definizione di più versioni di una stessa entità (nel nostro caso il modulo) dell'applicazione, con un approccio che riprende le idee dei modelli come Odyssey e Aura, per applicarle ad una strutturazione a moduli derivata da ASSIST.

A questo punto pensiamo di aver posto le basi per presentare in modo definitivo la sintassi di ASSISTANT, applicata ad un esempio più complesso di applicazione adattiva e context aware che vedremo nel prossimo capitolo.

Capitolo 6

Un esempio completo

Presentiamo ora un esempio completo di applicazione ASSISTANT, studiato nell'ambito del progetto In.Sy.Eme come test-bed per una applicazione di gestione delle emergenze.

Questo esempio è stato trattato, con diverse sfaccettature, in [21, 22]. Lo riprendiamo, per dare al lettore un'idea più chiara della reale utilità delle riconfigurazioni considerate in ASSISTANT, e dello studio richiesto per lo sviluppo di una applicazione pervasiva per tale modello.

In una applicazione di gestione di inondazioni fluviali la parte di previsione (eseguita costantemente e non solo in situazioni critiche) utilizza dei modelli di idrodinamica per prevedere il livello dei fiumi e, nel caso di straripamenti, le probabili zone interessate dagli allagamenti. Questi modelli fanno largo uso di sistemi lineari per risolvere equazioni differenziali parziali[72]. Le matrici dei sistemi da risolvere normalmente non sono generali, ma piuttosto sparse e con caratteristiche peculiari che permettono una risoluzione efficiente: ci troviamo spesso a lavorare con sistemi lineari tridiagonali.

Esistono diversi algoritmi per la risoluzione di sistemi lineari tridiagonali, ognuno con caratteristiche differenti. In questa ottica, uno studio dei vari algoritmi porterebbe già alla definizione di più operation, con algoritmi diversi, da scegliere in base all'ambiente di esecuzione. Nel nostro caso ci siamo focalizzati su un unico algoritmo di tipo diretto. Tra questi, alcuni esempi importanti sono la *twisted factorization* e la *cyclic reduction*[40]. Abbiamo scelto il secondo perché facilmente generalizzabile a sistemi "a banda" o "tridiagonali a blocchi".

Successivamente nel capitolo presenteremo l'algoritmo base nella sezione 6.1, ed una versione specifica studiata per una migliore parallelizzazione (sezione 6.2). Di queste forniremo, come al solito, una implementazione in ASSIST; passiamo poi a descrivere l'ambiente e le infrastrutture su cui l'applicazione verrà eseguita nella sezione 6.3. Continuiamo presentando in modo

completo l'esempio di applicazione studiato dal gruppo (sezione 6.4), le considerazioni su quale versioni dei moduli eseguire sulle varie piattaforme (6.5) e, per finire, le riconfigurazioni proposte per il modulo principale nella sezione 6.6.

6.1 Cyclic Reduction

Una matrice tridiagonale è della forma 6.1, che permette una rappresentazione compatta dell'intero sistema utilizzando quattro elementi per ogni riga, come riportato in 6.2, con $a_1 = c_n = 0$ e i k_i che rappresentano il vettore dei termini noti.

$$\mathcal{A} = \begin{pmatrix} b_1 & c_1 & 0 & \cdots & \cdots & \cdots & 0 \\ a_2 & b_2 & c_2 & \ddots & & & \vdots \\ 0 & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & & \ddots & \ddots & \ddots & c_{n-1} \\ 0 & \cdots & \cdots & \cdots & 0 & a_n & b_n \end{pmatrix} \tag{6.1}$$

$$\mathcal{S} = \begin{pmatrix} a_1 & b_1 & c_1 & k_1 \\ \vdots & \vdots & \vdots & \vdots \\ a_n & b_n & c_n & k_n \end{pmatrix} \tag{6.2}$$

L'algoritmo di *cyclic reduction* originale, ideato da Golub e Hockney, nasce per la risoluzione dell'equazione di Poisson tramite il calcolo di un insieme di sistemi tridiagonali [54]. L'algoritmo prevede di avere un numero di righe nella forma $2^q - 1$, ma studi successivi[71] ne hanno dimostrato l'applicabilità anche per matrici più generali. Nel nostro caso, per semplicità, riportiamo l'algoritmo originale. Per una trattazione più approfondita della cyclic reduction consigliamo [53].

L'idea base dell'algoritmo è di applicare una trasformazione per eliminare le incognite con indice dispari. In questo modo ci troviamo con un sistema di dimensione dimezzata. Applicando la trasformazione iterativamente, si riesce ad ottenere un sistema con una singola incognita dalla soluzione ovvia. A questo punto, mediante una procedura di riempimento, si ottengono, sempre in modo iterativo, i valori per tutte le altre incognite.

L'algoritmo è quindi naturalmente suddiviso in due parti: *trasformazione* e *risoluzione*.

Fase di Trasformazione

Questa fase si suddivide in $q-1$ passi con $q = \log_2(N+1)$, e ad ogni passo trasforma un numero sempre minore di righe. Le righe interessate dalla trasformazione al passo l sono quelle con indice divisibile da 2^l, ovvero le i tali che $i \bmod 2^l = 0$.

Ad ogni passo la trasformazione necessita dei dati del precedente, e modifica i valori come riportato in 6.3.

$$\begin{array}{ll} a_i^l = \alpha_i a_{i-2^{l-1}}^{l-1} & c_i^l = \gamma_i c_{i+2^{l-1}}^{l-1} \\ b_i^l = b_i^{l-1} + \alpha_i c_{i-2^{l-1}}^{l-1} + \gamma_i & k_i^l = k_i^{l-1} + \alpha_i k_{i-2^{l-1}}^{l-1} + \gamma_i k_{i+2^{l-1}}^{l-1} \\ & \textit{dove} \\ \alpha_i = -\dfrac{a_i^{l-1}}{b_{i-2^{l-1}}^{l-1}} & \gamma_i = -\dfrac{c_i^{l-1}}{b_{i+2^{l-1}}^{l-1}} \end{array} \tag{6.3}$$

Fase di Risoluzione

Questa fase, che produce i valori del vettore X delle soluzioni, è suddivisa in q passi (uno in più della precedente), ed utilizza i dati prodotti da tutti gli step di trasformazione per calcolare la soluzione.

Ad ogni passo $l = q, q-1, ..., 1$ vengono calcolate le soluzioni per le righe i tali che $i \bmod 2^{l-1} = 0$, se non già calcolate precedentemente, secondo la formula in 6.4.

$$\begin{aligned} x_i &= \frac{k_i^{l-1} - a_i^{l-1} x_{i-2^{l-1}} - c^{l-1} x_{i+2^{l-1}}}{b_i^{l-1}} \\ x_i &= 0, \text{ per } i < 0 \text{ e } i > N \end{aligned} \tag{6.4}$$

Parallelizzazione dell'algoritmo

I passi dell'algoritmo sono illustrati graficamente in figura 6.1.

La struttura di questo algoritmo suggerisce una parallelizzazione di tipo *data-parallel con stencil*, ma in un'ottica parallela i processori rimarrebbero inutilizzati per gran parte del tempo durante entrambe le fasi (ad ogni passo si lavora con la metà delle righe del passo precedente). Questa versione, infatti, è stata pensata per minimizzare il numero di operazioni nell'ottica di una esecuzione sequenziale. Dal punto di vista prestazionale questo algoritmo si comporta quindi molto meglio con parallelizzazioni di tipo *farm*.

Per questo motivo presentiamo, nel listato 6.1, la sola versione farm. Durante i test abbiamo anche sviluppato una versione data-parallel, che si

```
typedef struct { double a,b,c,k; } row;

parmod TridiagonalSolver(input_stream row System[N]
    output_stream double Solution[N]){

  topology none Pv;
  do input_section{
    guard1: on , , System {
      distribution System on_demand to Pv;
    }
  }while(true)
  virtual_processors {
    elaborazione( in guard1 out Solution){
      VP {
        FCyclicReduction(in A out Solution);
      }
    }
  }
  output_section{
      collects Solution from ANY Pv;
  }
}
```

Listato 6.1: Modulo Cyclic Reduction parallelizzata tramite farm

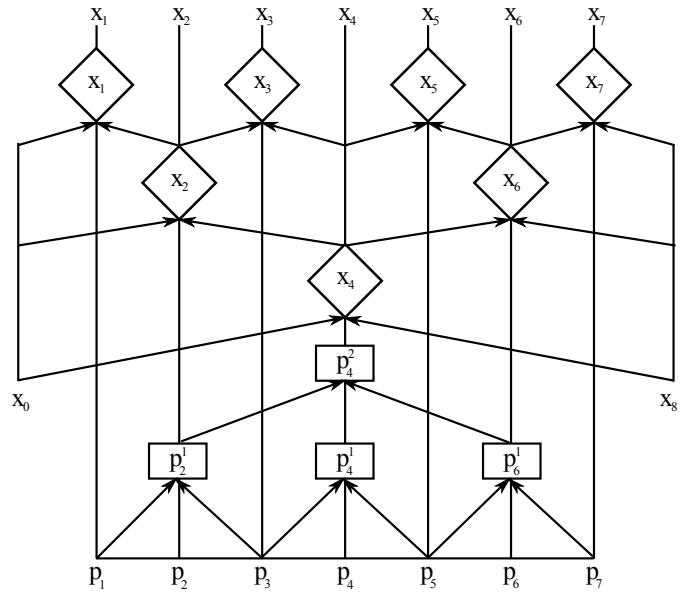


Figura 6.1: Rappresentazione grafica dei passi dell'algoritmo Cyclic Reduction e delle dipendenze tra i dati (stencil).

è però rivelata molto meno performante dei quella farm; evitiamo perciò di riportarla.

6.2 Parallel Cyclic Reduction

Il calcolo su un numero sempre più basso di righe della Cyclic Reduction rende l'algoritmo poco adatto ad una parallelizzazione di tipo data-parallel; per questo motivo è stata studiata una variante, chiamata *Parallel Cyclic Reduction*, più adatta a questa forma di parallelismo, che cerca di mitigare il problema.

In questa versione si cerca di superare i problemi della Cyclic Reduction vista prima aumentando il numero di operazioni nella fase di trasformazione, per sfruttare al meglio i processori; ovviamente il lavoro aggiuntivo non è completamente perso, e permette di ridurre significativamente la fase di risoluzione, eseguita in un solo passo e senza dipendenze funzionali tra le righe. Vediamo meglio l'algoritmo:

Fase di Trasformazione

In questo caso si applicano ancora le formule in 6.3, ma per tutte le righe, ad ogni passo, e questa volta per q passi. L'attivazione su tutte le righe porta la dipendenza funzionale su $i \pm 2^{l-1}$ alla generazione di indici di riga negativi o non esistenti. Per tutti questi casi utilizziamo una riga fittizia, come riportato in 6.5

$$\begin{pmatrix} a_i & b_i & c_i & k_i \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 0 \end{pmatrix} \quad i < 0 \text{ e } i > N \tag{6.5}$$

In questo modo tutti i nodi utilizzati eseguono la stessa quantità di lavoro, mantenendo quindi il carico tra essi bilanciato. Purtroppo aumenta notevolmente anche la quantità di dipendenze logiche e quindi di dati trasferiti, in quanto il calcolo su ogni riga ne richiede due del passo precedente.

Fase di Risoluzione

La caratteristica importante di questa versione di cyclic reduction è che ci permette di calcolare le soluzioni senza una procedura iterativa di riempimento.

Infatti l'equazione 6.4 rimane valida per calcolare le soluzioni, ma l'applicazione di un ulteriore passo di trasformazione ci ha portato ad avere tutte le dipendenze della forma $x_{i-2^{l-1}}$ ad una riga $i < 0$ e quelle $x_{i+2^{l-1}}$ ad una

riga $i > N$. Perciò i due termini additivi della 6.4 si annullano e la soluzione può essere calcolata semplicemente con la 6.6.

$$x_i = \frac{k_i^{q-1}}{b_i^{q-1}} \tag{6.6}$$

Questo ci porta ad una fase di risoluzione senza dipendenze logiche ed eseguibile, in parallelo, su tutte le righe in un solo passo.

Parallelizzazione dell'algoritmo

I passi di questa versione dell'algoritmo sono illustrati graficamente in figura 6.2. Questa versione presenta molte caratteristiche interessanti nell'ottica di una versione data-parallel, ma anche per quella sequenziale.

Innanzitutto questa versione sfrutta in modo migliore i processori, in quanto tutti lavorano ad ogni passo, ed in questo modo riusciamo a diminuire i passi totali. Purtroppo, come possiamo vedere anche nella figura 6.2, la quantità di dati scambiati aumenta considerevolmente; in un ambiente parallelo a scambio di messaggi questa caratteristica potrebbe limitare in modo consistente le prestazioni dell'algoritmo, in quanto non possiamo sovrapporre il calcolo alle comunicazioni. In una architettura a memoria condivisa, invece, possiamo sperare in un comportamento migliore, in quanto non abbiamo bisogno di copiare i dati delle dipendenze logiche ma possiamo accedervi direttamente.

La Parallel Cyclic Reduction presenta anche caratteristiche interessanti per quanto riguarda l'occupazione di memoria. Infatti, la versione originale richiede, nella fase di risoluzione, i valori di tutti gli step di trasformazione; su matrici grandi la quantità di memoria richiesta potrebbe non essere non trascurabile, soprattutto per esecuzioni interamente sequenziali, in cui allochiamo tutto nella memoria del singolo nodo. In quest'ottica questo secondo algoritmo può essere ottimizzato, durante la fase di trasformazione, per mantenere solo i dati a partire dal penultimo ultimo step non ancora completamente terminato. Nella versione sequenziale questo si traduce nel dover mantenere solo due copie della matrice: una per lo step precedente, una per quello attuale.

Per questo motivo questo algoritmo, pur richiedendo più tempo a causa dei maggiori calcoli eseguiti, può essere utilizzato anche per versioni sequenziali pure o in farm, per i dispositivi con poca memoria come PDA e SmartPhone.

Riportiamo perciò due differenti versioni di questo algoritmo, parallelizzato tramite farm (listato 6.2) e tramite data-parallel (listato 6.3).

```
typedef struct { double a,b,c,k; } row;

parmod TridiagonalSolver(input_stream row System[N]
    output_stream double Solution[N]){

  topology none Pv;
  input_section{
    guard1: on , , System {
      distribution System on_demand to Pv;
    }
  }while(true)
  virtual_processors {
    elaborazione( in guard1 out Solution){
      VP {
        FParaCyclicReduction(in A out Solution);
      }
    }
  }
  output_section{
      collects Solution from ANY Pv;
  }
}
```

Listato 6.2: Modulo per Parallel Cyclic Reduction parallelizzata tramite farm. L'unica differenza rispetto alla 6.1 è la proc invocata

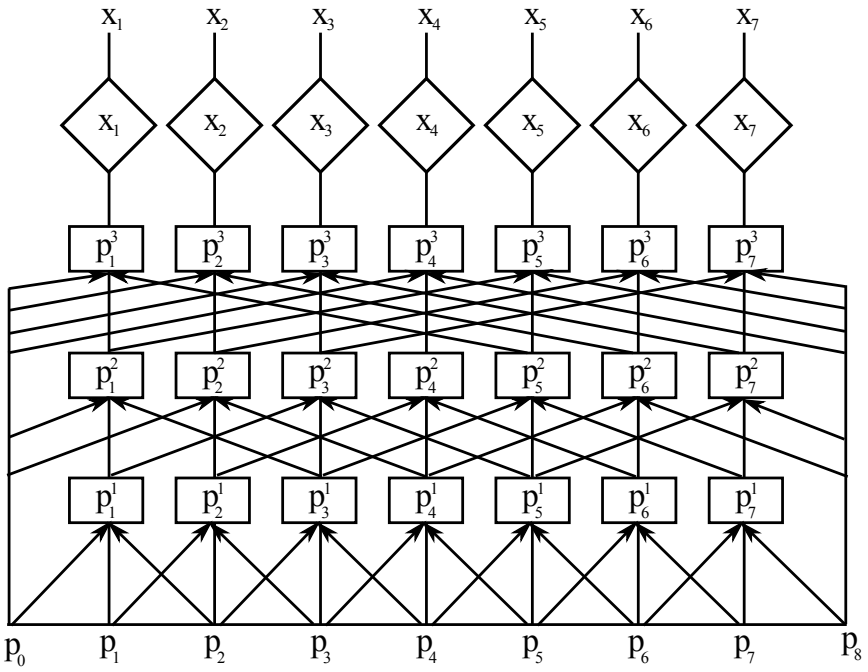


Figura 6.2: Rappresentazione grafica dei passi dell'algoritmo Parallel Cyclic Reduction e delle dipendenze tra i dati (stencil).

```
typedef struct { double a,b,c,k; } row;

parmod TridiagonalSolver(input_stream row System[N]
    output_stream double Solution[N]){
  topology array [i:N] Pv;

  attribute row M[N][LOG_2_N];
  attribute row empty;
  stream long out_matrix;
  init{ empty.a=0; empty.b=1; empty.c=0; emprty.k=0; }
  do input_section{
    guard1: on , , System {
      distribution System[*i0] scatter to M[i0][0];
    }
  }while(true)
  virtual_processors {
    elaborazione( in guard1 out out_matrix){
      VP i=1..N {
        for(l=0;l<LOG_2_N-1;l++){ //Transformation
          left = i - pow(2,l-1);
          right = i + pow(2,l-1);
          if(left < 0 && right < N)
            FTrasform(in M[i][l-1],empty,M[right][l-1] out
                M[i][l]);
          else if(sx >= 0 && dx => N)
            FTrasform(in Mat[i][l-1],Mat[left][l-1],empty out
                Mat[i][l]);
          else if(left < 0 && dx => N)
            FTrasform(in Mat[i][l-1],empty,empty out Mat[i][l]);
          else
            FTrasform(in Mat[i][l-1],Mat[left][l-1],
                Mat[right][l-1] out Mat[i][l]);
        }
        //Solving
        FSolve(in Mat[i][LOG_2_N-1] out out_matrix);
    }
  }
  output_section{
    collects out_matrix from ALL Pv[i] {
      double elem; double ris[N];
      AST_FOR_EACH(elem) {ris[i]=elem;}
      assist_out(result, ris);
    }<>;
  }
}
```

Listato 6.3: Modulo per Parallel Cyclic Reduction parallelizzata tramite data-parallel con stencil

La versione farm è identica a quella vista precedentemente, se non per la proc invocata. Il data-parallel invece merita qualche cenno ulteriore. Un lettore attento avrà notato che lo stencil della computazione è di tipo *variabile*: è definito staticamente, ma cambia ad ogni passo. Purtroppo, come possiamo vedere nel codice, la presenza di casi speciali (quelli che utilizzano la riga vuota vista in 6.5) e di uno stencil calcolato in modo complesso ($i \pm 2^{l-1}$) richiedono un lavoro notevole da parte del compilatore per riconoscere lo stencil come *variabile* e non *dinamico*. Questo è importante, in quanto nel primo caso il compilatore potrebbe conoscere l'entità delle comunicazioni ad ogni passo, ottenere un modello dei costi e ragionare su differenti ottimizzazioni che, nel secondo caso, non potrebbe fare.

6.3 La piattaforma per la gestione delle emergenze

Vediamo ora nel dettaglio l'insieme di risorse su cui pensiamo di sviluppare l'applicazione di gestione delle emergenze per il progetto InSyEme. L'applicazione studiata tratta il problema, molto diffuso in Italia, di previsione e gestione delle inondazioni. L'estensione geografica delle aree da monitorare anche per un singolo fiume è significativa, e richiede una infrastruttura informatica che segua tutto il letto del fiume per ottenere misurazioni fisiche del livello di esso in vari punti.

La necessità di utilizzare algoritmi di previsione molto pesanti rende necessaria la presenza di un centro di elaborazione molto potente. Difficilmente questo sarà situato direttamente nella zona dell'emergenza, a causa sia della sua estensione fisica sia della necessità di mantenere questo centro lontano da zone di pericolo.

I sensori devono essere collegati al centro di calcolo mediante una rete ad alta velocità, per inviare tutti i dati raccolti al fine di effettuare successive elaborazioni. La rete verrà anche utilizzata, nel caso di una emergenza, per inviare i risultati delle computazioni ai dispositivi mobili presenti nell'area interessata. L'eccessiva estensione dell'area da monitorare rende impraticabile l'utilizzo di una unica connessione tra sensori e centro di calcolo, e suggerisce un partizionamento dell'area da monitorare in piccole sottoaree, ognuna coordinata da un nodo speciale collegato tramite una connessione dedicata ad alta velocità col centro.

Proponiamo quindi una infrastruttura organizzata gerarchicamente, come quella riportata in figura 6.3, con le seguenti caratteristiche:

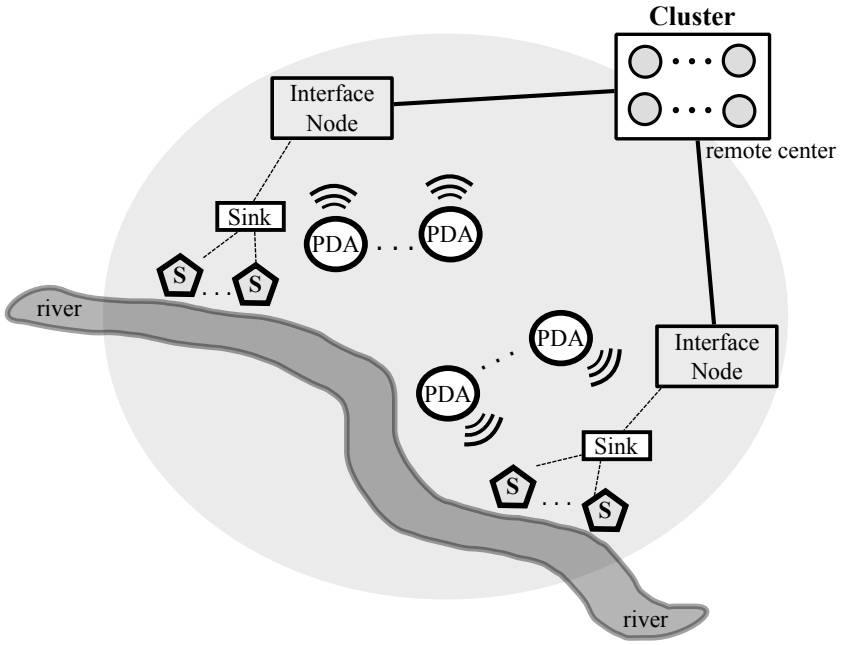


Figura 6.3: La struttura computazionale immaginata nell'ambito del progetto InSyEme

- Abbiamo una o più reti di sensori, che acquisiscono i dati necessari per eseguire previsioni e/o computazioni pesanti.
- I calcoli di previsione vengono normalmente eseguiti su un cluster di macchine ad alte prestazioni, situato in un centro di calcolo distante dal luogo di emergenza.
- Durante le emergenze possiamo assumere di avere un gruppo abbastanza numeroso di PDA nella zona di interesse, utilizzati dalle forze che gestiscono l'emergenza.
- I sensori sono organizzati in reti autonome, e controllati da un nodo specifico chiamato *Sink* che li interfaccia col resto del sistema.
- Le reti di sensori ed i PDA sono collegati al cluster per mezzo di nodi dedicati (che chiameremo *nodi di interfaccia*), collocati staticamente in zone ben precise del territorio; questi nodi sono singole macchine ma abbastanza potenti, dotate di processori multicore e connessioni dedicate ad alta velocità verso il centro di calcolo.

Questa architettura gerarchica permette l'esecuzione delle applicazioni anche in casi critici: in assenza della connessione verso il cluster possiamo

contare sulle potenzialità del nodo di interfaccia per eseguire l'applicazione in modalità ridotta, ed utilizzare i dispositivi mobili presenti nel caso anche il nodo di interfaccia sia occupato oppure non disponibile.

Inoltre dobbiamo considerare anche l'aspetto "context-aware" della parte di applicazione che verrà eseguita sui PDA della protezione civile: ci troviamo quindi in un ambiente pervasivo a tutti gli effetti, e abbiamo bisogno di strumenti adatti per sfruttarlo correttamente.

6.4 Una applicazione per la previsione delle inondazioni

In questa sezione presentiamo lo schema generale di una classica applicazione previsionale, e l'esempio da noi realizzato per emulare il comportamento di una piccola parte di essa.

Una applicazione di previsione di inondazioni è suddivisa in più parti, riportate in figura 6.4.

- **Geographic Information System (GIS)** Questo modulo si occupa di immagazzinare tutti i dati relativi all'ambiente naturale su cui eseguire la previsione; questi dati possono essere determinati staticamente (come la conformazione del territorio, il letto del fiume, la presenza di strade, etc) oppure dinamicamente, come i dati raccolti dai sensori (che modellano un'informazione variabile nel tempo). Come illustrato in figura, queste informazioni sono necessarie ad ogni singola parte dell'applicazione.

- **Meteorological Prediction Model** Necessario per avere delle stime sulle precipitazioni future, utilizzate nella parte previsionale.

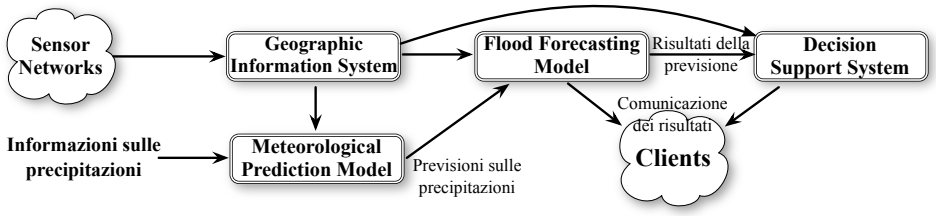


Figura 6.4: La struttura di una applicazione di previsione e gestione delle inondazioni

- **Flood Forecasting Model** Il vero modello previsionale. Riceve i dati dal GIS e dal modello meteorologico, li elabora e fornisce delle previsioni sul futuro livello del fiume e su possibili inondazioni.

- **Decision Support System** Utilizzando i dati del modello previsionale elabora le strategie per gestire al meglio le emergenze.

Una piccola ma significativa parte dell'applicazione

Vista la dimensione di questa applicazione abbiamo cercato di isolarne una parte significativa, che ci permettesse da un lato di tener conto di aspetti di adattività e context-awareness, dall'altro di ottenere risultati riportabili all'intera applicazione. In questa ottica ci siamo focalizzati sulla parte previsionale.

In commercio si trovano pacchetti software già pronti che realizzano tutte le parti di un sistema previsionale; tra questi segnaliamo i modelli Mike 11 e Mike 21[73] e TUFLOW[72]. Quest'ultimo si basa sul calcolo di equazioni alle derivate parziali per massa e momento tramite la risoluzione di sistemi lineari. Preso un insieme di punti geografici, per ognuno di esso richiede la risoluzione di quattro sistemi tridiagonali. Possiamo modificare la qualità della previsione aumentando il numero di punti campionati oppure la dimensione dei sistemi lineari utilizzati.

Lo studio precedente sulla Cyclic Reduction si pone perciò nell'ottica di studiare uno dei meccanismi di base del modello previsionale. Non essendo interessati allo studio di tutti i problemi legati all'applicazione, approssimiamo quest'ultima con un modulo che si occupa di risolvere i quattro sistemi tridiagonali a partire dall'insieme di dati necessario per generarli.

La nostra applicazione diventa quindi quella in figura 6.5, composta da tre moduli:

- **TridiagonalSolver**: rappresenta il modello previsionale: riceve i dati di un punto geografico e da questi genera e risolve i quattro sistemi lineari tridiagonali presentati in TUFLOWS; il risultato di questa simulazione viene inviata al modulo visualizzatore.

- **Generator**: genera i dati da inviare al "modello previsionale". Questi dati dovrebbero essere generati a partire dai campionamenti del sensore, ma attualmente questa parte del sistema è emulata.

- **Viewer**: visualizza i dati ricevuti dal "modello previsionale"; non ci effettua ulteriori elaborazioni, ma li stampa a video.

Figura 6.5: L'esempio di applicazione ASSISTANT nell'ottica della gestione delle emergenze

I moduli ASSIST presentati precedentemente per la Cyclic Reduction variano quindi leggermente: non ricevono più il sistema da risolvere, ma un insieme di valori: posizione del punto (x,y) a cui applicare il modello, profondità del fiume in quel punto, altezza dalla superficie e velocità media del flusso (u,v). In base a questi 6 valori siamo in grado di generare i quattro sistemi lineari, risolverli ed ottenere due vettori di flussi: Fx e Fy.

Per non appesantire ulteriormente la trattazione non riportiamo la nuova versione dei moduli, ma sottolineiamo una nuova caratteristica molto importante. Questa modellazione, infatti, ci permette di applicare una nuova, importante forma di adattività: ora il modulo può modificare la dimensione dei sistemi risolti, mentre prima, a causa delle interfacce verso l'esterno non modificabili, la dimensione della matrice di input e del vettore di output erano fissate.

Vista l'importanza di questa forma di adattività pensiamo comunque di studiare questo problema in generale: vogliamo poter definire tipi di dato di dimensione variabile nell'ottica di modificarne il grado di accuratezza.

6.5 La Cyclic Reduction su differenti piattaforme

Una volta definita l'applicazione, abbiamo iniziato a scrivere dei prototipi di possibili applicazioni ASSISTANT per le varie piattaforme. Vista la mancanza di un modello definitivo e, soprattutto, di una sua implementazione, abbiamo realizzato i prototipi nell'ottica di studiare gli strumenti a basso livello da utilizzare nell'implementazione di ASSISTANT. Per alcune versioni abbiamo anche scritto i moduli in ASSIST, ma i test che riportiamo riguardano implementazioni con strumenti a più basso livello (socket, MPI, librerie di comunicazione specifiche, etc).

Nell'ottica del lavoro su InSyEme abbiamo realizzato versioni del modulo adatte ad essere utilizzate su tutte le piattaforme di esecuzione possibili:

cluster, *nodo di interfaccia*, *PDA*. Vediamo in dettaglio il lavoro svolto ed i risultati ottenuti, suddivisi per ambiente di esecuzione.

6.5.1 Versione per Cluster

Abbiamo realizzato più versioni della Cyclic Reduction per cluster, utilizzando la libreria di comunicazione MPI. Sono state implementate la forme parallele farm e data-parallel. La piattaforma di esecuzione è il cluster del gruppo di ricerca, *Pianosa*.

Pianosa è composto da 32 nodi Pentium III 800Mhz con 512 KB di cache ed 1 GB di memoria principale. I nodi sono connessi tramite una rete Fast Ethernet a 100 Mbit/s. Pur non essendo recente questo cluster ci permette di studiare la scalabilità di una applicazione su un discreto numero di nodi; i risultati di questa versione vanno quindi analizzati in questa ottica: non con l'idea di ottenere i tempi di servizio di un cluster moderno, ma di conoscere quali implementazioni della cyclic reduction potranno essere verosimilmente implementate su un cluster.

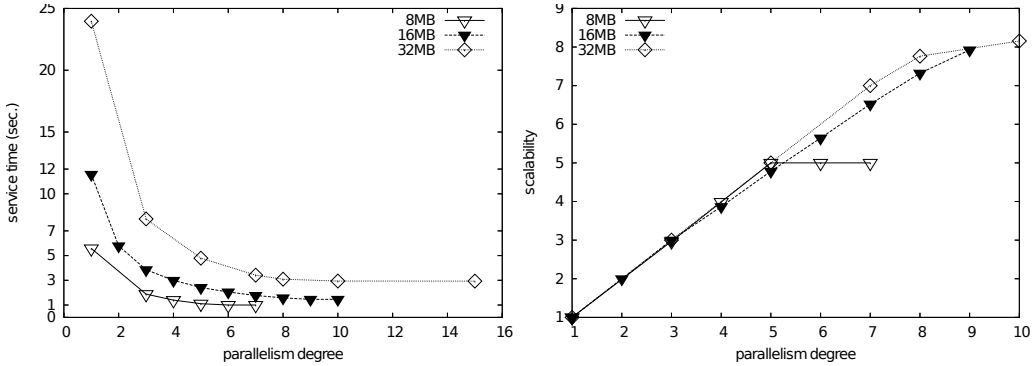
In questo scenario la versione data-parallel presenta grossi problemi, sia di scalabilità che di prestazioni assolute, a causa del traffico eccessivo tra i nodi durante la singola computazione. Queste comunicazioni non possono essere sovrapposte al calcolo, e quindi rappresentano tempo "perso", in cui i nodi rimangono fermi, in attesa del dato.

Per questo motivo su un cluster la versione farm si comporta molto meglio, e rimane in assoluto la scelta preferibile. I risultati prestazionali di questa versione sono riportati nei grafici in figura 6.6, che riportano scalabilità e tempi di servizio al variare del grado di parallelismo. Abbiamo eseguito i test con tre differenti dimensioni di matrice: 2^{18}, 2^{19} e 2^{20} righe, che occupano rispettivamente 8, 16 e 32 MB. Questo per studiare l'andamento delle prestazioni al variare della qualità richiesta, per poter definire diverse politiche adattive.

I risultati mostrano una scalabilità molto buona del farm, in linea con quella ideale, fino a quando non si raggiunge un tempo di servizio più basso del tempo per l'invio di un sistema sulla rete; questi risultati sono tutto sommato soddisfacenti, perché in linea col modello dei costi di questa forma parallela.

6.5.2 Versione per Nodo di Interfaccia

Per il nodo di interfaccia abbiamo studiato due differenti piattaforme, entrambe di tipo multicore, su singolo nodo. Queste piattaforme sono molto più moderne rispetto al cluster: questa volta, oltre a studiare la scalabilità



(a) Tempo di servizio (b) Scalabilità

Figura 6.6: Prestazioni del farm di Cyclic Reduction sul cluster Pianosa

teorica della Cyclic Reduction su questa tipologia di nodi, possiamo anche ottenere delle stime molto valide sui limiti delle attuali tecnologie utilizzabili sui nodi di interfaccia.

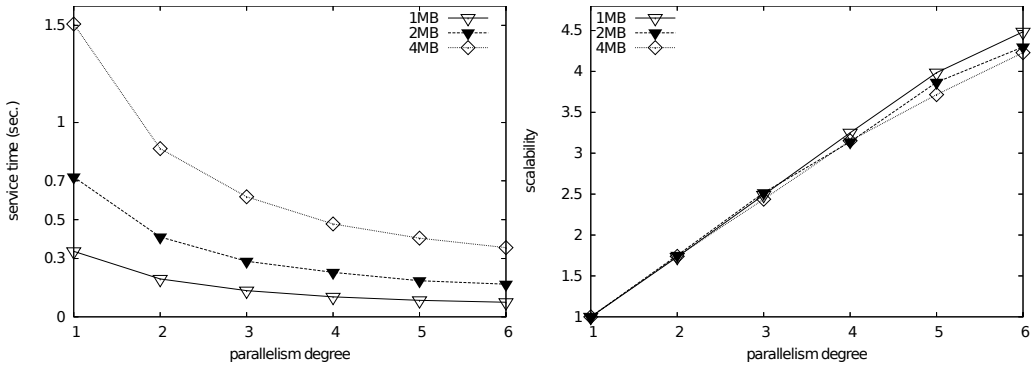
I tempi di servizio ottenuti possono quindi essere considerati anche nell'ottica di conoscere il tempo di risposta effettivo dell'applicazione nella situazione di una emergenza.

Abbiamo considerato come possibile nodo di interfaccia due architetture molto differenti:

- un server di ultima generazione, dotato di due processori Intel E5420 Quad Core, per un totale di 8 Core a 2.5Ghz di frequenza, 12 MB di cache e 8GB di memoria principale.

- un server dotato di processore IBM Cell, architettura multicore sostanzialmente differente dalla precedente a causa dei core non omogenei. Questo è stato emulato con una PS3, dotata di processore Cell e 256MB di memoria principale.

Piattaforma Intel E5420 Sulla prima piattaforma abbiamo eseguito sia la versione farm che la data-parallel, ed abbiamo ottenuto risultati molto buoni per entrambe; in questa architettura, grazie alla memoria condivisa, possiamo realizzare in modo efficiente anche la *Parallel Cyclic Reduction*, che diventa un'ottima soluzione nel caso la macchina non abbia abbastanza memoria per ospitare le versioni farm. Anche sulla macchina da noi utilizzata, infatti, la dimensione massima delle matrici per il farm risulta essere 2^{20} righe: già con matrici grandi 2^{21} righe avremmo bisogno di più di 10 GB di ram al grado di parallelismo 8. La versione data-parallel, invece, è stata eseguita con matrici fino a 2^{25} righe.



(a) Tempo di servizio (b) Scalabilità

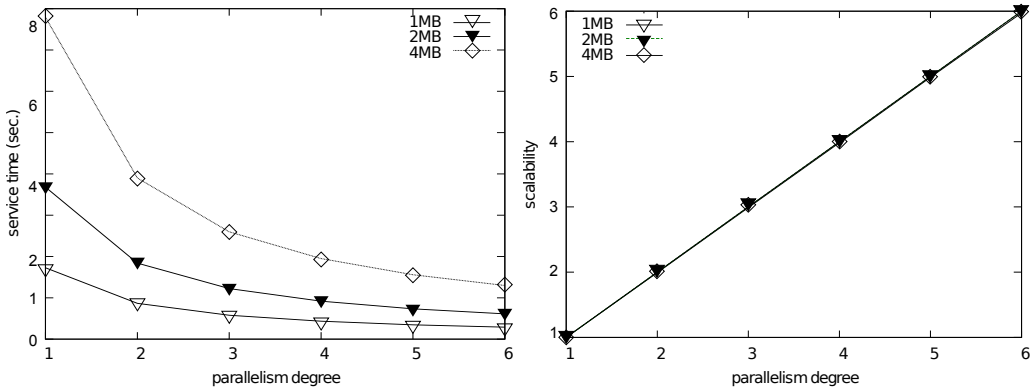
Figura 6.7: Prestazioni del data-parallel di Parallel Cyclic Reduction sul server Intel Dual Quad Core

Il codice utilizzato è lo stesso della versione per cluster, e eseguito utilizzando il supporto per memoria condivisa. Questo ci ha permesso di riutilizzare il codice già sviluppato, al costo di una perdita di prestazioni.

Allo stesso modo, per comportamenti non su stream la versione data-parallel risulta più veloce, in quanto, con un numero adeguato di nodi, la *Parallel Cyclic Reduction* ottiene una latenza minore rispetto alla versione sequenziale di *Cyclic Reduction*. In figura 6.7 riportiamo i risultati, per matrici più piccole: da 2^{15}, 2^{16} e 2^{17}, in quanto verosimilmente nei casi di esecuzione sul nodo di interfaccia siamo nel mezzo di una emergenza ed i risultati richiesti dagli operatori devono essere prodotti in un tempo molto basso. La scalabilità del data-parallel è buona, ma a causa delle comunicazioni dello stencil non sovrapposte al calcolo non riesce comunque a raggiungere quella teorica.

La non perfetta scalabilità è dovuta in buona parte all'utilizzo di un sistema a scambio di messaggi: in futuro potremmo anche studiare una versione ottimizzata per l'utilizzo diretto della memoria condivisa, ed evitare le copie dei dati, con un approccio simile a quello riportato in [12]; in questo modo la scalabilità dovrebbe migliorare notevolmente ed avvicinarsi a quella teorica.

Piattaforma IBM Cell In questo caso non abbiamo potuto utilizzare la versione MPI, in quanto al momento non esistono implementazioni pubbliche di questa libreria sul Cell. Sviluppare un programma su un processore Cell richiede un approccio differente a causa della non omogeneità dei core: se vogliamo utilizzare al meglio le 8 SPU dobbiamo scrivere del codice sequenziale ottimizzato ad-hoc per queste unità.



(a) Tempo di servizio (b) Scalabilità

Figura 6.8: Prestazioni del data-parallel di Parallel Cyclic Reduction sul server IBM Cell

In questo caso la memoria limitata delle SPU (256KB di local storage, da utilizzare per programma e dati) rende necessaria una implementazione "intelligente" per minimizzare i trasferimenti dalla memoria principale. A causa di una grande difficoltà di implementazione per la versione data-parallel, abbiamo realizzato solo la versione farm.

Un altro grosso limite della macchina da noi utilizzata è la quantità di memoria ram disponibile, di soli 256MB. In questo caso una implementazione farm con la *Cyclic Reduction* originale diventa molto limitata, in quanto possiamo utilizzare matrici grandi al massimo 1MB (2^{15} righe). Sopra questo limite il programma continua ad essere eseguibile, in quanto il sistema utilizza lo spazio di swap, ma le prestazioni vengono compromesse in modo significativo. Abbiamo perciò creato un farm di *Parallel Cyclic Reduction*, che ci permette di risolvere sistemi con dimensioni analoghe a quelli delle altre piattaforme (fino a 2^{19} righe, 16MB).

I risultati di questa versione sono raccolti nella figura 6.8. I tempi di questa versione sono direttamente confrontabili con quelli del data-parallel sul processore Intel, in quanto entrambi sono eseguiti sullo stesso algoritmo, stesse dimensioni ma forme parallele differenti. Rispetto all'Intel i tempi sono più alti a causa dell'architettura differente, ma lavorando con un farm abbiamo una scalabilità molto vicina a quella ideale.

6.5.3 Versione per PDA

Nell'ottica di utilizzare i dispositivi pervasivi abbiamo anche effettuato un primo test su un PDA in nostro possesso. Si tratta di un ARM 926T a

Righe Matrice	Dimensione Sistema	Tempo di Calcolo
$2^{12}-1$	128K	3.84
$2^{13}-1$	256K	8.08
$2^{14}-1$	512K	17.16
$2^{15}-1$	1M	36.08
$2^{16}-1$	2M	75.88
$2^{17}-1$	4M	157.92

Tabella 6.1: Prestazioni della versione sequenziale di Parallel Cyclic Reduction sul PDA

300Mhz, dotato di 64MB di ram. Su questo sistema non abbiamo abbastanza memoria per poter eseguire la *Serial Cyclic Reduction* su matrici di dimensioni considerevoli; da questo punto di vista si comporta decisamente meglio la *Parallel Cyclic Reduction*, che richiede una quantità di memoria di circa 2 volte la dimensione della matrice in ingresso (contro le $log_2(N)$ della Serial). Su questo sistema siamo riusciti a risolvere sistemi con matrici grandi fino a 8 MegaByte (2^{18} righe) con la versione Parallel, mentre con la versione Serial possiamo arrivare al massimo ad 1 MegaByte (corrispondenti a 2^{15} righe). Il guadagno è notevole, ma si paga con prestazioni minori.

Abbiamo effettuato un test sequenziale per la *Parallel Cyclic Reduction*, che ci ha dato i risultati riportati in tabella 6.1. Questi tempi possono essere utilizzati per ottenere una stima di una eventuale parallelizzazione di tipo farm: con una scalabilità vicina alla ideale ci basterebbero pochi dispositivi per ottenere tempi ragionevoli (di qualche secondo) con sistemi da 1MB; assumendo un buon dispiego di personale possiamo sperare di avere anche 20-30 PDA nella zona dell'emergenza, e sfruttandoli tutti potremmo raggiungere tempi di servizio decenti anche con matrici più grandi.

6.6 Le riconfigurazioni proposte

A questo punto abbiamo abbastanza elementi per poter stimare, in una infrastruttura come quella di gestione delle emergenze, come e quando utilizzare le differenti versioni di Risolutore di Sistemi in casi critici.

Condizioni ideali

In una situazione ideale tutta l'infrastruttura informatica è presente ed utilizzabile. Abbiamo quindi a disposizione il cluster in un centro di calcolo remoto, un nodo di interfaccia ed un insieme di PDA. In questa situazione

un agente presente sul posto richiede una previsione a medio termine per la zona in cui si trova. Non impone nessun vincolo sulla qualità del risultato ottenuto, ma specifica un tempo massimo di esecuzione: entro 10 minuti la previsione deve essere visualizzata sul suo PDA.

Per semplicità assumiamo che l'applicazione ASSISTANT di figura 6.5 sia già in esecuzione, con la seguente disposizione:

- **Generator** sul nodo Sink, che riceve i dati direttamente dai sensori e può generare le richieste di elaborazione per i singoli punti.
- **Tridiagonal Solver** sul cluster, già allocato a causa di una precedente richiesta.
- **Viewer** sul PDA dell'agente che ha richiesto la previsione.

Nella figura 6.5 sono illustrati solo gli stream per la computazione; a questi aggiungiamo inizialmente due ulteriori stream, che portano ad una struttura illustrata in figura 6.9a, per i seguenti eventi:

1. *start*, inviato dal Viewer al Generator per richiedere l'inizio di una computazione;
2. *serviceTime*, inviato sempre dal Viewer ma questa volta al Tridiagonal Solver per richiedere un certo tempo di servizio massimo.

Con questi due stream addizionali possiamo modellare interamente la richiesta di una nuova previsione entro un tempo limite. Quando l'evento *start* arriva al Generator questo inizia la generazione dei dati per una nuova previsione, suddividendo l'area con la precisione richiesta dal Viewer tramite l'evento, e li invia al Tridiagonal Solver. Intanto, anche quest'ultimo ha ricevuto un evento, che lo porterà a riconfigurarsi mediante un cambio di grado di parallelismo, per poter rispettare la richiesta ricevuta dal Viewer con un numero di nodi sufficiente e al tempo stesso non eccessivo. L'evento *serviceTime* non contiene esattamente il tempo di risposta per l'intera previsione, in quanto il modulo Tridiagonal Solver lavora su piccole parti di essa in modo indipendente. Il Viewer, invece, ha una visione più generale della previsione: sa in quante parti è suddivisa (ha specificato tale valore sull'evento *start*), quante ne ha già ricevute e può quindi conoscere il numero di risoluzioni devono ancora essere eseguite. Da questi dati può richiedere un tempo di servizio che porti alla fine dell'intera previsione entro il tempo specificato dall'utente. Successivamente il modulo Viewer potrà inviare nuovi eventi *serviceTime* per aggiustare il tempo di servizio medio del modulo risolutore.

Possiamo già iniziare a descrivere l'**event-operation graph** del modulo risolutore (figura 6.9b): abbiamo una unica operation (clusterOP) e un unico evento, che genera una riconfigurazione non funzionale.

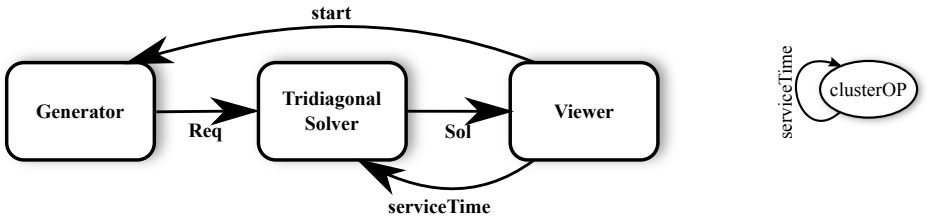
Dopo qualche minuto l'operatore potrebbe accorgersi di aver sbagliato stime e di poter aspettare più o meno tempo rispetto a quello inizialmente comunicato al modulo; in questo caso può richiedere questa modifica al Viewer che invierà nuovi eventi di tipo *serviceTime* al risolutore.

Problemi sulla connessione al cluster

Durante la computazione potrebbero verificarsi gravi problemi alla connessione tra nodo di interfaccia e cluster. Il cluster potrebbe diventare irraggiungibile, ed è necessario passare ad una computazione sui nodi rimasti. Assumiamo invece che rimanga stabile la connessione locale, quella tra PDA, sensori e nodo di interfaccia. Viene generato un evento, *mainNetOff*, da una entità esterna ai moduli. Vedremo nel prossimo capitolo che questa verrà modellata come una interfaccia primitiva, ma al momento non è necessario approfondire l'argomento.

In questa situazione il luogo migliore dove eseguire il risolutore tridiagonale è sicuramente il nodo di interfaccia. Come abbiamo visto nella sezione precedente in questo caso dobbiamo (o ci conviene) cambiare operation, e passare ad una implementazione della *Parallel Cyclic Reduction*. Assumiamo di avere come nodo di interfaccia il server con processore Intel, perciò la scelta ricade su una versione data-parallel; chiamiamo questa operation *InterfaceNodeOP*.

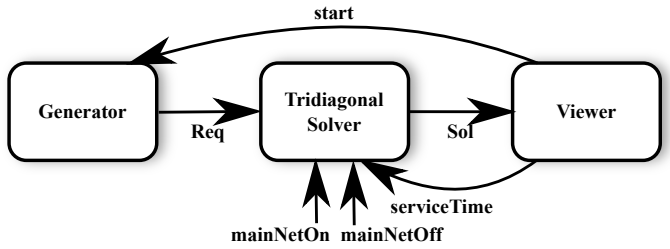
Abbiamo perciò un cambio di operation; in corrispondenza della nuova allocazione dobbiamo anche decidere come configurare il modulo, in modo da rispettare il tempo di risposta richiesto dall'utente. Questa volta possiamo



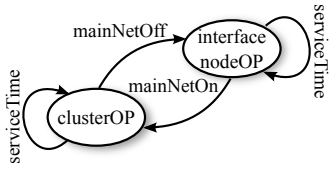
(a) Grafo di moduli e stream

(b) Event-Operation Graph del Tridiagonal Solver

Figura 6.9: Il primo passo di modellazione per l'esempio di applicazione ASSISTANT ed il relativo grafo di operation



(a) Grafo di moduli e stream



(b) Event-Operation Graph del Tridiagonal Solver

Figura 6.10: Il secondo passo di modellazione per l'esempio di applicazione ASSISTANT ed il relativo grafo di operation

modificare il grado di parallelismo fino ad un massimo di 8, ma verosimilmente questo non basterà per rispettare i requisiti dell'agente. Sfruttiamo allora la possibilità di scegliere una dimensione di matrice più piccola, per rispettare i vincoli dati.

Anche in questo caso prevediamo la presenza dell'evento *responseTime*, che non causa un cambio di operation ma una semplice riconfigurazione non funzionale. Infine prevediamo un ulteriore evento, *mainNetOn*, per rappresentare il ritorno della connessione verso il cluster, che causerà un passaggio alla operation *clusterOP*.

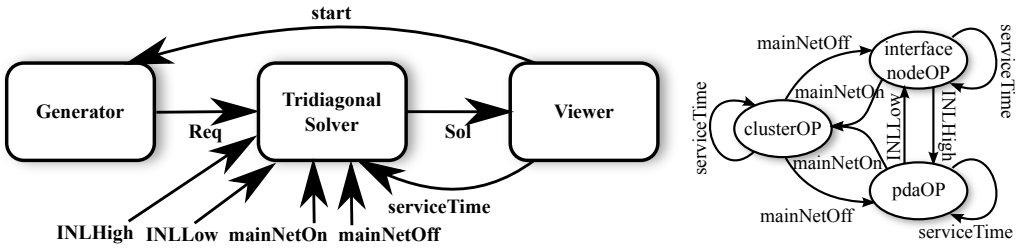
In figura 6.10 sono riportati il grafo dei moduli e l'event-operation graph modificati.

Nodo di interfaccia sovraccarico

Per ultimo, consideriamo la situazione in cui il nodo di interfaccia sia sovraccarico e che non possa sopportare anche il carico di lavoro richiesto dal modulo Tridiagonal Solver. Abbiamo sempre a disposizione l'insieme di PDA per poter eseguire una veloce approssimazione dell'intera simulazione. Introduciamo perciò una nuova operation: *pdaOP* che, in base ai risultati visti precedentemente, implementa un farm di *Parallel Cyclic Reduction* sequenziali.

Anche questa operation prevederà una riconfigurazione nel caso di ricezione dell'evento *responseTime*, esattamente come le altre. Si possono aggiungere PDA precedentemente non utilizzati, oppure diminuire ulteriormente la dimensione della matrice.

Inseriamo due nuovi eventi: Interface Node Load High (abbreviato con *INLHigh*) ed il suo complementare *INLLow*. Con questo evento definiamo lo switch tra *interfacenodeOP* e *pdaOP*. Questo non ci basta per modellare



(a) Grafo di moduli e stream

(b) Event-Operation Graph del Tridiagonal Solver

Figura 6.11: Il terzo ed ultimo passo di modellazione per l'esempio di applicazione ASSISTANT ed il relativo grafo di operation

tutte le possibilità: dobbiamo modificare anche le reazioni all'evento *mainNetOff*, come riportato in figura 6.11b. Stiamo definendo una transizione da clusterOP, per il solito evento, verso due differenti operation in base allo stato interno: il modulo dovrà infatti decidere, in base all'attuale carico del nodo di interfaccia, quale delle due utilizzare.

A questo punto il grafo delle riconfigurazioni è terminato; riportiamo in figura 6.11 la versione finale dell'applicazione. Vedremo, nel prossimo capitolo, come descrivere tutto questo all'interno di ASSISTANT.

6.7 Conclusioni

In questo capitolo è stata analizzata in modo approfondito una applicazione con requisiti di adattività, context awareness e alte prestazioni. Abbiamo riportato questo studio per fornire un esempio di possibile applicazione ASSISTANT completa e, allo stesso tempo, per dare un'idea dei passi da seguire durante l'analisi di una applicazione di questo tipo.

L'applicazione (o almeno la sua semplificazione presentata) si presta ad essere modellata con il modello presentato in questa tesi. Abbiamo discusso le caratteristiche di varie implementazioni del solito modulo per architetture differenti e alcune politiche su cui definire le riconfigurazioni.

Non è stata ancora fornita una vera implementazione ASSISTANT, in quanto, fino a questo momento, abbiamo descritto le caratteristiche del modello ma non la sua sintassi. Questa verrà presentata nel prossimo capitolo e, finalmente, potremo scrivere l'esempio qui presentato in ASSISTANT.

Capitolo 7

ASSISTANT: ASSIST with Adaptivity and coNText awareness

Vediamo ora in dettaglio il modello ASSISTANT, presentandone la sintassi ed illustrando come questa aderisce ai concetti visti nei capitoli precedenti.

Il modello non è ancora definito in modo ufficiale, e stiamo studiando continuamente modifiche e miglioramenti che permettano di rappresentare i concetti che consideriamo importanti per le applicazioni pervasive ad alte prestazioni.

Quello che presentiamo non è perciò un lavoro ben definito e necessariamente coerente, ma piuttosto il risultato di un primo studio sommario di tutte le caratteristiche richieste e delle considerazioni fin'ora riportate. Ci rendiamo conto che una trattazione sommaria di alcuni aspetti può aver portato alla mancanza di parti anche fondamentali nel modello, che verrà quindi esteso in futuro, in base ai risultati degli studi più approfonditi.

Nell'ottica di dare un'idea più completa del modello e della sua applicabilità useremo in modo intensivo l'esempio del capitolo 6, molto significativo in quanto parte di una applicazione reale. In questo modo alla fine del capitolo avremo raggiunto anche una implementazione ASSISTANT completa di una applicazione pervasiva ad alte prestazioni.

Vedremo inizialmente, nella sezione 7.1, come inserire tutte le caratteristiche che abbiamo descritto all'interno di un modello derivato da ASSIST, per dare forma al linguaggio di ASSISTANT. Dopo aver descritto i concetti principali passeremo alla definizione del linguaggio stesso: come descrivere una applicazione (sezione 7.2), la nuova sintassi per descrivere un modulo parallelo (sezione 7.3) e, per finire, come definire i comportamenti adattivi e context-aware di un modulo (sezione 7.4).

7.1 Mantenere il modello unificante di ASSIST

Amalgamare le estensioni proposte nel capitolo 5 all'interno del modello di ASSIST, mantenendone la filosofia e le caratteristiche principali, ci ha portato a definire un nuovo approccio nello sviluppo di una applicazione parallela adattiva, in cui:

- Una applicazione è sempre definita come un grafo di moduli interconnessi da stream.
- I moduli applicativi ereditano le caratteristiche dei parmod di ASSIST.
- La *riconfigurazione* è supportata nativamente dal modello, e allo sviluppatore viene richiesto solo di definire le differenti versioni dei moduli per le riconfigurazioni funzionali.
- Le entità dello strato BLUE fanno parte dell'applicazione. Non vengono programmate a parte con un differente formalismo, ma in modo consistente e coerente al resto dell'applicazione.
- Queste sono perciò inserite nella descrizione dei singoli moduli, in modo che il programmatore possa descrivere, in modo coerente, sia le versioni che il comportamento adattivo e context aware di un modulo.
- Allo stesso modo, anche la modellazione del contesto rientra nel modello: fa parte dell'applicazione e viene descritto attraverso moduli che producono eventi.
- I dati del contesto e sulle risorse vengono perciò modellati come *eventi*.
- I sensori vengono modellati tramite moduli particolari, la cui implementazione viene già fornita in modo nativo dall'ambiente, chiamati *Interfacce Primitive*.
- I meccanismi di fault tolerance sono introdotti in modo trasparente all'utente e, a differenza della maggior parte dei lavori presenti in letteratura, possono utilizzare protocolli ottimizzati basati sulla conoscenza della struttura parallela dei vari moduli.

Utilizzando l'esperienza maturata in ASSIST abbiamo deciso di descrivere un programma ASSISTANT mediante un *linguaggio di coordinamento*, che definisce la *struttura* del programma con una sintassi tipo linguaggio imperativo, fornendo costrutti specifici per la descrizione di programmi paralleli

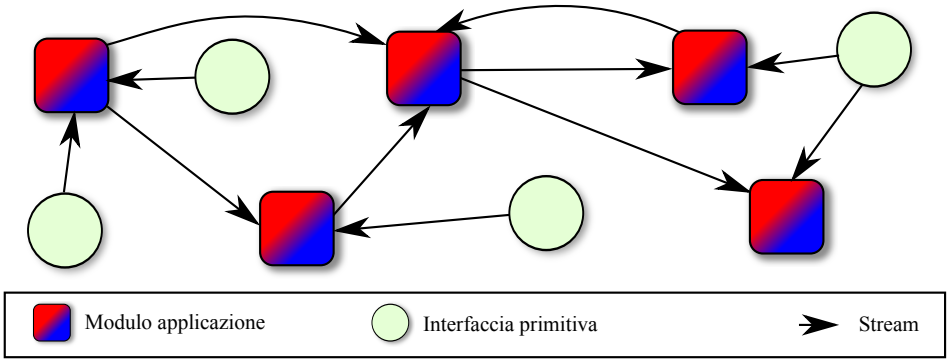


Figura 7.1: Grafo di una applicazione ASSISTANT

adattivi. Per il codice sequenziale da eseguire nei singoli moduli, come per ASSIST, è possibile utilizzare codice *C/C++*. Per ora non pensiamo di introdurre il supporto a Fortran per non complicare ulteriormente il supporto, ma non certo per problemi di tipo modellistico.

Il *linguaggio di coordinamento* è derivato da ASSIST-CL, è fortemente tipato e ne eredita i meccanismi per la definizione di tipi da parte dello sviluppatore.

Anche con ASSISTANT non è previsto l'uso di oggetti. Questo per vari motivi, ma principalmente per garantire la compatibilità con linguaggi imperativi come il *C*. Permettere l'utilizzo di un gruppo di linguaggi sequenziali diversi richiede infatti che i tipi del *linguaggio di coordinamento* siano compatibili con quelli di ogni linguaggio, e siamo quindi costretti a definire un sottoinsieme di caratteristiche minimo che sia supportabile in tutti. In ogni caso, all'interno del codice sequenziale lo sviluppatore può sfruttare in tutte le sue potenzialità il linguaggio scelto.

7.2 Il grafo di moduli

La struttura di una applicazione ASSISTANT è rappresentabile, come per ASSIST, da un grafo di moduli, interconnessi tra stream. In questa struttura i tre strati RED-BLUE-GREEN, seppur individuabili, sono inseriti in una unica descrizione. Infatti l'uniformare stream ed eventi ci porta ad una naturale fusione degli strati GREEN e RED, in linea con la nostra idea di "differenti visioni della stessa applicazione", in cui tutti i flussi di dati (siano essi provenienti da moduli o da interfacce primitive; eventi o dati su stream) sono modellati tramite un unico grafo. Nella figura 7.1 troviamo una rappresentazione del grafo di moduli di una applicazione ASSISTANT. Utilizzando

i tre colori mostriamo come la stessa applicazione verrebbe modellata utilizzando il concetto di "strati" visto precedentemente. Un modulo applicativo è composto da una parte RED ed una BLUE, in quanto la sua descrizione contiene sia la parte "computazionale" che il comportamento adattivo e context aware. Allo stesso livello troviamo anche le interfacce primitive, che rappresentano il GREEN.

Un programma ASSISTANT rimane quindi, come per il predecessore, espresso attraverso la descrizione di un grafo di moduli. La sintassi è molto simile a quella di ASSIST, ma con alcune ovvie differenze. Il grafo viene definito tramite il costrutto *application*, con la sintassi riportata nel listato 7.1, che permette di dichiarare:

- gli stream tipati, tramite la parola chiave *stream*;
- i moduli, secondo le due tipologie presenti in ASSISTANT:
  - moduli paralleli che compongono la logica applicativa, tramite la parola chiave *parmod* ereditata da ASSIST
  - moduli che rappresentano le interfacce primitive e producono eventi, con la parola chiave *primitive_interface*
- Le connessioni tra moduli nella forma di *stream di input* e *stream di output*.

Con questa sintassi possiamo descrivere anche l'applicazione previsionale vista nel capitolo precedente. Per far ciò dobbiamo però introdurre alcune interfacce di contesto: un lettore attento si ricorderà che nella figura 6.11a erano rimasti degli stream "esterni", di cui non si descriveva il produttore. Inseriamo perciò due interfacce di contesto, per modellare altrettanti "generatori di eventi":

- **Network_Monitor**: una interfaccia primitiva per il monitoring della rete; nel nostro caso siamo interessati a monitorare la rete tra Nodo di Interfaccia e Centro di calcolo, per la generazione dei due eventi "mainNetOff" e "mainNetOn".
- **Load_Monitor**: interfaccia primitiva per monitorare il carico computazionale di un nodo; genera gli eventi "Interface Node Load High" e "Interface Node Load Low".

A questo punto abbiamo tutto il necessario per poter descrivere l'applicazione: nel listato 7.2 trovate la definizione del grafo di moduli, e nella figura 7.2 la sua rappresentazione grafica. In questa, per facilitarne la lettura, abbiamo deciso di colorare gli stream, in modo da distinguere velocemente quali

```
application {
  stream <tipoStream_1> <nomeStream_1>;
    ...
  stream <tipoStream_M> <nomeStream_M>;

  parmod <nomeModulo_1>
      (input_stream<nomeStream_{1,1}>,...,<nomeStream_{1,K_1}>
      output_stream<nomeStream_{1,K_1+1}>,...,<nomeStream_{1,L_1}>);
    ...
  parmod <nomeModulo_N>
      (input_stream<nomeStream_{N,1}>,...,<nomeStream_{N,K_N}>
      output_stream<nomeStream_{N,K_N+1}>,...,<nomeStream_{N,L_N}>);

  primitive_interface <nomeModulo_{N+1}>
      (input_stream<nomeStream_{N+1,1}>,...,<nomeStream_{N+1,K_{N+1}}>
      output_stream<nomeStream_{N+1,K_{N+1}+1}>,...,<nomeStream_{N+1,L_1}>);
    ...
  primitive_interface <nomeModulo_P>
      (input_stream<nomeStream_{P,1}>,...,<nomeStream_{P,K_P}>
      output_stream<nomeStream_{P,K_P+1}>,...,<nomeStream_{P,L_P}>);
}
```

Listato 7.1: Sintassi per la dichiarazione del grafo dei moduli che compone l'applicazione ASSISTANT

contengono dati per le computazioni e quali eventi. Sottolineiamo però come questa distinzione non appaia (almeno, non a questo livello) nel modello.

Spieghiamo velocemente i tipi di dato utilizzati per gli eventi:

- **serviceTime** rappresenta un tempo di servizio in secondi, quindi la sua rappresentazione ideale è un numero in virgola mobile;
- **start** è l'evento per avviare una nuova previsione; conterrà un valore, intero, per specificare il numero di partizioni in cui suddividere lo spazio dell'emergenza;
- **mainNetOff** e **mainNetOn** sono eventi di tipo "acceso"-"spento": la rete è disponibile oppure no; tutta l'informazione è già racchiusa nell'invio dell'evento, che potrebbe quindi non avere tipo, ma questo non è possibile e scegliamo quindi un semplice booleano;
- **InterfaceNodeLoadLow** e **High**, oltre a segnalare un carico eccessivamente alto o basso del nodo, possono informarci dell'effettivo livello di utilizzazione, in percentuale.

```
application{
  stream Request Req;
  stream Solution Sol;

  stream double serviceTime;
  stream long start;

  stream bool mainNetOff;
  stream bool mainNetOn;
  stream double INLLow;
  stream double INLHigh;

  parmod Generator(input_stream start output_stream Req);
  parmod TridiagonalSolver(input_stream Req, serviceTime,
      mainNetOff, mainNetOn, INLLow, INLHigh output_stream Sol);
  parmod Viewer(input_stream Sol output_stream
      start,serviceTime);

  primitive_interface Load_Monitor(output_stream
      INLLow,INLHigh);
  primitive_interface Network_Monitor(output_stream
      mainNetOff,mainNetOn);
}
```

Listato 7.2: Definizione del grafo dei moduli dell'applicazione trattata nel capitolo 6

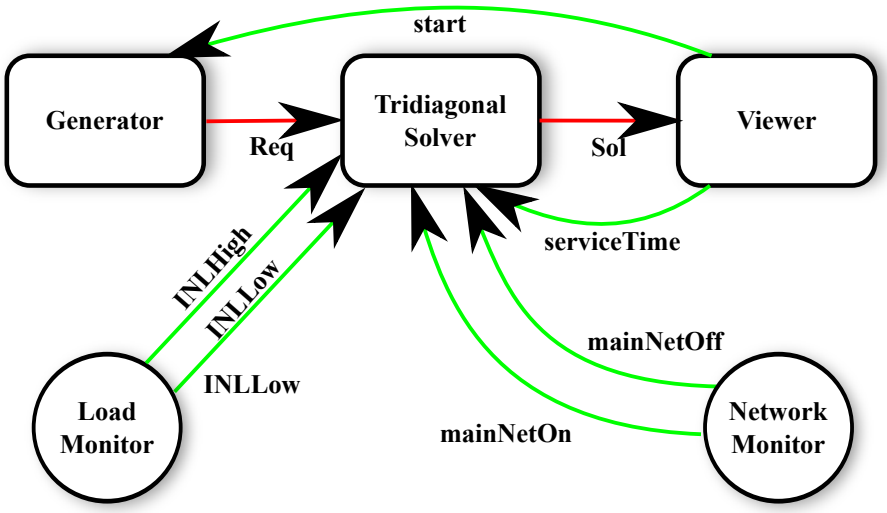


Figura 7.2: Grafo dei moduli corrispondente al listato 7.2

Differenti tipologie di moduli

Rispetto ad ASSIST sono state profondamente modificate le tipologie di moduli esistenti. Abbiamo mantenuto il *modulo parallelo*, in quanto punto centrale del modello, ma quello *sequenziale* è stato rimosso per far posto ad una nuova tipologia di moduli, le *interfacce primitive*.

7.2.1 Il modulo sequenziale

La prima differenza riguarda i moduli sequenziali: questi non sono più presenti in ASSISTANT, in quanto ritenuti superflui. Questa decisione non impatta sull'espressività del linguaggio, in quanto è sempre possibile definire moduli paralleli costituiti da un solo processore virtuale, che si comportano quindi a tutti gli effetti come moduli "sequenziali". Permette invece di amalgamare la definizione dei moduli, trasformando la versione sequenziale da entità separata a caso particolare di quello parallelo. In ASSISTANT, quindi, il concetto di *parmod* corrisponde a quello di modulo applicativo.

7.2.2 Le interfacce primitive

Abbiamo introdotto questa tipologia di modulo per modellare principalmente le interfacce di contesto. Possono essere viste come una sorta di "adattatori", che si occupano di trasformare i segnali ricevuti dai sensori (in qualsiasi forma essi siano) in uno stream di dati o eventi, e permettere l'interazione di questi dispositivi con i moduli dell'applicazione ASSISTANT. Questa conversione, necessaria per far interagire i dispositivi con le applicazioni, non può essere effettuata automaticamente dal supporto, in quanto abbiamo già visto come ogni tipologia di sensore utilizzi api differenti per lo scambio di informazioni.

Inoltre possiamo realizzare come interfacce primitive anche meccanismi software di monitoring delle risorse, come abbiamo già anticipato attraverso i moduli "Load Monitor" e "Network Monitor" descritti precedentemente.

Ipoteticamente il contenuto di queste interfacce dovrebbe essere fornito dai produttori di sensori o dal supporto del modello (da qui il nome di "interfaccia primitiva"), ma nel caso questi non lo offrano può essere sempre definito dal programmatore dell'applicazione.

La tecnica di programmazione di questi moduli non è stata ancora studiata a fondo; per il momento proponiamo un modello simile a quello dei moduli sequenziali ASSISTANT, dove il contenuto del modulo è dato da un codice sequenziale scritto dall'utente, nel linguaggio C++. Nel listato 7.3 trovate la sintassi proposta al riguardo. Per permettere l'inclusione di file

```
primitive_interface <nomeModulo>(
  input_stream <tipo_1> <nome_1>,...,<tipo_K> <nome_K>
  output_stream <tipo_K+1> <nome_K+1>,...,<tipo_M> <nome_M>){
inc< <sorgente_1>,...,<sorgente_L> >
path< <directory_1>,...,<directory_S> >
obj< <binario_1>,...,<binario_G> >
{
  codice sequenziale in C++
}
```

Listato 7.3: Sintassi per la definizione di una interfaccia primitiva

sorgenti e binari esterni, abbiamo riutilizzato la sintassi già definita per le proc ASSIST.

Le modalità di definizione di interfacce primitive "esterne", sviluppate da terzi (ad esempio i produttori di sensori) non sono state ancora studiate in dettaglio. A livello di modello ci aspettiamo di poter definire un meccanismo di *include*, simile a quello dei linguaggi sequenziali, per poter utilizzare moduli "esterni", senza doverne dare la rispettiva implementazione.

7.3 Il parmod

Il modulo parallelo rimane l'entità base di ASSISTANT per descrivere un'applicazione parallela. Eredita le caratteristiche base di ASSIST, ma viene esteso per definire le riconfigurazioni funzionali e la logica di adattività e context-awareness.

Un parmod racchiude più comportamenti, chiamati *operation*; la sua definizione coincide quindi con quella delle varie operation che lo compongono. Tra queste lo sviluppatore dovrà decidere quella attiva al momento dell'avvio dell'applicazione. Oltre alle operation aggiungiamo una sezione (chiamata *global_state*) che permette di definire uno stato "comune" a tutte queste; vedremo più avanti nella trattazione il significato e la sua utilità. Abbiamo quindi uno schema come quello riportato nel listato 7.4.

Possiamo quindi già iniziare a descrivere il modulo ASSISTANT più importante nell'esempio di applicazione del capitolo 6: il **Tridiagonal Solver**. Abbiamo tre operation: "clusterOP","interfacenodeOP" e "pdaOP". Per il momento lasciamo lo stato globale vuoto; più avanti, se necessario, lo modificheremo. Lo scheletro del modulo è presente in figura 7.5.

```
parmod <nomeModulo>(
    input_stream <tipo_1> <nome_1>,...,<tipo_K> <nome_K>
    output_stream <tipo_K+1> <nome_K+1>,...,<tipo_M> <nome_M>){
  global_state{ ... }
  initial operation <nomeOperation_1> { ... }
  operation <nomeOperation_2> { ... }
    ...
  operation <nomeOperation_N> { ... }
}
```

Listato 7.4: Sintassi per la dichiarazione di un parmod composto da più operation

```
parmod TridiagonalSolver(input_stream Request Req, double
    serviceTime, bool mainNetOff, bool mainNetOn, double INLLow,
    double INLHigh output_stream Solution Sol){

  global_state{}

  initial operation clusterOP{ ... }
  operation interfacenodeOP{ ... }
  operation pdaOP{ ... }
}
```

Listato 7.5: Scheletro del parmod "Tridiagonal Solver"

7.3.1 Le operation

Ogni singola operation definisce in modo completo il comportamento del modulo parallelo; per questo motivo all'interno di essa troviamo la sintassi tipica del *parmod* ASSIST, suddiviso in:

- Definizione della *topologia*,
- Sezione di *stato*,
- Sezione di *input*,
- Sezione dei *processori virtuali*,
- Sezione di *output*,

La sintassi è rimasta invariata, se non per la dichiarazione dello stato interno, raggruppata in una nuova sezione apposita chiamata *local_state*. Il nome della sezione è stato scelto per sottolineare la caratteristica di "località" di quest'ultimo, differente da quello visto precedentemente in quanto legato alla

singola operation. In ogni caso il contenuto riprende la sintassi di ASSIST per **attributi**, **stream interni** e la parte di inizializzazione delimitata dal costrutto **init**.

Il resto è stato trattato in modo dettagliato nel capitolo 4, e perciò non ci soffermiamo ulteriormente su questo punto.

A quelle ereditate dal parmod ASSIST si aggiungono però due nuove sezioni:

- *on_event*, necessaria per descrivere il comportamento in risposta ad un evento;
- *nodes*, necessaria per descrivere quali nodi utilizzare per eseguire l'operation.

La sezione *nodes*

Con questa sezione possiamo definire i nodi su cui eseguire la particolare operation. Nell'ottica di definire comportamenti specifici per particolari tipologie di risorse, questa sezione è fondamentale: qui possiamo descrivere l'insieme di nodi da utilizzare. Almeno in una prima parte dello sviluppo permetteremo anche di definire staticamente i nodi su cui eseguire l'operation, in modo da non dover utilizzare un motore di resource discovery. Quando questo sarà sviluppato ed abbastanza potente potremo anche rimuovere la parte di definizione statica.

Per la parte dinamica, il programmatore deve definire solamente una o più tipologie dei nodi; una volta scelta l'operation, il motore di resource discovery cercherà un numero sufficiente di nodi con quelle particolari tipologie. I tipi di nodo non sono definiti staticamente, ma forniti dal programmatore stesso all'interno della definizione dell'applicazione ASSISTANT. Ogni nodo disponibile ad eseguire una parte di applicazione dovrà in qualche modo presentarsi al sistema di resource discovery, e in quel momento comunicherà la sua tipologia.

Nel listato 7.6 trovate una prima sintassi per la sezione nodes, mentre nel 7.7 la parte di definizione dei tipi di nodo presenti nel sistema. Questa si trova all'esterno della definizione del parmod, in quanto comune a tutta l'applicazione.

Per la parte statica, invece, richiediamo al programmatore l'insieme minimo di caratteristiche necessarie per comunicare con la macchina: indirizzo ip e (probabilmente) sistema operativo. Con queste il sistema è in grado di

```
nodes{
  dynamic{
    use node_type <typeName_1>;
    ...
    use node_type <typeName_N>;
  }
  static{
    use node{ address=<ad_1>; operating_system=<os_1>; }
    ...
    use node{ address=<ad_M>; operating_system=<os_M>; }
  }
}
```

Listato 7.6: Sintassi per la definizione dei nodi da utilizzare per l'esecuzione della operation

```
node_types{
  type <typeName_1>;
  ...
  type <typeName_N>;
}
```

Listato 7.7: Sintassi per la definizione dei tipi di nodo presenti nell'applicazione

comunicare con la parte di supporto "locale" in esecuzione sul nodo stesso, e definire automaticamente tutte le restanti caratteristiche (set di istruzioni del processore, meccanismi di comunicazione, etc).

La sezione *on_event*

La sezione fondamentale per definire un modulo reattivo agli eventi: al suo interno possiamo definire le azioni da intraprendere alla ricezione di un evento. Vista l'importanza di questo nuovo costrutto, ne approfondiremo il significato successivamente in una sezione apposita.

Per non appesantire la lettura, in questo caso non forniamo l'intera sintassi del costrutto *operation*, ma riportiamo in dettaglio (figura 7.8) la definizione della "clusterOP"; la sintassi generale è facilmente ricavabile, in quanto identica (se non per piccole parti) a quella del parmod ASSIST fornita nel capitolo 4. Nell'esempio abbiamo lasciato vuota la sezione on_event, che verrà trattata in modo approfondito più avanti. Per quanto riguarda la sezione nodes, chiediamo di scegliere dinamicamente tra i nodi di tipo "clusterno-

```
initial operation clusterOP{
  topology none Pv;
  local_state { }
  do input_section{
    guard1: on , , System {
      distribution System on_demand to Pv;
    }
  }while(true)
  virtual_processors {
    elaborazione( in guard1 out Sol){
      VP {
        FTridiagonalSolverWithCyclicReduction(in Req out Sol);
      }
    }
  }
  output_section{
    collects Sol from ANY Pv;
  }
  nodes{
    dynamic{ use node_type clusternode; }
  }
  on_event{ ... }
}
```

Listato 7.8: Definizione della operation "clusterOP" del "Tridiagonal Solver"

de", ovvero le macchine che fanno parte del cluster localizzato nel centro di calcolo.

7.3.2 Lo stato globale

In ASSISTANT abbiamo inserito una distinzione sulle variabili di stato, che ha portato alla definizione di una sezione **global_state** per il parmod ed una **local_state** per le singole operation.

La definizione di uno stato "locale" alle operation si rende necessaria, in quanto ognuna di esse può implementare un algoritmo completamente differente, ed obbligare il programmatore ad utilizzare le stesse variabili per tutte ne abbasserebbe enormemente l'espressività.

Infatti si potrebbe pensare di definire solamente uno stato globale, su cui tutte le operation possono lavorare; non abbiamo problemi di concorrenza, in quanto per ogni modulo rimane attiva una sola operation alla volta. Dobbiamo però pensare che in una applicazione parallela lo stato è partizionato o replicato, e due operation diverse possono avere strutture completamente differenti: il partizionamento dei dati necessario per una operation potrebbe

non essere definibile sull'altra. Inoltre abbiamo parlato spesso della possibilità di realizzare operation "leggere" nell'occupazione di memoria. Definire uno stato con le variabili utilizzate da tutte le operation annullerebbe completamente questa possibilità.

Queste considerazioni suggeriscono la necessità di uno stato proprio della operation, e hanno portato alla definizione dello stato "locale". Questo però non elimina la necessità di uno spazio "condiviso" tra le varie operation, o meglio di informazioni *persistenti* ad un cambio di comportamento. Abbiamo quindi inserito anche uno stato globale, condiviso tra tutte le operation. In questo stato abbiamo normali variabili, né partizionate né replicate.

Come abbiamo già accennato per questo stato non sono previsti accessi concorrenti tra differenti operation, in quanto in ogni istante ne sarà in esecuzione una sola. Questo però non risolve tutti i problemi: il modulo è al suo interno parallelo, perciò sono comunque necessarie delle politiche di accesso allo stato. Per il momento lasciamo questo compito al programmatore; vedremo in lavori futuri, se necessario, tecniche migliori per gestire queste informazioni.

7.4 Riconfigurazioni di un parmod

Passiamo ora alla parte veramente innovativa di ASSISTANT, quella che permette di definire un comportamento reattivo agli eventi per ottenere moduli adattivi e context-aware. Questa parte modella lo strato BLUE dell'applicazione, ed è descritta nella sezione **on_event** delle singole operation.

7.4.1 La sezione On Event

Questa sezione definisce il comportamento del modulo al verificarsi degli eventi di contesto. Per facilitare la programmazione, abbiamo deciso di spostare la gestione degli eventi dal modulo intero alle singole operation. In questo modo lo sviluppatore può decidere la reazione del modulo agli eventi in base alla operation attiva.

All'interno della sezione *on event* il programmatore può definire le azioni da intraprendere al verificarsi di differenti *eventi* o *combinazioni di eventi*. Gli eventi arrivano ad un modulo tramite degli stream. Le azioni da eseguire alla loro ricezione possono perciò essere definite esattamente come per la input section della operation: tramite una guardia con priorità.

Abbiamo quindi una sintassi come quella riportata in 7.9: possiamo associare a condizioni anche complesse di eventi un particolare comportamento.

```
on_event{
  <prority_1>: <event_combination_1> do { ... }
  ...
  <prority_N>: <event combination_N> do { ... }
}
```

Listato 7.9: Sintassi del costrutto *on_event*

Se sono verificate più condizioni contemporaneamente viene scelta quella con priorità più alta e, a parità di priorità, non deterministicamente.

Combinazioni di eventi

Le combinazioni di eventi permesse sono più complesse di quelle della input section, in modo da garantire una maggiore flessibilità per questo costrutto. In particolare possiamo definire condizioni non solo sulla presenza di un evento ma anche sul suo valore, che può essere confrontato con delle costanti, delle variabili del modulo o delle funzioni primitive fornite dal modello. Oltre a ciò possiamo anche definire condizioni booleane tra eventi, tramite le parole chiave **AND**, **OR** e **NOT**. Sono perciò permesse combinazioni come:

```
on_event{
  0: mainNetOff AND INLHigh do { ... }
  0: serviceTime < 10 do { ... }
  0: mainNetOff OR (NOT INLHigh) do { ... }
}
```

In questa sezione si possono utilizzare tutti gli stream non precedentemente utilizzati nella "input section", per evitare "conflitti" tra le due sezioni. È comunque in linea col concetto di dati-eventi: la suddivisione tra stream di dati e stream di eventi, pur non presente al livello del grafo di applicazione ASSISTANT, emerge all'interno del singolo parmod, che può utilizzare un particolare stream *o* per i dati *o* per gli eventi.

Le variabili di stato del modulo devono essere identificate in modo univoco; perciò non possono essere quelle replicate dello stato locale alla operation; per il momento abbiamo deciso che si può accedere in lettura e scrittura *solo* alle variabili del "global_state".

Un'altra questione importante riguarda l'operatore **NOT** su un evento; questo rappresenta la *non presenza* dell'evento, che non va confusa con un evento presente ma con valore *false*. Il primo si tratta con l'operatore **NOT**, il secondo con una condizione sul valore dell'evento, del tipo (mainNetOff == **false**).

Reazione ad un evento

La reazione più comune ad un evento è una riconfigurazione, che porta ad una variazione del comportamento del modulo. Le due modalità di riconfigurazione viste fino a questo momento sono esprimibili tramite due costrutti differenti:

- **parallelism** *grado di parallelismo*: permette di richiedere una riconfigurazione non funzionale tramite il cambio del grado di parallelismo;
- **switch_to** *nome operation*: permette di richiedere una riconfigurazione funzionale.

A cui si aggiungono la loro composizione ed il meccanismo per generare nuovi eventi:

- **switch_to** *nome operation* **parallelism** *grado di parallelismo*: permette di richiedere una riconfigurazione funzionale e, contemporaneamente, modificare il grado di parallelismo;
- **notify** *stream valore*: permette di generare un nuovo evento, con un valore determinato.

Chiaramente nella maggior parte dei casi prima dell'invocazione di questi costrutti bisogna eseguire qualche calcolo, verificare alcune condizione, etc. Per questo motivo la reazione ad un evento può essere descritta con linguaggio di programmazione standard (C++) arricchito con i costrutti visti. Questo permette, ad esempio, di calcolare il grado di parallelismo adeguato prima di invocare il costrutto **parallelism**. Possiamo però descrivere anche comportamenti avanzati, perché in questo codice si può utilizzare:

- alcune funzioni *built-in*, come i modelli di costo della particolare forma parallela e la conoscenza dell'attuale grado di parallelismo;
- i valori degli eventi che hanno attivato la guardia;
- lo stato interno del modulo, nella forma del "global_state".

Inoltre non è necessario riconfigurare veramente il modulo: alla ricezione di un evento possiamo semplicemente modificare lo stato globale, richiedere la riconfigurazione solo sotto determinate condizioni oppure utilizzare riconfigurazioni differenti in base allo stato del modulo.

Le funzioni built-in Elenchiamo velocemente le funzioni che abbiamo pensato per il momento; questo insieme è ovviamente destinato a variare nel tempo: ci potremmo accorgere dell'inutilità di alcune funzioni e della necessità di altre.

- ***double*** *estimatedServiceTime(****string*** *operation,* ***int*** *parallelismDegree)*: fornisce una stima del tempo di servizio dell'attuale operation, stimando il tempo di calcolo del codice sequenziale con misurazioni sulle precedenti esecuzioni oppure tramite una analisi euristica del codice.

- ***double*** *estimatedLatency(****string*** *operation,* ***int*** *parallelismDegree)*: analoga alla precedente, ma restituisce la latenza invece che il tempo di servizio.

- ***int*** *estimatedParallelismForST(****string*** *operation,* ***double*** *serviceTime)*
  int *estimatedParallelismForLat(****string*** *operation,* ***double*** *serviceTime)*: rappresentano le funzioni "inverse" alle precedenti, restituendo il grado di parallelismo minimo necessario per ottenere un tempo di servizio o una latenza dati;

- ***double*** *currentServiceTime()*
  double *currentLatency()*: restituiscono, rispettivamente, il tempo di servizio e la latenza dell'operation;

- ***int*** *currentParallelismDegree()*: restituisce il grado di parallelismo corrente.

La stima che utilizziamo sul tempo di calcolo della parte sequenziale viene calcolata normalmente tramite una analisi delle esecuzioni pregresse dello stesso codice. Ovviamente questa stima non è calcolabile alla prima esecuzione della operation; in questo caso verrà sostituita da una stima ottenuta attraverso una analisi euristica del codice sequenziale. Ovviamente questa può essere molto distante dai risultati reali, ma è comunque meglio di niente.

Le funzioni "estimatedParallelismFor..." sono, al contrario di quello che si potrebbe pensare intuitivamente, molto utili: il comportamento di un programma parallelo non è sempre lineare, e possiamo avere risultati inaspettati, come un *aumento* del tempo di servizio con l'aggiunta di nodi, oppure "picchi" destinati poi a riscendere grazie alla modifica dell'implementazione fornita dal supporto. In questo modo racchiudiamo tutte queste problematiche all'interno delle funzioni fornite dall'ambiente di sviluppo. Potremmo anche trovarci nella situazione di non poter garantire il tempo richiesto, ed in questo caso le funzioni restituiranno il codice di errore -1.

Consumazione degli eventi

Prima di continuare la trattazione vogliamo soffermarci sul concetto molto particolare della "consumazione" di un evento. Si tratta di una parte fondamentale della **on_event**, in quanto un utilizzo non corretto può portare ad un comportamento del modulo molto diverso da quello aspettato ad una prima analisi.

Spiegando la semantica del costrutto abbiamo detto che si tratta di una guardia sugli stream che trasportano eventi. La semantica di una guardia (ripresa dal linguaggio ECSP[18]) si adatta al trattamento dei dati ma potrebbe risultare "anomala" in caso di eventi. I dati sugli stream non vengono mai persi, e rimangono "in attesa" di essere trattati. Allo stesso modo un evento arrivato su uno stream rimane in attesa di essere gestito finché non viene attivata una guardia che lo utilizzi.

Questo comportamento potrebbe essere, in alcuni casi, non voluto sugli eventi: finché un evento non viene trattato, quelli inviati sullo stesso stream e generati successivamente non possono essere analizzati. Questo può creare grossi problemi se utilizzato insieme alle guardie con condizioni sul valore dell'evento: la condizione indicata potrebbe non verificarsi mai, ed in quel caso "disabilitiamo" la ricezione di quel tipo di eventi, perché non passeremo mai all'ascolto dei successivi.

Facciamo un esempio con il seguente codice, preso da un esempio realistico. Si consideri lo stream **TS_MAX** da cui si riceve una richiesta sul tempo di servizio che il modulo dovrebbe rispettare. In un primo momento si potrebbe pensare di utilizzare una guardia sul valore ricevuto, del tipo:

```
0: (currentServiceTime() > TS_MAX) do { ... }
```

Dove si attiva la guardia quando il tempo di servizio corrente supera quello richiesto. In questa situazione, però, il TS-MAX sullo stream potrebbe essere sensibilmente superiore a quello del modulo; in questo caso la guardia non si attiverà neanche successivamente. A questo punto, se viene inviata una nuova richiesta di TS-MAX, anche minore dell'attuale tempo di servizio, la guardia non verrà comunque attivata, perché il primo valore ricevuto rimane in attesa di essere consumato. In questo caso si rende necessaria una differente scrittura:

```
0: (TS_MAX) do {
   if (currentServiceTime() > TS_MAX){ ... }
}
```

dove consumiamo subito l'evento, appena ricevuto. Una volta attivati, controlliamo se è necessaria una riconfigurazione. In questo modo l'evento viene comunque rimosso dallo stream, e siamo pronti per riceverne uno nuovo.

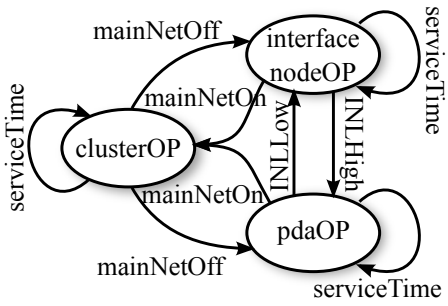


Figura 7.3: Event-Operation Graph del Tridiagonal Solver

Con questo termina la trattazione della sezione "on_event". Passiamo ad una applicazione pratica, ovvero il problema di descrivere queste sezioni per il modulo "Tridiagonal Solver".

7.4.2 Le riconfigurazioni del Tridiagonal Solver

Vediamo ora in dettaglio degli esempi reali per la sezione **on_event**, scrivendo quelle delle tre operation del modulo "Tridiagonal Solver". Descriveremo, a piccoli passi, l'**event-operation graph** presentato alla fine del capitolo precedente e che riportiamo nella figura 7.3.

clusterOP

Iniziamo con la descrizione del costrutto on_event per la clusterOP. Possiamo ricevere degli eventi di tipo "serviceTime", che richiedono di rispettare un determinato tempo di servizio. Alla ricezione di questo evento vogliamo riconfigurarci per rispettare la richiesta. Potremmo quindi definire una guardia del tipo:

```
0: (serviceTime) do {
  if (currentServiceTime() > serviceTime){
    int newpar =
        estimatedParallelismForST("clusterOP",serviceTime);
    parallelism newpar;
  }
```

In questo codice, però, non consideriamo la possibilità di liberare risorse se abbiamo un tempo di servizio minore di quello richiesto. Una implementazione più intelligente potrebbe essere la seguente:

```
0: (serviceTime) do {
  int newpar =
      estimatedParallelismForST("clusterOP",serviceTime);
```

```
    if (currentServiceTime() > serviceTime)
      parallelism newpar;
    else if (abs(newpar-currentParallelismDegree()) >
      THRESHOLD)
      parallelism newpar;
  }
```

Così descriviamo anche una riconfigurazione se il nostro grado di parallelismo è maggiore di quello necessario. Ovviamente questa verrà eseguita solo se la differenza tra il grado corrente e quello ideale è abbastanza elevata da giustificare l'overhead della riconfigurazione.

Passiamo ora a descrivere le operazioni per la ricezione dell'evento "mainNetOff", che rappresenta una caduta della rete tra cluster e nodo di interfaccia. In questo caso il grafo definisce due possibili operation switch, verso "interfacenodeOP" e "pdaOP", in base all'attuale utilizzo delle risorse del nodo di interfaccia.

In generale non possiamo conoscere questo valore, ma nel nostro caso abbiamo definito appositamente due eventi per rappresentarlo. Nel grafo eventi-operation non abbiamo modellato la cosa, ma in realtà possiamo ricevere gli eventi "INLLow" e "INLHigh" anche durante l'esecuzione su cluster, perché questi sono generati da una interfaccia primitiva esterna al modulo. Seguendo le considerazioni di prima dovremmo in qualche modo "consumare" ugualmente gli eventi; visto che li consumiamo, possiamo salvarci in una variabile il contenuto dell'ultimo evento (che rappresenta la percentuale di utilizzazione del nodo) per poterlo utilizzare successivamente. Inseriamo perciò una variabile nello stato globale:

```
global_state{
  double interfaceNodeLevel = 0;
}
```

e due guardie per gestire gli eventi:

```
  0: (INLLow) do {
    interfaceNodeLevel = INLLow;
  }
  0: (INLHigh) do {
    interfaceNodeLevel = INLHigh;
  }
```

Abbiamo ora tutto il necessario per definire la guardia per l'evento "mainNetOff":

```
  0: (mainNetOff) do {
    if(interfaceNodeLevel <= 30){
      systemSize = 131071;
```

```
      switch_to "interfacenodeOP";
    }else{
      systemSize = 32767;
      switch_to "pdaOP";
    }
  }
```

In questo caso non specifichiamo il nuovo grado di parallelismo, che verrà definito automaticamente dal sistema; se il tempo di servizio ottenuto non è in linea con le richieste, il Viewer potrà sempre inviare un nuovo evento ServiceTime. Abbiamo aggiunto anche l'assegnamento ad una variabile, systemSize. Questa ci serve per definire la dimensione del sistema che la nuova operation dovrà eseguire (ricordiamo infatti che nodeinterfaceOP e pdaOP possono eseguire il calcolo su dimensioni di sistemi differenti). Anche questa variabile sarà allocata nello stato globale.

Raggruppiamo ora tutte le guardie descritte nel listato 7.10, per formare la on_event della clusterOP. Aggiustiamo anche le priorità per gestire eventi concorrenti nel giusto ordine: a priorità massima mettiamo l'evento mainNetOff, mentre gli altri possiamo inserirli alla stessa priorità in quanto indipendenti.

interfacenodeOP

Passiamo ora ad analizzare l'operation "interfacenodeOP". La trattazione degli eventi "mainNetOn" e "INLHigh" si ottiene semplicemente con uno switch di operation:

```
0: (INLHigh) do {
  interfaceNodeLevel = INLHigh;
  systemSize = 32767;
  switch_to "pdaOP";
}
1: (mainNetOn) do {
  switch_to "clusterOP";
}
```

Nel primo caso salviamo il carico attuale per mantenere la variabile globale aggiornata ed impostiamo la dimensione di default per l'operation su pda; inoltre trattiamo con priorità maggiore l'evento per la nuova presenza della rete in quanto ci permette di tornare subito al cluster. Gli eventi "mainNetOff" e "INLLow" possono non essere "ascoltati", in quanto per definizione questa operation viene utilizzata solo se la rete è non utilizzabile e il carico sul nodo basso, mentre la ricezione degli eventi complementari ai due provoca immediatamente uno switch di operation. Sono quindi eventi che non dovrebbero arrivare durante l'esecuzione della "interfacenodeOP".

```
clusterOP{
  ...
  on_event{
    0: serviceTime do {
      int newpar =
          estimatedParallelismForST("clusterOP",serviceTime);
      if (currentServiceTime() > serviceTime){
        parallelism newpar;
      }
      else if (abs(newpar-currentParallelismDegree()) >
          THRESHOLD){
        parallelism newpar;
      }
    }
    0: INLLow do {
      interfaceNodeLevel = INLLow;
    }
    0: INLHigh do {
      interfaceNodeLevel = INLHigh;
    }
    1: mainNetOff do {
      if(interfaceNodeLevel <= 30){
        systemSize = 131071;
        switch_to "interfacenodeOP";
      }else{
        systemSize = 32767;
        switch_to "pdaOP";
      }
    }
  }
}
```

Listato 7.10: Sezione on_event per la clusterOP

Vediamo ora il trattamento dell'evento serviceTime. Questo è più complesso dei precedenti, perché oltre al cambio del grado di parallelismo possiamo modificare la dimensione dei sistemi risolti. Per come la abbiamo definita, non si tratta di una "riconfigurazione" vera e propria, in quanto non passiamo ad un'altra operation né effettuiamo una riconfigurazione funzionale.

Semplicemente modificando una variabile dello stato globale il codice sequenziale della operation genera sistemi di dimensioni differenti, modificando in modo sostanziale la precisione ed il tempo di calcolo. Cerchiamo di capirne il funzionamento: in questo caso la operation implementa un data-parallel; nella input section riceve i valori necessari per generare le matrici e, a questo punto, accederà al valore presente nello stato globale per definire la dimensione delle stesse, che una volta generate saranno scatterizzate ai VP.

Il valore nello stato globale viene modificato solo all'interno della "on_event" e letto nella input section; in questa sequenza non abbiamo problemi di accesso concorrente: le attivazioni del modulo precedenti alla modifica utilizzeranno il vecchio valore, quelle successive il nuovo. Per il comportamento del modulo, quindi, sappiamo che tutte le attivazioni successive alla modifica utilizzeranno la nuova dimensione del sistema e saranno perciò più veloci.

Questa modifica non va però effettuata ad ogni evento "serviceTime", ma solo se le riconfigurazioni non funzionali non permettono di rispettare il tempo di servizio richiesto. Abbiamo quindi la seguente guardia:

```
0: (serviceTime) do {
  int newpar =
      estimatedParallelismForST("interfacenodeOP",serviceTime);
  if(newpar==-1){
    systemSize = (systemSize-1)/2;
  } else if(newpar==1){
    systemSize = ((systemSize+1)*2)-1;
  } else{
    if (currentServiceTime() > serviceTime)
      parallelism newpar;
    else if (abs(newpar-currentParallelismDegree()) >
        THRESHOLD)
      parallelism newpar;
  }
}
```

Per prima cosa ci calcoliamo il grado di parallelismo necessario per ottenere il tempo di servizio richiesto. Il risultato, per definizione della funzione, sarà calcolato sui tempi delle esecuzioni precedenti, e quindi con la dimensione dei sistemi attuale. Se il risultato della funzione è -1 sappiamo di non avere abbastanza nodi per ottenere il tempo richiesto; dimezziamo perciò la dimensione dei sistemi. Al contrario, se il grado di parallelismo necessario è

```
interfacenodeOP{
  ...
  on_event{
    0: (serviceTime) do {
      int newpar = estimatedParallelismForST(
          "interfacenodeOP",serviceTime);
      if(newpar==-1){
        systemSize = (systemSize-1)/2;
      } else if(newpar==1){
        systemSize = ((systemSize+1)*2)-1;
      } else{
        if (currentServiceTime() > serviceTime)
          parallelism newpar;
        else if (abs(newpar-currentParallelismDegree()) >
            THRESHOLD)
          parallelism newpar;
      }
    }
    1: (INLHigh) do {
      interfaceNodeLevel = INLHigh;
      systemSize = 32767;
      switch_to "pdaOP";
    }
    2: (mainNetOn) do {
      switch_to "clusterOP";
    }
  }
}
```

Listato 7.11: Sezione on_event per la interfacenodeOP

1, possiamo permetterci di aumentare le dimensioni dei sistemi ed ottenere comunque tempi di servizio ragionevoli.

Se non ci troviamo in questi due casi estremi, invece, modifichiamo semplicemente il grado di parallelismo, con la stessa tecnica vista per la operation precedente.

Nel listato 7.11 troviamo la on_event completa.

pdaOP

Terminiamo la trattazione con la on_event dell'ultima operation rimasta. Questa non presenta nuovi concetti rispetto alle precedenti, perciò riportiamo direttamente l'intera sezione nel listato 7.12.

Anche in questo caso l'accesso in scrittura da parte della "on_event" sulla variabile *systemSize* non costituisce un problema; questa volta abbiamo

```
pdaOP{
  ...
  on_event{
    0: (serviceTime) do {
      int newpar =
          estimatedParallelismForST("pdaOP",serviceTime);
      if(newpar==-1){
        systemSize = (systemSize-1)/2;
      } else if(newpar==1){
        systemSize = ((systemSize+1)*2)-1;
      } else{
        if (currentServiceTime() > serviceTime)
          parallelism newpar;
        else if (abs(newpar-currentParallelismDegree()) >
            THRESHOLD)
          parallelism newpar;
      }
    }
    1: (INLLow) do {
      interfaceNodeLevel = INLLow;
      systemSize = 131071;
      switch_to "interfacenodeOP";
    }
    2: (mainNetOn) do {
      switch_to "clusterOP";
    }
  }
}
```

Listato 7.12: Sezione on_event per la pdaOP

un farm, perciò la dimensione del sistema viene letta in momenti differenti dai singoli VP. Questo provoca dei momenti di "disallineamento" tra i VP attualmente in esecuzione, dove alcuni hanno letto il nuovo valore, altri il vecchio. Questo però non costituisce comunque un problema, in quanto stiamo parlando di un farm, dove i VP sono tra loro assolutamente indipendenti.

7.5 Conclusioni

In questo capitolo abbiamo presentato la sintassi completa di ASSISTANT, descrivendo in modo abbastanza accurato tutte le parti modificate rispetto a quella iniziale di ASSIST.

Abbiamo finalmente chiarito le modalità per descrivere il comportamento adattivo e context aware di un parmod, attraverso la sezione "on_event" che

modella la parte BLUE dell'applicazione.

Siamo perciò riusciti a definire in modo completo (e con il modello di ASSISTANT) l'applicazione di gestione delle emergenze descritta nel capitolo 6.

Con la definizione del linguaggio di ASSISTANT e di una applicazione completa per tale modello termina la tesi.

Nel prossimo capitolo analizzeremo in modo sommario alcuni dei concetti ancora in fase di definizione e perciò non ancora ben formalizzati, ma molto importanti per raggiungere una definizione abbastanza precisa di ASSISTANT per poter iniziare una prima implementazione degli strumenti di supporto al modello.

Capitolo 8

Conclusioni e Sviluppi Futuri

Con questa tesi abbiamo studiato i modelli presenti in letteratura per descrivere applicazioni pervasive con requisiti ad alte prestazioni. Al momento non esistono modelli specifici, perciò ci siamo orientati verso gli strumenti utilizzati nel "Pervasive Computing" e nel "High Performance Computing". Nessuno di questi offriva le caratteristiche richieste, ed abbiamo quindi continuato verso la definizione di un nuovo modello studiato appositamente.

Le basi su cui è stato modellato ASSISTANT sono quelle delle forme di parallelismo, che ci permettono di rappresentare in modo semplice ed efficiente applicazioni parallele anche complesse. Su queste abbiamo inserito meccanismi per gestire le problematiche di adattività e context awareness tipiche dei programmi pervasivi, riuscendo perciò a fornire un ambiente per la descrizione di applicazioni pervasive con requisiti di alte prestazioni.

Per la definizione non siamo partiti da zero, ma piuttosto cercando di riutilizzare le nostre conoscenze sugli ambienti di programmazione paralleli strutturati. Abbiamo quindi esteso il modello offerto da ASSIST, un linguaggio per applicazioni parallele ad alte prestazioni sviluppato dal nostro gruppo di ricerca, e considerato più potente e flessibile dei classici ambienti a "skeleton".

Su questo modello abbiamo inserito dei meccanismi per definire moduli paralleli con comportamenti multipli che, affiancati ad un sistema di gestione di eventi, permettono di esprimere adattività e context awareness per le singole componenti dell'applicazione.

Per dimostrare l'applicabilità e la potenza del linguaggio introdotto abbiamo mostrato un esempio di applicazione pervasiva ad alte prestazioni, modellata tramite i concetti introdotti e descritta utilizzando la sintassi di ASSISTANT.

In questa tesi abbiamo presentato solamente il *modello* di ASSISTANT, senza però parlare di una sua eventuale implementazione. Gli esempi portati in questa tesi e nei precedenti articoli sono stati prodotti senza l'ausilio di tool automatici, ma scrivendo direttamente il codice che dovrebbe essere prodotto da un eventuale compilatore ASSISTANT.

Lo sviluppo dell'intero ambiente di sviluppo non è ancora iniziato (se non con piccoli test-bed di prova), perché in realtà ci sono ancora molti aspetti importanti da chiarire prima di passare alla scrittura del codice. Tra gli sviluppi futuri riteniamo quindi fondamentale lo studio di questi aspetti, per poter arrivare ad una prima implementazione del compilatore di ASSISTANT e degli strumenti di supporto necessari.

In queste ultime pagine riportiamo i principali ambiti su cui vogliamo continuare la ricerca, suddivisi tra obiettivi a breve/medio termine e obiettivi a lungo termine.

8.1 Obiettivi a breve e medio termine

In questa sezione parliamo dei principali aspetti su cui stiamo procedendo in questo momento, e che consideriamo fondamentali per ogni studio successivo. Alcuni di questi sono nati, o comunque stati analizzati almeno in parte, durante la redazione di questa tesi. Non ne abbiamo parlato per motivi di spazio e di tempo, ma li vogliamo comunque accennare velocemente in questa sezione.

Modellazione delle interazioni tra RED e BLUE dei singoli moduli

Si tratta del problema su cui stiamo concentrando le nostre forze, e che ci permetterà di sbloccare qualche prima forma di implementazione (anche prototipale e non completa) degli strumenti di ASSISTANT.

Non abbiamo sottolineato in modo significativo questo aspetto durante la tesi, ma in realtà il problema di come far interagire la "sezione on_event" col resto del parmod non è banale. Per comodità in questa trattazione separiamo il parmod tra parte "computazionale" e "logica di riconfigurazione", riprendendo quindi la suddivisione tra RED e BLUE.

In letteratura l'interazione tra le due parti viene spesso trattata come una interazione completamente asincrona: la parte BLUE (nel nostro caso costituita dalla "on_event") può decidere di effettuare una riconfigurazione. A questo punto comunica questa volontà al RED che, quando possibile, effettuerà materialmente la riconfigurazione. Con questa tipologia di interazioni

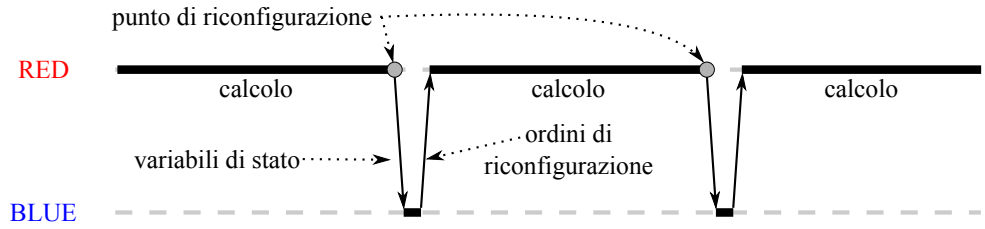


Figura 8.1: Modalità di interazione "sincrona" tra RED e BLUE

tra la decisione e l'effettiva riconfigurazione può passare un tempo anche molto lungo e non sempre determinabile.

Con ASSISTANT vorremmo risolvere (nel limite del possibile) questo problema, e proponiamo un approccio concettualmente molto differente, in cui le due parti del parmod collaborano attivamente nella fase di riconfigurazione. Innanzitutto osserviamo come, in realtà, l'osservazione degli eventi potrebbe non avvenire in modo continuativo, ma solo in determinati momenti.

Finché la parte RED non raggiunge un punto in cui è disposta ad effettuare una riconfigurazione, l'osservazione degli eventi è superflua, e può anzi portare a situazioni particolari, in quanto differenti richieste di riconfigurazione, seppur generate in tempi differenti, si possono trovare contemporaneamente in coda al RED, che dovrebbe decidere se effettuarle entrambe o solo alcune.

Parallelamente a questo riprendiamo un altro concetto importante: per la gestione degli eventi abbiamo spesso bisogno anche di variabili di stato del modulo, che si trovano nativamente nella parte RED.

Questi aspetti ci portano a modellare la riconfigurazione con un protocollo sincrono, come quello illustrato in figura 8.1. Intuitivamente ci possiamo rendere conto che, per i motivi appena descritti, pur essendo una modalità sincrona offre prestazioni allineate a quelle della modalità asincrona vista prima.

Questo schema ricorda molto un'altra modalità di interazione, formalizzata da Gerace per la realizzazione di unità di elaborazione a partire da un microprogramma [49]. Il modello di Gerace prevede proprio la separazione di tali unità in due reti sequenziali distinte, chiamate *Parte Operativa* e *Parte Controllo*.

Possiamo trovare una forte analogia, considerando il RED la parte operativa, il BLUE la parte controllo e i punti in cui il RED è disposto a riconfigurarsi come "segnale di clock".

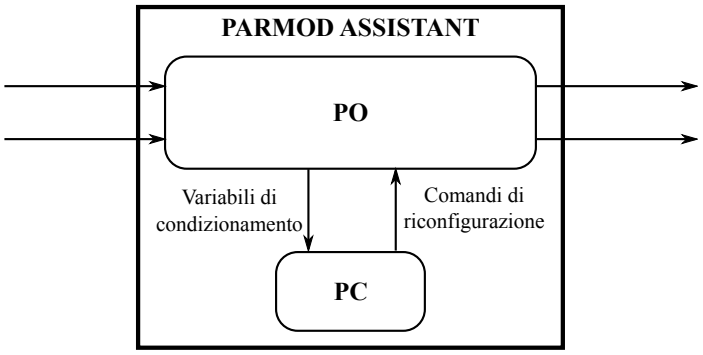


Figura 8.2: Modulo rappresentato con la modellazione PC-PO

La nostra proposta è quindi quella di modellare le interazioni tra i due strati del parmod in stile PC-PO, come illustrato in figura 8.2. Ovviamente il lavoro da fare è ancora molto, soprattutto nell'ottica di non introdurre overhead inutili. Da questo punto di vista, infatti, in realtà la nostra PO è composta da un insieme di entità separate ed indipendenti, che possono quindi raggiungere il punto di "riconfigurazione" in modo indipendente tra di loro. Con la modellazione appena fornita, intuitivamente, queste si dovrebbero "fermare", per aspettare di raggiungere congiuntamente il punto e solo allora inviare le variabili di condizionamento alla PC. Questo introduce un overhead notevole, che vorremmo in qualche modo eliminare. Abbiamo già qualche idea su come superare questo problema, ma le stiamo ancora formalizzando.

Dobbiamo comunque sottolineare che non siamo ancora certi che questa modellazione si adatti perfettamente al problema. Ne siamo fortemente convinti, ma non siamo ancora riusciti a dimostralo. Perciò è possibile che nel tempo questa possibilità venga abbandonata verso altre ritenute più valide.

Gestione della dinamicità dei nodi

Un secondo punto su cui focalizzare i nostri studi è quello sulla gestione efficace dell'alta dinamicità dei nodi, già accennato durante la tesi.

In un'ottica adattiva si potrebbero considerare le scomparse dei nodi come dei semplici eventi che, al pari di tutti gli altri, necessitano di essere gestiti. In questo caso dobbiamo trattare come interazioni RED-BLUE tutti i problemi legati alla fault tolerance: checkpointing, ripristino dello stato, allocazione di nuovi nodi, etc.

La loro trattazione risulta però scollegata dalla "on_event", che modella solo quegli eventi che servono al programmatore per definire il comporta-

mento adattivo e context aware. Allo stesso tempo complicherebbe in modo spropositato la logica del BLUE, che dovrebbe cooperare in modo molto più stretto e frequente con la RED.

Per dominare questa complessità proponiamo di trattare la dinamicità in modo profondamente diverso, inserendo un supporto RED che sia automaticamente *fault tolerant*. In questo modo la logica di riconfigurazione si occupa solo degli eventi definiti dall'utente all'interno del modulo, e diventa quindi molto più semplice da descrivere.

Ovviamente anche questa tecnica ha degli svantaggi, basta pensare al "Tridiagonal Solver" descritto nei capitoli precedenti: in caso di interruzione della connessione col cluster abbiamo un'azione combinata di "supporto fault tolerant" e "supporto alle riconfigurazioni". In questo caso una gestione unitaria, da parte dello strato BLUE, potrebbe portare ad un trattamento più veloce della situazione, rispetto ad un supporto che si occupa prima di ripristinare come può la computazione sui nodi rimasti e, solo successivamente, di gestire l'evento che porterà ad una riconfigurazione funzionale.

Il lavoro in questo campo deve perciò tendere alla definizione di un comportamento indipendente tra le due parti ma il più possibile efficiente in tutti i casi.

Ottimizzazione dell'operation switch

Dal nostro punto di vista un ulteriore punto critico da studiare a fondo è la definizione di meccanismi di ottimizzazione per la parte di "operation switch".

Quando, nel capitolo 5, abbiamo effettuato la distinzione tra *riconfigurazioni funzionali* e *riconfigurazioni non funzionali* la motivazione fornita era, principalmente, una questione di prestazioni.

Con le riconfigurazioni non funzionali, infatti, possiamo definire delle tecniche per "sospendere" la computazione attuale e "riprenderla" esattamente dallo stesso punto sulla nuova configurazione, mantenendo una semantica del modulo corretta.

Sulle riconfigurazioni funzionali questo non è possibile, almeno non in generale: essendo computazioni differenti, non possiamo riprendere l'esecuzione della nuova con lo stato della precedente. Una semantica corretta è ottenibile lasciando gestire ogni elemento dello stream da una sola operation. In questa ottica, al momento di una riconfigurazione, possiamo fondamentalmente scegliere tra due possibilità: lasciare l'esecuzione sul dato attuale con il vecchio comportamento e gestire i successivi col nuovo, oppure annullare l'esecuzione attuale e rieseguirla completamente col nuovo comportamento.

In ogni caso abbiamo una possibile perdita notevole di prestazione, che potrebbe essere in alcuni casi immotivata: crediamo infatti che esistano algoritmi "compatibili", che possono sfruttare uno i dati dell'altro per continuare la computazione.

Per questi algoritmi potremmo utilizzare la tecnica introdotta con le riconfigurazioni non funzionali per sfruttare parte del lavoro eseguito sull'elemento corrente prima della riconfigurazione. Con buona probabilità lo stato tra le due operation non sarebbe comunque identico; ci immaginiamo perciò delle "trasformazioni", per adattare lo stato di un comportamento a quello di un altro e così non perdere tutta o parte della computazione già eseguita.

Un possibile esempio di algoritmi "compatibili" che stiamo studiando ultimamente si ricava sempre dai problemi di calcolo numerico. Per il problema di fattorizzare una matrice nella forma QR (ovvero come prodotto di una matrice unitaria ed una triangolare superiore) esistono più algoritmi, tra cui:

- fattorizzazione mediante trasformazioni di HouseHolder;
- fattorizzazione mediante rotazioni di Givens.

Un primo studio ci ha portato a dimostrare che i due algoritmi sequenziali sono tra loro "compatibili". Entrambi si basano sul concetto di trasformare ad ogni passo una colonna della matrice iniziale per raggiungere la forma triangolare; in questo caso le colonne annullate da un algoritmo possono essere utilizzate interamente dall'altro, che può procedere all'annullamento delle sole successive ed ottenere, alla fine, sempre una fattorizzazione di tipo QR.

Non abbiamo ancora dimostrato la compatibilità anche nel caso parallelo, ma ci proponiamo sicuramente di continuare su questa strada.

Repository di codice e dati

Un altro aspetto che necessita di un ulteriore studio è la realizzazione di un repository di codice e dati efficace anche su piattaforme pervasive.

In questa tipologia di sistemi si rende necessario un approccio "dinamico" anche nella distribuzione dei dati e del codice da eseguire: per ovvi motivi non possiamo infatti assumere di avere il codice ed i dati necessari alle varie operation già su tutti i nodi "candidati" per una possibile riconfigurazione.

Ogni dispositivo conterrà normalmente una piccola parte del supporto di ASSISTANT, coi compiti di avvisare il resto del sistema (soprattutto il motore di resource discovery) della sua presenza e ad abilitare il dispositivo all'eventuale esecuzione di una parte dell'applicazione.

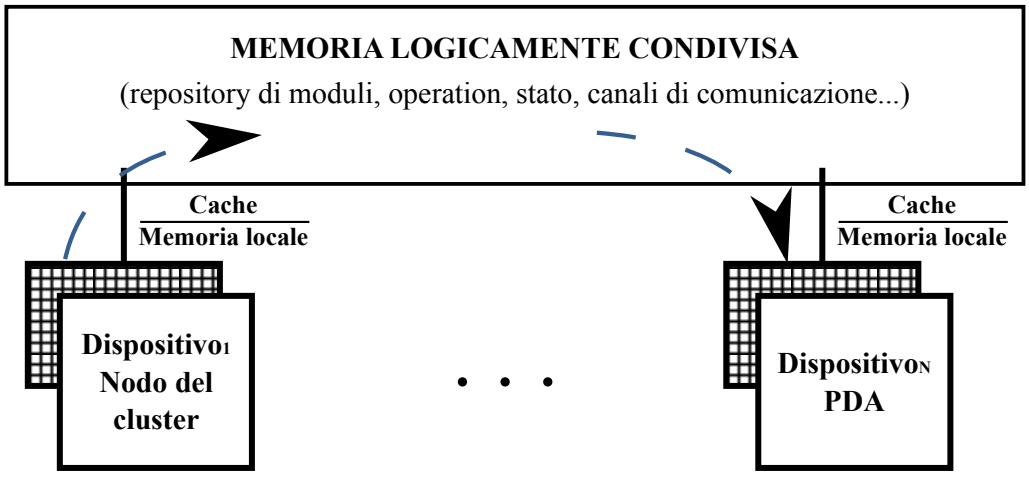


Figura 8.3: Modellazione dell'ambiente di esecuzione tramite memoria logicamente condivisa e processori anonimi

Se il sistema decide di spostare l'esecuzione su un nuovo nodo non ancora utilizzato, i programmi necessari ed i dati della computazione dovranno essere portati sul nodo scelto. Per far questo ci immaginiamo la presenza di uno spazio logicamente condiviso tra tutti i nodi, che contenga indistintamente tutte le informazioni richieste, tra cui programmi e dati. Ogni nodo, tramite l'accesso a questo spazio, può reperire tutte le informazioni necessarie e spostarle sulla propria memoria.

La presenza di una memoria condivisa che contenga tutte le informazioni, e di nodi generici che possono eseguire in modo indifferente diverse parti dell'applicazione trova una certa analogia col modello a *processori anonimi con memoria condivisa* studiato per le architetture parallele. Possiamo infatti vedere i nodi come dei "processori generici" su cui vengono eseguite, in base ad una politica di scheduling decisa dai manager dei vari moduli, parti dell'applicazione. La memoria logicamente condivisa contiene tutti i dati dell'intera applicazione, ed al momento dell'esecuzione ogni nodo scarica tutte le informazioni necessarie all'esecuzione nella sua memoria locale (che corrisponde logicamente ad una cache). Questa modellazione, illustrata in figura 8.3, potrebbe offrirci molti vantaggi nella trattazione di alcuni problemi, come ad esempio la gestione della consistenza tra memoria locale e quella logicamente condivisa.

Ovviamente nella realizzazione di questo spazio condiviso ritroviamo i problemi di fault-tolerance già affrontati in tutta la tesi. Pensiamo di risolvere il problema tramite tecniche di replicazione dei dati, in modo da mantenere più copie degli stessi e permettere ai nodi che implementano la memoria di

disconnettersi improvvisamente. In questo ambito si potrebbe anche studiare in modo più approfondito la tecnica presentata in [2] e nel mio lavoro di tirocinio ([26]) per implementare una memoria logicamente condivisa fault-tolerant.

8.2 Obiettivo a lungo termine

Ovviamente l'obiettivo a lungo termine di questa tesi, ed in generale dell'intero gruppo di ricerca, è di giungere ad una implementazione utilizzabile di ASSISTANT. Per raggiungere questo obiettivo dovremo prima risolvere molti problemi, tra cui quelli presentati nella sezione precedente.

Nell'ottica dell'implementazione il lavoro di ASSIST ci sarà sicuramente di aiuto, in quanto con buona probabilità molti dei tool sviluppati per quell'ambiente potranno essere utilizzati come base per ASSISTANT.

Ci riferiamo in particolare al compilatore del linguaggio, che rappresenta sicuramente una delle parti più complesse dell'ambiente, ma per il quale possiamo contare quantomeno sull'esperienza acquisita durante lo sviluppo di ASSIST, e con buona probabilità alcune delle sue parti potranno essere riutilizzate o comunque adattate.

Bibliografia

[1] G. D. Abowd, C. G. Atkeson, J. Hong, S. Long, R. Kooper, and M. Pinkerton. Cyberguide: a mobile context-aware tour guide. *Wirel. Netw.*, 3(5):421–433, 1997.

[2] M. Aldinucci, G. Antoniu, M. Danelutto, and M. Jan. Fault-tolerant data sharing for high-level grid programming: A hierarchical storage architecture. In M. Bubak, S. Gorlatch, and T. Priol, editors, *Achievements in European Research on Grid Systems*, CoreGRID Series, pages 67–81. Springer, Nov. 2007.

[3] M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, S. Magini, P. Pesciullesi, L. Potiti, R. Ravazzolo, M. Torquati, M. Vanneschi, and C. Zoccolo. The implementation of ASSIST, an environment for parallel and distributed programming. In H. Kosch, L. Böszörményi, and H. Hellwagner, editors, *Proc. of 9th Intl Euro-Par 2003 Parallel Processing*, volume 2790 of *LNCS*, pages 712–721, Klagenfurt, Austria, Aug. 2003. Springer.

[4] M. Aldinucci, S. Campa, M. Danelutto, P. Dazzi, P. Kilpatrick, D. Laforenza, and N. Tonellotto. Behavioural skeletons for component autonomic management on grids. In M. Danelutto, P. Fragopoulou, and V. Getov, editors, *Making Grids Work*, CoreGRID, pages 3–15. Springer, Aug. 2008.

[5] M. Aldinucci, S. Campa, M. Danelutto, M. Vanneschi, P. Kilpatrick, P. Dazzi, D. Laforenza, and N. Tonellotto. Behavioural skeletons in gcm: autonomic management of grid components. In D. El Baz, J. Bourgeois, and F. Spies, editors, *Proc. of Intl. Euromicro PDP 2008: Parallel Distributed and network-based Processing*, pages 54–63, Toulouse, France, Feb. 2008. IEEE.

[6] M. Aldinucci, M. Coppola, M. Danelutto, N. Tonellotto, M. Vanneschi, and C. Zoccolo. High level grid programming with ASSIST. *Computational Methods in Science and Technology*, 12(1):21–32, 2006.

[7] M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, and C. Zoccolo. Assist as a research framework for high-performance grid programming environments. Technical Report TR-04-09, Università di Pisa, Dipartimento di Informatica, Italy, Feb. 2004.

[8] M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, and C. Zoccolo. ASSIST as a research framework for high-performance grid programming environments. In J. C. Cunha and O. F. Rana, editors, *Grid Computing: Software environments and Tools*, chapter 10, pages 230–256. Springer, Jan. 2006.

[9] M. Aldinucci, M. Danelutto, and P. Dazzi. Muskel: an expandable skeleton environment. *Scalable Computing: Practice and Experience*, 8(4):325–341, Dec. 2007.

[10] M. Aldinucci, M. Danelutto, and P. Kilpatrick. Autonomic management of non-functional concerns in distributed and parallel application programming. In *Proc. of Intl. Parallel & Distributed Processing Symposium (IPDPS)*, Rome, Italy, May 2009. IEEE.

[11] M. Aldinucci, M. Danelutto, A. Paternesi, R. Ravazzolo, and M. Vanneschi. Building interoperable grid-aware ASSIST applications via Web-Services. In G. R. Joubert, W. E. Nagel, F. J. Peters, O. Plata, P. Tirado, and E. Zapata, editors, *Parallel Computing: Current & Future Issues of High-End Computing (Proc. of PARCO 2005, Malaga, Spain)*, volume 33 of *NIC*, pages 145–152, Germany, Dec. 2005. John von Neumann Institute for Computing.

[12] M. Aldinucci, M. Torquati, and M. Meneghin. FastFlow: Efficient parallel streaming applications on multi-core. Technical Report TR-09-12, Università di Pisa, Dipartimento di Informatica, Italy, Sept. 2009.

[13] ARM Ltd. Arm9/10/11 and cortex technical reference manual. Technical report, ARM Ltd, 2008. Available at http://infocenter.arm.com/help/index.jsp.

[14] ARM Ltd. Arm announces 2ghz capable cortex-a9 dual core processor implementation, 2009. Press Release, available at http://www.arm.com/news/25922.html.

[15] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: a structured high level programming language and its structured support. *Concurrency Practice and Experience*, 7(3):225–255, May 1995.

[16] B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi. SkIE: A heterogeneous environment for HPC applications. *Parallel Computing*, 25(13-14):1827–1852, 1999.

[17] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.

[18] F. Baiardi and M. Vanneschi. *Linguaggi per la programmazione concorrente*. Edizioni Franco Angeli, 1992.

[19] M. Baldauf, S. Dustdar, and F. Rosenberg. A survey on context-aware systems. *Int. J. Ad Hoc Ubiquitous Comput.*, 2(4):263–277, 2007.

[20] C. Bertolli. *Fault Tolerance for High-Performance Applications Using Structured Parallelism Models*. PhD thesis, Università degli Studi di Pisa, Dipartimento di Informatica, 2008.

[21] C. Bertolli, D. Buono, G. Mencagli, and M. Vanneschi. Expressing adaptivity and context awareness in the assistant programming model. In ICST, editor, *Proceedings of the Third International ICST Conference on Autonomic Computing and Communication Systems (Autonomics)*, 2009. ISBN: 978-963-9799-72-1.

[22] C. Bertolli, D. Buono, A. Pascucci, S. Lametti, G. Mencagli, M. Meneghin, and M. Vanneschi. A programming model for high-performance adaptive applications on pervasive mobile grids. In *Parallel and Distributed Computing and Systems (PDCS)*, 2009. To Be Published.

[23] C. Bertolli, J. Gabarró, and M. Meneghin. A markov model for fault-tolerant task parallel computations. In T. Priol and M. Vanneschi, editors, *From Grids To Service and Pervasive Computing (Proc. of the CoreGRID Symposium 2008)*, CoreGRID, pages 123–137, Las Palmas, Spain, Aug. 2008. Springer.

[24] G. Biegel and V. Cahill. A framework for developing mobile, context-aware applications. In *PERCOM '04: Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom'04)*, page 361, Washington, DC, USA, 2004. IEEE Computer Society.

[25] M. Bruno. Supporto all'eterogeneità nell'ambiente di programmazione parallela assist. Bachelor's thesis, Università degli Studi di Pisa, 2008.

[26] D. Buono. Gestione di dati scientifici su griglie computazionali. Bachelor's thesis, Università degli Studi di Pisa, 2008.

[27] H. Chen. *An Intelligent Broker Architecture for Pervasive Context-Aware Systems*. PhD thesis, University of Maryland, Baltimore County, December 2004.

[28] H. Chen, T. Finin, and A. Joshi. An ontology for context-aware pervasive computing environments. *Knowl. Eng. Rev.*, 18(3):197–207, 2003.

[29] K. Cheverst, N. Davies, K. Mitchell, A. Friday, and C. Efstratiou. Developing a context-aware electronic tourist guide: some issues and experiences. In *CHI '00: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 17–24, New York, NY, USA, 2000. ACM.

[30] P. Ciullo, S. Magini, L. Potiti, C. Zoccolo, and G. Virdis. *Tutorial ASSIST-CL*. Università di Pisa, aa, 1.3 edition, 11 2007.

[31] M. Cole. *Algoritmic skeletons: structured management of parallel computation*. MIT Press, 1989.

[32] M. Coppola, P. Pesciullesi, R. Ravazzolo, and C. Zoccolo. A parallel knowledge discovery system for customer profiling. In M. Danelutto, M. Vanneschi, and D. Laforenza, editors, *Proc. of 10th Intl. Euro-Par 2004 Parallel Processing*, volume 3149 of *LNCS*, pages 638–643. Springer, Aug. 2004.

[33] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[34] M. Danelutto, C. Migliore, and C. Pantaleo. An alternative implementation schema for ASSIST parmod. In *Proc. of Intl. Euromicro PDP: Parallel Distributed and network-based Processing*, pages 56–63, Montbéliard, France, Feb. 2006. IEEE.

[35] M. Danelutto and P. Teti. Lithium: A structured parallel programming enviroment in Java. In *Proc. of ICCS: Intl. Conference on Computational Science*, volume 2330 of *LNCS*, pages 844–853. Springer, Apr. 2002.

[36] G. D'Angelo. Raffinamento e sviluppo di uno strumento di esecuzione remota e resource-brokering per griglia computazionale. Bachelor's thesis, Università degli Studi di Pisa, 2006.

[37] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[38] N. Davies, A. Friday, and O. Storz. Exploring the grid's potential for ubiquitous computing. *IEEE Pervasive Computing*, 3(2):74–75, 2004.

[39] A. K. Dey and G. D. Abowd. Towards a better understanding of context and context-awareness. *CHI 2000 Workshop on the What, Who, Where, When, and How of Context-Awareness*, 2000.

[40] I. S. Duff and H. A. van der Vorst. Developments and trends in the parallel solution of linear systems. *Parallel Comput.*, 25(13-14):1931–1970, 1999.

[41] G. Edjlali, G. Agrawal, A. Sussman, and J. H. Saltz. Data parallel programming in an adaptive environment. In *IPPS '95: Proceedings of the 9th International Symposium on Parallel Processing*, pages 827–832, Washington, DC, USA, 1995. IEEE Computer Society.

[42] R. Fantacci, M. Vanneschi, C. Bertolli, G. Mencagli, and D. Tarchi. Next generation grids and wireless communication networks: towards a novel integrated approach. *Wirel. Commun. Mob. Comput.*, 9(4):445–467, 2009.

[43] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 48–63, New York, NY, USA, 1999. ACM.

[44] M. P. I. Forum. Mpi: A message-passing interface. Technical report, 1994.

[45] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. In *Intl J. Supercomputer Applications, 11 (2)*, pages 115–128, 1997.

[46] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, November 1998.

[47] D. Garlan, D. Siewiorek, A. Smailagic, and P. Steenkiste. Project aura: Toward distraction-free pervasive computing. *IEEE Pervasive Computing*, 1(2):22–31, 2002.

[48] D. Gelernter and N. Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):96, 1992.

[49] G. B. Gerace. *La logica dei sistemi di Elaborazione*. Editori Riuniti, 1987.

[50] A. J. Gonzalez and R. Ahlers. Context-based representation of intelligent behavior in training simulations. *Trans. Soc. Comput. Simul. Int.*, 15(4):153–166, 1998.

[51] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI: The Complete Reference*. The MIT Press, 2nd edition, September 1998.

[52] V. Hingne, A. Joshi, T. Finin, H. Kargupta, and E. Houstis. Towards a pervasive grid. *Parallel and Distributed Processing Symposium, International*, 0:207b, 2003.

[53] R. Hockney and C. Jesshope. *Parallel Computers: Architecture, Programming and Algorithms*. Institute of Physics Publishing, 1981.

[54] R. W. Hockney. A fast direct solution of poisson's equation using fourier analysis. *J. ACM*, 12(1):95–113, 1965.

[55] J. Indulska and P. Sutton. Location management in pervasive systems. In *ACSW Frontiers '03: Proceedings of the Australasian information security workshop conference on ACSW frontiers 2003*, pages 143–151, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[56] D. J. Lillethun, D. Hilley, S. Horrigan, and U. Ramachandran. Mb++: An integrated architecture for pervasive computing and high-performance computing. In *RTCSA '07: Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 241–248, Washington, DC, USA, 2007. IEEE Computer Society.

[57] S. Magini, P. Pesciullesi, L. Potiti, G. Virdis, and A. Pascucci. *Descrizione Loader GEA*, 1.1 edition, 2006.

[58] G. Mencagli. Un modello di programmazione parallela per applicazioni adattive e context-aware. Master's thesis, Università degli Studi di Pisa, 2007.

[59] I. Merelli, L. Milanesi, D. D'Agostino, A. Clematis, M. Vanneschi, and M. Danelutto. Using parallel isosurface extraction in superficial molecular modeling. In *1st Intl. Conference on Distributed Frameworks for Multimedia Applications (DFMA 2005)*, pages 288–294, Besançon, France, 2005. IEEE.

[60] K. Nahrstedt, H.-h. Chu, and S. Narayan. Qos-aware resource management for distributed multimedia applications. *J. High Speed Netw.*, 7(3-4):229–257, 1998.

[61] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptation for mobility. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 276–287, New York, NY, USA, 1997. ACM.

[62] M. Parashar and J.-M. Pierson. When the grid becomes pervasive: A vision for pervasive grids. Technical report, Department of Electrical and Computer Engineering at Rutgers, The State University of New Jersey, 2007.

[63] T. Priol and M. Vanneschi. *From Grids To Service and Pervasive Computing*. Springer Publishing Company, Incorporated, 2008.

[64] M. Satyanarayanan. Accessing information on demand at any location. mobile information access. *Personal Communications, IEEE*, 3(1):26–33, Feb 1996.

[65] M. Satyanarayanan. Pervasive computing: vision and challenges. *Personal Communications, IEEE [see also IEEE Wireless Communications]*, 8(4):10–17, 2001.

[66] M. Satyanarayanan. The evolution of coda. *ACM Trans. Comput. Syst.*, 20(2):85–124, 2002.

[67] D. C. Schmidt, D. F. Box, and T. Suda. ADAPTIVE — A Dynamically Assembled Protocol Transformation, Integration and eValuation Environment. *Concurrency: Practice and Experience*, 5(4):269–286, 1993.

[68] D. B. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Comput. Surv.*, 30(2):123–169, 1998.

[69] J. P. Sousa and D. Garlan. Aura: an architectural framework for user mobility in ubiquitous computing environments. In *WICSA 3: Proceedings of the IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture*, pages 29–43, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.

[70] O. Storz, A. Friday, and N. Davies. Towards 'ubiquitous' ubiquitous computing: an alliance with 'the grid'. In *First Workshop on System Support for Ubiquitous Computing Workshop (Ubisys 2003) in association with Fifth International Conference on Ubiquitous Computing*, Seattle, Washington, U.S., Oct. 2003.

[71] R. A. Sweet. A cyclic reduction algorithm for solving block tridiagonal systems of arbitrary dimension. *SIAM Journal on Numerical Analysis*, 14(4):706–720, 1977.

[72] B. Syme. Dynamically linked two-dimensional/one-dimensional hydrodynamic modelling program for rivers, estuaries and coastal waters. Technical report, WBM Oceanics Australia, 1991. available at: http://www.tuflow.com/Downloads/.

[73] J. Thompson, H. R. Sorensen, H. Gavin, and A. Refsgaard. Application of the coupled mike she/mike 11 modelling system to a lowland wet grassland in southeast england. *J. of Hydrology*, 293(1-4):151–179, 2004.

[74] M. Turoff. Past and future emergency response information systems. *Commun. ACM*, 45(4):29–32, 2002.

[75] J. Undercoffer, F. Perich, A. Cedilnik, L. Kagal, and A. Joshi. A secure infrastructure for service discovery and management in pervasive computing. *Mobile Networks and Applications*, 8(2):113–125, 2003.

[76] Università degli Studi di Firenze. Home page del progetto miur-firb "insyeme". `http://www.unifi.it/insyeme/`.

[77] M. Vanneschi. Parallel paradigms for scientific computing. In *Proc. of the European School on Computational Chemistry (1999, Perugia, Italy)*, number 75 in Lecture Notes in Chemistry, pages 170–183. Springer, 2000.

[78] M. Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28(12):1709–1732, Dec. 2002.

[79] M. Vanneschi and L. Veraldi. Dynamicity in distributed applications: issues, problems and the ASSIST approach. *Parallel Computing*, 33(12):822–845, Dec. 2007.

[80] P. Veríssimo, V. Cahill, A. Casimiro, K. Cheverst, A. Friday, and J. Kaiser. Cortex: Towards supporting autonomous and cooperating sentient entities. In *Proceedings of European Wireless 2002*, pages 595–601, Florence, Italy, Feb. 2002.

[81] R. Want, A. Hopper, V. Falc ao, and J. Gibbons. The active badge location system. *ACM Trans. Inf. Syst.*, 10(1):91–102, 1992.

[82] M. Weiser. The computer for the 21st century. *Scientific American*, 265(3):66–75, September 1991.

[83] M. Weiser. Some computer science issues in ubiquitous computing. *SIGMOBILE Mob. Comput. Commun. Rev.*, 3(3):12, 1999.

[84] M. Wu, A. Friday, G. S. Blair, T. Sivaharan, P. Okanda, H. D. Limon, C.-F. Sørensen, G. Biegel, and R. Meier. Novel component middleware for building dependable sentient computing applications. In *ECOOP'04 workshop on component-oriented approaches to context-aware systems (online proceedings)*, Oslo, Norway, June 2004.