Università degli Studi di Pisa

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea Triennale in Informatica

Relazione di Tirocinio

Supporto a meccanismi di comunicazione per architetture many-core

Candidato:
Federico Mariti

Relatore:
Prof. Marco Vanneschi

Anno Accademico 2012/2013

Indice

1 Introduzione **5**

2 Caratteristiche dell'architettura complessiva **8**
2.1 Reti di interconnessione 8
2.1.1 User Dynamic Network 9
2.2 Architettura della memoria principale 10
2.3 Architettura del sottosistema di memoria cache 10

3 Specifica dei meccanismi di comunicazione **13**
3.1 Descrizione dei canali al livello firmware 13
3.2 Specifica dei canali di comunicazione 15
3.3 Implementazione dei canali con memoria condivisa 18
3.3.1 Comunicazione simmetrica 18
3.3.2 Comunicazione asimmetrica in ingresso 22
3.3.3 Comunicazione simmetrica senza barriera di memoria . . 23
3.4 Implementazione dei canali con UDN 25
3.4.1 Comunicazione simmetrica 25
3.4.2 Comunicazione asimmetrica in ingresso 26
3.4.3 Comunicazioni con grado di asincronia maggiore di 1 . . . 28
3.4.4 Implementazione generica 28

4 Esperimenti **30**
4.1 Misurazione della latenza di comunicazione 30
4.1.1 L'applicazione di misurazione 31
4.1.2 Risultati 32
4.2 Benchmark moltiplicazione matrice-vettore 36
4.2.1 Descrizione del problema 36
4.2.2 Sul problema scelto 38
4.2.3 Il metodo di parallelizzazione scelto 39
4.2.4 Sulla latenza di accesso alla memoria 42
4.2.5 Descrizione dell'implementazione 43
4.2.6 Descrizione del metodo di misurazione 44
4.2.7 Risultati delle misurazioni 44

5 Conclusioni **56**
5.1 Sviluppi futuri 57

Elenco delle figure

2.1 Schema del processore Tilera TILE*Pro*64 9
3.1 Componenti di un canale di comunicazione firmware 13
3.2 Canali di comunicazione tra processi 15
3.3 Comunicazione simmetrica su memoria condivisa, non ottimizzata 20
3.4 Comunicazione simmetrica su memoria condivisa 21
3.5 Comunicazione asimmetrica su memoria condivisa 23
3.6 Comunicazione simmetrica su UDN 25
3.7 Comunicazione asimmetrica su UDN 27
4.1 Comportamento temporale dell'applicazione "ping-pong" 32
4.2 Latenza di comunicazione dei canali 33
4.3 Computazione sequenziale del benchmark 37
4.4 Esempio di due computazioni su stream con diversa grana 38
4.5 Computazione dell'implementazione parallela del benchmark . . 41
4.6 Tempi di servizio del sottosistema Map 48
4.7 Tempi di completamento dello stream 49
4.8 Scalabilità del tempo di completamento 50
4.9 Aumento del tempo di calcolo di un singolo prodotto scalare . . . 52
4.10 Tempi di servizio della multicast 53
4.11 Confronto prestazioni delle due implementazioni dei canali 55

Elenco delle tabelle

3.1 Modalità di accesso ad un canale su memoria condivisa 20
4.1 Nomi dei canali di comunicazione realizzati 31
4.2 Latenza dei canali di comunicazione asimmetrici 34
4.3 Tempi di servizio ideali del benchmark 45
4.4 Tempi di servizio effettivo e scalabilità della Map *migliori* quando la Map è collo di bottiglia. 47
4.5 Tempi di servizio effettivo e scalabilità della Map quando la Map non è collo di bottiglia. Vengono mostrati i valori relativi al grado di parallelismo vicino al valore ottimo calcolato. 47

1 Introduzione

Nella programmazione parallela è consolidato l'uso di paradigmi di parallelismo [5] al fine di ottenere una bassa complessità di progettazione dell'applicazione, una semantica e un modello dei costi ben definiti. Il supporto a tali paradigmi è fornito mediante un linguaggio parallelo di alto livello, oppure per mezzo di una libreria, e fa uso di meccanismi di cooperazione e comunicazione tra processi, astraendone dall'implementazione. Allo stesso tempo, nell'ambito di applicazioni parallele con grana fine, è necessario disporre di meccanismi di cooperazione e comunicazione tra processi che siano il più possibile efficienti al fine di produrre prestazioni che scalino con il numero di processi coinvolti. Una implementazione di tali meccanismi che sia capace di minimizzare gli overhead è ottenibile se la progettazione è specifica rispetto alla macchina usata, sfruttando caratteristiche e strumenti della specifica macchina. Relativamente a macchine multi-core (CMP), generalmente con memoria condivisa, l'approccio classico allo sviluppo del supporto ai meccanismi fa uso dei livelli della memoria condivisa. Tuttavia alcune macchine CMP offrono al programmatore altri supporti architetturali che possono essere utilizzati in alternativa o in congiunzione alla memoria condivisa per il supporto ai meccanismi detti. Ad esempio è tendenza comune la realizzazione di nuove macchine CMP, soprattutto rivolte al networking o al processamento di segnali, caratterizzate da più reti on-chip di interconnessione dei core (NoC) usate per distinte finalità. Alcune di queste macchine, come il Tilera TILE*Pro*64, o il NetLogic XLP832, permettono l'uso riservato all'utente di una di queste reti, al fine di realizzare comunicazioni inter-processors senza l'ausilio di memoria condivisa, e quindi senza ricorrere a spin-locks o semafori.

Da una parte la ricerca su sistemi di sintesi di paradigmi di parallelismo è in continua evoluzione e ha prodotto numerosi risultati [2, 4], dall'altra parte è necessario cercare nuovi approcci all'implementazione di meccanismi di cooperazione e comunicazione tra processi. Si cercano quindi soluzioni avanzate che sfruttino le caratteristiche della specifica macchina per la realizzazione efficiente dei meccanismi fondamentali per un supporto ai paradigmi paralleli e che consentano la scalabilità delle prestazioni in presenza di computazioni a grana fine. Tali meccanismi saranno usati nel supporto alle forme di parallelismo e i dettagli implementativi risulteranno del tutto invisibili all'utente.

In questo lavoro di tirocinio si indaga il possibile guadagno prestazionale nell'uso di reti di interconnessione tra core, rispetto ad un uso canonico della memoria condivisa, all'interno di un supporto a forme di comunicazione tra processi con modello a scambio di messaggi. Si tratta di un primo esperimento su questa tematica, ragion per cui, il supporto non è generale ma è specializzato in un ben preciso tipo di comunicazione: l'uso di canali di comunicazione per lo scambio di riferimenti, con un grado di asincronia unitario. È preso in esame il Network Processor Tilera TILE*Pro*64 [3], il quale integra, in un singolo chip, 64 core interconnessi da cinque reti di tipo mesh bidimensionale, caratterizzate da bassissima latenza di trasmissione. Queste strutture di interconnessione sono indipendenti e riservate a compiti specifici, una di queste, chiamata UDN, è a disposizione esclusiva dell'utente, che la può usare per realizzare comunicazioni tra core.

Altri lavori hanno indagato le prestazioni derivanti dall'uso di questo tipo di supporto architetturale per l'implementazione di altri meccanismi che rivestono

un ruolo chiave nel supporto alle applicazioni di grana fine. Ad esempio è pensabile l'utilizzo della UDN per ottimizzare l'implementazione di meccanismi di sincronizzazione tra processi, realizzando l'attesa di un evento come una ricezione sulla rete e rendendo il meccanismo di sveglia indipendente dalla memoria condivisa [6].

Il principale vantaggio nell'utilizzo di strutture di interconnessione tra core per l'implementazione di un supporto alle comunicazioni risiede nella riduzione degli overhead presenti nelle implementazioni classiche (con memoria condivisa) dei canali di comunicazione tra processi, in particolare la latenza per la sincronizzazione a strutture dati condivise e la latenza per garantire la coerenza della cache. Esistono altri vantaggi nell'uso di supporti architetturali diversi dalla memoria condivisa per la realizzazione delle comunicazioni:

- Il disaccoppiamento tra la comunicazione dei processi e l'accesso ai dati in memoria riduce le richieste al sottosistema di memoria condivisa, ai livelli di cache e alle strutture di interconnessione che le gestiscono. Ciò produce una diminuzione del tempo di accesso medio alla memoria rispetto ad una realizzazione delle comunicazioni con la memoria.

- È possibile una forma parziale di sovrapposizione del tempo di comunicazione al tempo di calcolo, anche nel caso in cui i core della macchina non siano provvisti di processori di comunicazione. L'uso di una rete di interconnessione applica in modo primitivo il paradigma di comunicazione a scambio di messaggi, lasciando il processo mittente libero di eseguire altri compiti dopo aver istruito la rete all'invio del messaggio.

L'obiettivo del tirocinio è la realizzazione e il confronto di almeno due versioni dello stesso supporto alle comunicazioni: uno che utilizzi l'approccio "classico", e quindi sia implementato grazie all'uso della memoria condivisa; l'altro che usi l'approccio "nuovo", che consiste nell'uso della rete di interconnessione tra processori messa a disposizione dalla macchina Tilera TILE*Pro*64. Le forme di comunicazione rese disponibili dal supporto sono quelle di uso più comune nei paradigmi di programmazione parallela: canale simmetrico unidirezionale e canale asimmetrico unidirezionale in ingresso. Entrambi trasportano oggetti di tipo "riferimento" e tutt'e due hanno grado di asincronia unitario. Oggetti di tipo riferimento hanno dimensione costante, uguale alla dimensione della parola della macchina; il fatto di non dover gestire messaggi di dimensione arbitraria facilita l'implementazione del supporto con UDN. Questo tipo di implementazione dei canali è frequente in chip multicore per applicazioni di computer networking, dove i messaggi scambiati tra i core sono riferimenti a strutture dati complesse che rappresentano, con un certo livello di indirezione, pacchetti allocati in memoria. D'altronde è necessario tenere presente che con questo approccio il trasferimento vero e proprio dei dati è comunque affidato al sottosistema di cache dell'architettura.

Il supporto alle comunicazioni è stato realizzato per far fronte a computazioni di grana molto fine, per questo scopo, scambiare riferimenti ai dati risulta adeguato. La progettazione del supporto è stata guidata dalla conoscenza di una metodologia di programmazione parallela che fa affidamento sulle forme di parallelismo al fine di dominare la complessità di progettazione delle applicazioni parallele. Ciò ha derivato alcune scelte progettuali, ad esempio: il grado di asincronia unitario dei canali di comunicazione è accettabile per la maggior

parte di forme parallele; un numero massimo di quattro canali è sufficiente per la realizzazione di molte forme parallele.

Struttura della relazione

Capitolo 2 Viene descritta l'architettura complessiva del Tilera TILE*Pro*64, la macchina multicore usata, con particolare attenzione alle caratteristiche e peculiarità dei sottosistemi usati dal supporto: la gerarchia di memoria cache e la rete di interconnessione dei core;

Capitolo 3 Sono specificate le forme di comunicazione e i protocolli di comunicazione trattati. Viene inoltre fornita una descrizione ad alto livello delle implementazioni del supporto alle comunicazioni.

Capitolo 4 Nell'ultimo capitolo si descrivono gli esperimenti che sono stati realizzati al fine di confrontare le prestazioni delle due diverse implementazioni del supporto alle comunicazioni.

2 Caratteristiche dell'architettura complessiva

Il Chip multicore TILE*Pro*64 (figura 2.1) è costituito da 64 processing element (PE) identici, chiamati anche core, connessi tramite una struttura di interconnessione on-chip composta da cinque reti mesh bidimensionali indipendenti. Ogni processing element è costituito da:

- una CPU caratterizzata da una architettura VLIW e contenente due livelli di cache: il primo livello è costituito dalle parti istruzioni (L1i, 16KB) e dati (L1d, 8KB), e il secondo livello (64KB) contiene il DMA engine per supportare le comunicazioni memory-to-cache e cache-to-cache. Non è invece previsto il prefetching della memoria;
- una unità di switching, collegata alle reti di interconnessione dell'architettura, che quindi lo interfaccia con gli altri PE, con i controllori della memoria condivisa, e con i controllori delle unità di ingresso/uscita.

2.1 Reti di interconnessione

La Tilera iMesh [13] è una struttura di interconnessione dei processing element composta da 5 reti mesh bidimensionali identiche e indipendenti, ciascuna delle quali realizza il trasporto di un ben preciso tipo di traffico. Esistono tre reti che si occupano del trasporto di dati della memoria:

TDN Tile Dynamic Network, è responsabile del trasporto delle richieste da PE a PE (e.g. richieste di lettura o scrittura di un blocco o parola);

MDN Memory Dynamic Network, è responsabile del trasporto delle richieste da un PE alla memoria e viceversa, e delle risposte alle richieste inoltrate sulla TDN;

CDN Coherence Dynamic Network, trasporta i messaggi di invalidazione necessari per il meccanismo di coerenza del sottosistema di cache.

Le altre due reti **IDN** e **UDN** permettono una gestione estremamente efficiente di più flussi di dati realizzata attraverso una implementazione firmware delle CPU che consente il direzionamento dei flussi in distinte code di memorizzazione FIFO.

IDN I/O Dynamic Network, principalmente è usata per il trasferimento tra un PE e un dispositivo di I/O e tra I/O e memoria. È consentito l'uso di tale rete solo a servizi di sistema operativo;

UDN User Dynamic Network, accessibile da applicazioni eseguite al livello utente, permette la comunicazione tra i PE senza l'intervento di servizi di sistema. La documentazione di TILE*Pro*64 consiglia l'uso di questa rete come strumento di ottimizzazione di meccanismi di comunicazione [8].

L'ampiezza dei collegamenti è la parola della macchina, 32-bit. Il tempo di trasmissione sulla rete è trascurabile e l'unità di switching di ogni PE opera alla stessa velocità delle CPU. Ne segue che la latenza per la lettura di una parola da un buffer di ingresso in uno switch, applicare il routing, trasmettere la parola sull'interfaccia di uscita e memorizzare la stessa parola nel buffer dello

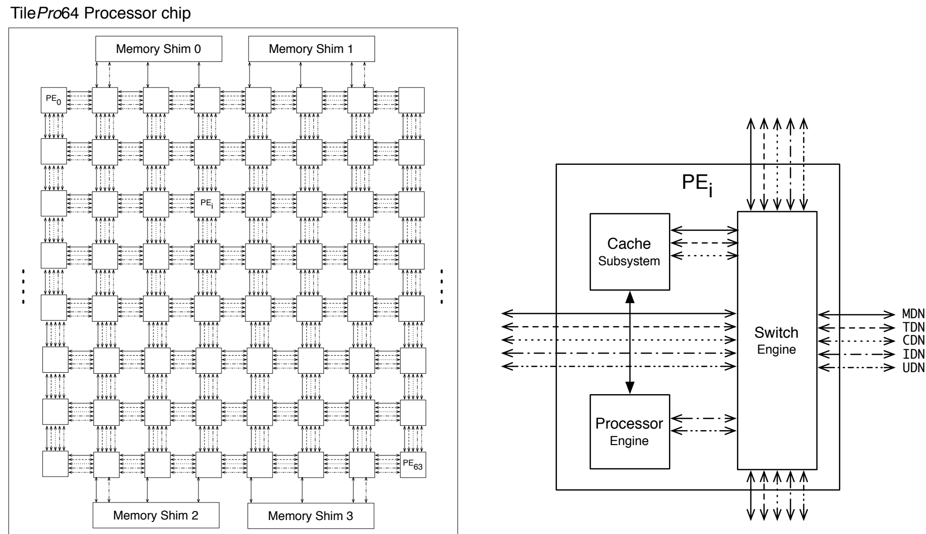


Figura 2.1: Rappresentazione parziale del processore TILE*Pro*64 con il dettaglio dell'architettura di un singolo Processing Element

switch vicino è pari ad un singolo ciclo di clock. La politica di routing è di tipo statico: viene attraversata prima la direzione X poi quella Y. Il controllo di flusso delle reti è di tipo *wormhole* con il flit (l'unità di buffering) della stessa dimensione dell'ampiezza del collegamento. Per ogni rete la massima dimensione del pacchetto è 129 parole.

2.1.1 User Dynamic Network

La UDN è collegata direttamente tra CPU e l'unità di switching del PE. Sono disponibili quattro flussi UDN, nella CPU esistono altrettante code verso le quali sono direzionati i pacchetti del flusso corrispondente; esiste inoltre una quinta coda, detta Catch-All, per qualsiasi altro flusso diverso dai quattro precedenti. Le code di demultiplexing UDN sono collegate direttamente alla Unità di Esecuzione delle Istruzioni della CPU. L'accesso ai flussi UDN avviene tramite letture o scritture a 4 registri generali riservati alla UDN. Il programmatore ha possibilità di uso della UDN per mezzo della libreria C messa a disposizione da Tilera, tuttavia, le vere richieste di scrittura e lettura alla UDN sono effettuate per mezzo di istruzioni assembler, come mostrato nel codice 2.1. L'attesa in seguito all'invocazione di un invio, o di una ricezione, è caratterizzata da un

```
add r59, r5, r6 // Somma r5 a r6 e invia il risultato
                // sul primo flusso UDN
add r5, r6, r60 // leggi una parola dal secondo flusso UDN,
                // sommala a r6 e scrivi il risultato in r5
```

Codice 2.1: Istruzioni assembler per accedere alla UDN

basso consumo di energia e da zero latenza alla sveglia. Si osserva che l'attesa in ricezione avviene quando non esistono dati nella coda del flusso UDN specificato. È possibile anche l'attesa in seguito all'invio quando la rete non è in grado di consumare immediatamente il pacchetto.

In conclusione la rete UDN offre molti vantaggi (bassissima latenza nella comunicazione PE-to-PE, zero latenza nella sveglia, bassi consumi) e presenta alcune limitazioni (il firmware realizza quattro flussi, la dimensione massima dei pacchetti è fissata a circa 118 parole). Tale servizio può perciò risultare interessante, e la documentazione ufficiale Tilera lo precisa [9], per ottimizzare l'implementazione di operazioni critiche per le prestazioni, mentre è raccomandato l'uso della memoria condivisa per la maggior parte delle comunicazioni. Nel caso di computazioni di grana fine, i meccanismi chiave sono quelli di sincronizzazione e comunicazione tra processi, la cui ottimizzazione fornisce miglioramenti notevoli alle prestazioni.

2.2 Architettura della memoria principale

Esistono quattro controllori della memoria principale collegati ai PE sul bordo superiore e inferiore della mesh. Le pagine di memoria condivisa hanno dimensione 64KB e sono allocate in modo interleave, in parti di 8KB, nei controllori di memoria ("stripped main memory mode"), tale configurazione rende uniforme l'accesso ai controllori di memoria bilanciando il carico di richieste. È possibile configurare l'allocazione delle pagine in specifici controllori. TILE*Pro*64 definisce un modello *rilassato* di consistenza della memoria, in altre parole l'ordinamento delle operazioni di store non è totale. Valgono le seguenti due proprietà:

- la sequenza delle istruzioni originale del programma può essere modificata a tempo di compilazione, come risultato di ottimizzazioni.
- le operazioni di store eseguite da un PE appaiono visibili simultaneamente a tutti gli altri PE, ma possono diventare visibili al PE che ha emesso la scrittura prima che avvenga la visibilità globale.

L'architettura mette a disposizione un'istruzione di barriera di memoria, chiamata anche *memory fence*, la quale stabilisce un ordinamento tra le istruzioni di memoria, altrimenti non ordinate: le operazioni di memoria nel programma prima dell'istruzione barriera di memoria, sono rese globalmente visibili prima di qualsiasi operazione dopo la barriera di memoria.

2.3 Architettura del sottosistema di memoria cache

La cache L1 è divisa nella parte istruzioni (L1i) e nella parte dati (L1d), la prima ha dimensione 16KB con blocchi di 64B, la seconda ha dimensione 8KB con blocchi di 16B. L'allocazione dei blocchi nelle due parti L1 avviene solo fault in lettura, le scritture sono write through. La cache L2 ha dimensione 64KB, blocchi di 64B e l'allocazione dei blocchi avviene con fault sia in lettura che in scrittura; la politica di scritture è write back.

La coerenza della memoria cache è garantita automaticamente, per mezzo del firmware della macchina. È possibile configurare la macchina in diverse modalità:

- memoria coerente nelle cache;
- memoria non coerente nelle cache, è onere del programmatore dell'applicazione garantire la coerenza mediante operazioni di flush e di deallocazione (è definita solo una primitiva che invalida il blocco nel PE che ha eseguito la primitiva stessa);
- non uso della gerarchia cache.

La coerenza automatica si basa su invalidazione, l'implementazione è *Directory-based*. In particolare la tabella Directory è implementata in modo distribuito nei PE della macchina. Per un certo blocco di cache b viene usato il termine Home con il consueto significato nei protocolli di coerenza cache: il PE che è Home per il blocco b detiene e gestisce le informazioni di coerenza della cache di b, e invia messaggi di invalidazione quando necessario. Un PE gestisce la coerenza, e contiene le entrate della directory, dei soli blocchi di cui è Home. Nell'architettura Tilera un PE Home per un certo blocco b ha anche la caratteristica di possedere sempre la copia aggiornata di b nella propria L2 (in un protocollo di coerenza tale PE verrebbe chiamato Owner di b oltre che Home). Questo consente di poter vedere un terzo livello di cache distribuito sulle L2: se un PE esegue una lettura da memoria che risulta miss sia nella L1d che nella L2 locali, tale richiesta viene inoltrata al PE che è Home del blocco corrispondente, piuttosto che essere inviata direttamente alla memoria. È onere della L2 di tale PE rispondere alla richiesta: se il blocco è allocato allora viene inviata immediatamente la risposta contenente il valore del blocco, altrimenti viene inoltrata la domanda di trasferimento alla memoria principale, e successivamente il messaggio di risposta al PE richiedente. Ne segue una caratterizzazione dei due livelli di cache: L1 è una cache privata per la propria CPU, la L2 è una cache condivisa con gli altri PE, in quanto può contenere sia blocchi acceduti dalla CPU locale, che blocchi di cui è Home e per i quali ha ricevuto una richiesta da un altro PE.

Lo schema con cui i blocchi di memoria vengono associati ai PE Home è dinamico e può essere controllato dal programmatore in modo da ottimizzare la specifica computazione, ad esempio rendendo massima la località delle informazioni nei PE. Sono disponibili tre modi per definire l'homing:

Local Homing è la modalità predefinita per lo stack dei processi e threads. Una pagina di memoria è Home nel PE che ha effettuato l'accesso. Ne segue che in tale modalità tutti gli accessi sono diretti ai controllori di memoria e non viene usato il meccanismo di L3 cache distribuito nelle cache L2 degli altri PE. Ciò è utile per dati privati in quanto non trarrebbero vantaggio dall'uso della cache di una altro PE come L3.

Hash-for-Home è la modalità predefinita per tutti gli altri dati del processo. Una pagina di memoria viene homed in modo sparso nei PE della macchina con granularità di un blocco L2. Tale dispersione delle pagine di memoria nella cache dei PE consente una distribuzione uniforme del traffico nelle reti del chip e un effettivo bilanciamento del traffico di richieste alla memoria tra l'insieme dei PE.

Remote Homing per una generica pagina di memoria viene impostato il corrispondente PE Home. Tale PE è anche chiamato L3 per la pagina. Le

richieste di accesso ad un qualsiasi blocco interno alla pagina, da parte di altri PE, vengono inoltrate al PE Home; quest'ultimo ha l'onere di rispondere con il trasferimento del blocco, eventualmente effettuando l'accesso alla memoria principale se il blocco non è presente nella sua L2. Tale configurazione è utile con strutture dati condivise, il cui accesso avviene con uno schema produttori-consumatore. Configurare la pagina dell'oggetto in questione come Home nel PE che esegue il consumatore aumenta la località degli accessi di tale processo, il quale si trova direttamente nella propria L2 la struttura modificata dai produttori. Le scritture effettuate dai produttori, infatti, sono write through nella cache del PE Home.

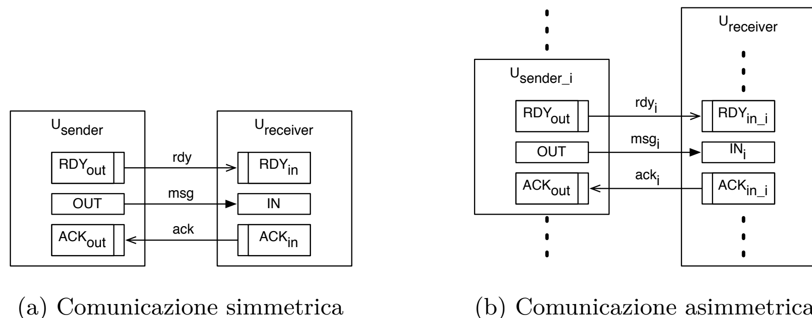
3 Specifica dei meccanismi di comunicazione

I meccanismi di message passing presi in considerazione sono canali unidirezionali di forma simmetrica e asimmetrica in ingresso, entrambi con grado di asincronia unitario e tipo di dati trasmessi "riferimento alla memoria condivisa". Viene fornita una descrizione astratta di questi meccanismi e delle relative primitive. Nelle sezioni successive sono descritte le implementazioni che fanno uso di due diversi supporti architetturali: la memoria condivisa con l'uso della gerarchia cache, e la rete di interconnessione tra i PEs.

3.1 Descrizione dei canali al livello firmware

Prima di descrivere l'implementazione delle forme di comunicazione considerate, viene descritto brevemente il protocollo tipico per realizzare la comunicazione di unità di elaborazione firmware in modo indipendente dal tempo [10]. Le idee alla base di questo protocollo sono utilizzate per realizzare dei meccanismi efficienti e ottimizzati sull'architettura di TILE*Pro*64.

Si consideri la comunicazione simmetrica, il protocollo definisce due eventi: "la presenza di un messaggio nel canale" e "la ricezione del messaggio da parte del ricevente"; nel livello firmware tali eventi sono realizzati tramite due collegamenti di un bit detti di Ready (Rdy) e di Acknowledgement (Ack) e da una coppia di indicatori di interfaccia per ciascun collegamento. Un canale di comunicazione a livello firmware è perciò costituito da tre componenti: l'interfaccia di uscita, il collegamento fisico e l'interfaccia di ingresso. Le interfacce contengono l'indicatore di uscita, quello di ingresso e il registro di uscita contenente il dato da trasmettere. Il collegamento è costituito dalle linee per connettere gli indicatori e i registri delle due interfacce. Un indicatore di interfaccia di uscita mette a disposizione l'operazione *set* che provoca l'assunzione del valore 1 nell'indicatore di interfaccia di ingresso ad esso collegato. Un indicatore di interfaccia di ingresso mette a disposizione l'operazione *test* con la quale viene letto il valore dell'indicatore, e l'operazione *reset* con l'esecuzione della quale viene impostato a 0 il valore di uscita dell'indicatore stesso. Considerando due unità U_{sender} e $U_{receiver}$ collegate dai collegamenti *rdy(1)*, *msg(L)*, *ack(1)* come in figura 3.1a, il protocollo di comunicazione è perciò definito come in codice 3.1



(a) Comunicazione simmetrica (b) Comunicazione asimmetrica

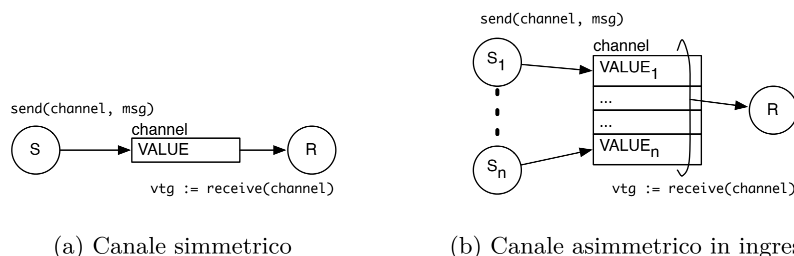
Figura 3.1: Componenti di un canale di comunicazione firmware

```
U_sender:send ::
  wait until ACK_out.test() is equal to 1
  send message msg to receiver and
    do RDY_out.set() and
    do ACK_out.reset()

U_receiver:receive ::
  wait until ACK_in.test() is equal to 1
  use msg received and
    do ACK_in.set() and
    do RDY_in.reset()
```

Codice 3.1: Descrizione astratta del protocollo di comunicazione del canale firmware simmetrico

Il protocollo di comunicazione del canale asimmetrico in ingresso segue da quello descritto nel caso simmetrico: dal punto di vista logico è come se venissero adottati tanti canali simmetrici quante sono le unità mittenti, ogni canale collega una unità mittente all'unità destinataria, figura 3.1b. Il comportamento della funzione di invio rimane invariato rispetto a quello descritto precedentemente. Quello della funzione di ricezione testa tutti gli indicatori di tipo Rdy e con una certa politica ne seleziona uno tra quelli attivi ed esegue il protocollo del caso simmetrico, leggendo il valore, resettando il Rdy e inviando l'ack per il canale scelto.



(a) Canale simmetrico (b) Canale asimmetrico in ingresso

Figura 3.2: Rappresentazione astratta di canali con grado di asincronia unitario

3.2 Specifica dei canali di comunicazione

Con canale di comunicazione si intende il collegamento logico tramite il quale due o più processi comunicano. L'astrazione di canale come meccanismo primitivo di scambio di informazioni tra processi viene fornita dal supporto a tempo di esecuzione della nostra libreria. Il canale *simmetrico* è un oggetto che mette in comunicazione un processo mittente con un processo destinatario, consentendo l'invio di messaggi dal processo mittente al processo destinatario. Un canale *asimmetrico in ingresso* è un oggetto che mette in comunicazione più processi mittenti con un singolo processo destinatario, consentendo a ciascun mittente l'invio di messaggi al processo destinatario. Il destinatario riceve i messaggi in modo non deterministico e senza saperne la provenienza, a meno di comunicazioni esplicite.

Per entrambi i canali sono definite due primitive di comunicazione, quella di invio e quella di ricezione le quali sono usate rispettivamente dal processo mittente e dal processo destinatario del canale di comunicazione. Si considera la seguente dichiarazione C delle primitive:

```
void sym_send(ch_sym_t *ch_descr, const void *msg)
void *sym_receive(ch_sym_t *ch_descr)

void asym_send(ch_asym_t *ch_descr, const void *msg, int
    rank)
void *asym_receive(ch_asym_t *ch_descr)
```

Ad ogni primitiva viene quindi passato, come primo parametro, il descrittore di canale come riferimento ad una struttura dati allocata in memoria condivisa. Non viene usata la tecnica zero-copy in quanto non è applicabile all'implementazione che fa uso della UDN. La primitiva di invio ha come secondo parametro il valore del messaggio, quella di ricezione ritorna il valore letto dal canale come valore di ritorno, il valore del messaggio è copiato prima nel canale poi nella variabile targa. Nel caso asimmetrico è necessario specificare alla primitiva di invio l'identificatore del mittente all'interno del canale, ciò è necessario per l'implementazione, come spiegato nelle sezioni successive per entrambi i supporti.

Si considera il seguente comportamento ad alto livello delle due forme di comunicazione con il grado di asincronia considerato:

canale simmetrico con grado di asincronia unitario prevede che il processo mittente possa inviare fino ad un messaggio senza attendere che il

destinatario lo abbia ricevuto. Nel caso in cui il mittente intenda inviare un secondo messaggio senza che il destinatario abbia ricevuto quello precedente occorre che il mittente attenda la ricezione da parte del destinatario.

canale asimmetrico in ingresso con grado di asincronia unitario ogni processo mittente può inviare fino ad un messaggio senza attendere che il destinatario lo abbia ricevuto. Nel caso in cui un generico mittente intenda inviare un secondo messaggio senza che il destinatario abbia ricevuto il messaggio precedentemente inviato dal quello specifico mittente, occorre che tale mittente attenda la ricezione da parte del destinatario del messaggio precedente.

Per quanto riguarda il comportamento del ricevente di un canale asimmetrico, non si impone una specifica politica di ricezione. Il comportamento che può essere considerato naturale è la ricezione FIFO dei messaggi inviati dai mittenti, tuttavia tale funzionamento non è imposto. Si può infatti parlare del canale asimmetrico in ingresso come un caso particolare di comportamento non deterministico del destinatario nella ricezione da più canali simmetrici.

Nelle sezioni successive verranno descritte le implementazioni dei due tipi di canale che sfruttano supporti architetturali diversi. I protocolli di comunicazione che costituiscono l'implementazione con i diversi supporti fanno riferimento al protocollo usato al livello firmware per la comunicazione di due unità di elaborazione indipendente dal tempo. Tale protocollo è chiamato nel seguito Rdy-Ack. Descriviamo qui, in modo astratto, le azioni dei processi comunicanti definite dal protocollo che assicurano la correttezza della comunicazione, ovvero:

- non si verifica mai la perdita di un messaggio,
- se il/un processo mittente invia un messaggio allora prima o poi il processo destinatario lo deve ricevere,
- se il processo destinatario riceve un messaggio allora prima o poi il processo mittente che ha effettuato l'invio di quel messaggio deve essere in grado di inviarne un altro.

```
send ::
  wait until acknowledgement signal is received
  send the message to receiver and
    reset acknowledgement flag and
    signal ready to the receiver

receive ::
  wait until ready signal is received
  copy the message received from sender into vtg and
    reset ready flag and
    signal acknowledgement to the sender
```

Codice 3.2: Descrizione astratta del protocollo di comunicazione per un canale simmetrico con grado di asincronia 1

Per il canale simmetrico si hanno le azioni delle due entità descritte nel codice 3.2. Il canale asimmetrico in ingresso ha protocollo simile, in quanto può essere visto come costituito da tanti canali simmetrici Rdy-Ack quanti sono i mittenti, dove ogni canale collega un mittente al destinatario. Il comportamento dell'invio di un messaggio è esattamente lo stesso di quello adottato nel canale simmetrico, mentre la ricezione ha l'onere di scegliere in modo non deterministico un canale da cui ricevere tra i canali pronti.

3.3 Implementazione dei canali con memoria condivisa

Sfruttando la memoria condivisa si ha una implementazione semplice e lock-free delle comunicazioni simmetrica e asimmetrica con il protocollo Rdy-Ack. Ciò è determinato dal fatto che non esiste un buffer o altre strutture dati sulle quali i processi devono accedere in modo indivisibile. L'adozione di un grado di asincronia unitario e la sincronizzazione tramite gli eventi Rdy e Ack permette di effettuare accessi al descrittore del canale senza la necessità di mutua esclusione.

3.3.1 Comunicazione simmetrica

Si considera la comunicazione simmetrica: il descrittore di canale è costituito da tre informazioni: *a*) il valore di Rdy, *b*) il valore di Ack, *c*) il valore del messaggio. Ogni informazione è allocata in una parola, in particolare abbiamo la seguente definizione del descrittore di canale: i flag di Rdy e di Ack sono di tipo intero, il messaggio è di tipo riferimento.

```
struct ch_sym_sm_rdyack_t {
  int rdy;
  int ack;
  void *value;
};
```

La segnalazione di un evento di Rdy o di Ack avviene impostando al valore 1 il corrispondente campo nel descrittore di canale. L'attesa attiva di un processo sul canale è realizza con la strategia *retry* sul valore del flag Rdy o Ack: si continua a testare il valore del flag fino a quando non assume valore 1. Il protocollo di comunicazione viene perciò realizzato con le seguenti azioni:

```
send(ch_descr, msg) ::
  wait until ch_descr->ack flag is equal to 1;
  reset ch_descr->ack flag to 0;
  copy msg into ch_descr->value entry;
  memory_fence();
  set ch_descr->rdy flag to 1;

receive(ch_descr) ::
  wait until ch_descr->rdy flag is equal to 1;
  reset ch_descr->rdy flag to 0;
  read the ch_decsr->value entry;
  set ch_descr->ack flag to 1;
```

Codice 3.3: Descrizione astratta del protocollo di comunicazione Rdy-Ack su memoria condivisa

La mutua esclusione nell'uso del valore del canale da parte dei processi mittente e destinatario è garantita dalla sincronizzazione dei due processi sui due valori di Rdy e Ack: ogni processo accede al valore del canale solo dopo che il processo partner ha notificato tale possibilità. Per garantire la correttezza della comunicazione è quindi sufficiente adottare la seguente condizione iniziale:

- all'avvio dell'applicazione tutti i canali devono avere descrittore con campo Ack inizializzato ad 1 e capo Rdy inizializzato a 0.

Tale condizione iniziale, insieme all'adozione del protocollo, garantisce la seguente proprietà:

- ogni notifica di evento Rdy o Ack trova sempre l'evento corrispondente precedentemente falso.

Il fatto che sia possibile evitare l'uso di meccanismi di lock per l'esecuzione indivisibile del codice è un aspetto positivo dell'implementazione con memoria condivisa del protocollo Rdy-Ack, in quanto si evitano i relativi overhead sulle prestazioni. Al fine di garantire la correttezza è tuttavia richiesto l'uso di scritture sincrone alla memoria condivisa. È necessario che le scritture effettuate dal mittente sul valore del canale e sul flag di Rdy siano viste dal destinatario nello stesso ordine. Come descritto in sezione 2.2, TILE*Pro*64 adotta un modello rilassato di consistenza della memoria, sia per quanto riguarda l'ordinamento di istruzioni all'interno di una CPU, sia per l'atomicità delle scritture in memoria condivisa. Ciò rende necessario l'uso di una barriera di memoria tra le due scritture effettuate dal mittente (riga 5 del codice 3.3), in modo che la scrittura del valore 1 sul flag Rdy sia sempre vista dal processo destinatario dopo la scrittura del messaggio nel canale. Il comportamento sincrono delle scritture in memoria, indotto dall'uso della barriera, produce overheads aggiuntivi rispetto al comportamento asincrono delle scritture. La barriera della memoria, e la corrispondente degradazione delle prestazioni, può essere eliminata con l'adozione di un protocollo di comunicazione alternativo, come mostrato in sezione 3.3.3.

La politica di attesa attiva *retry* è caratterizzata da un aumento dei conflitti alla memoria principale. Per evitare ciò e ridurre la latenza di sveglia si prendono in considerazione solo implementazioni che fanno uso della gerarchia cache. In tale scenario la prima lettura del valore di un flag causa il trasferimento del blocco corrispondente nei livelli L2 e L1D della cache locale, e le letture successive del valore del flag rimangano interne alla CPU, in quanto servite dalla cache locale. Quando il processo partner esegue la scrittura sul flag il meccanismo di coerenza della cache provvede a notificare il cambiamento e aggiornare il valore nella cache locale.

Come descritto in sezione 2.3 il meccanismo di coerenza della cache è configurabile in molti aspetti, in particolare è possibile impostare il PE Home per una certa pagina di memoria. In sezione 2.3 viene spiegato come l'uso della strategia *Remote Homing* sia conveniente in computazioni con singolo consumatore di una struttura dati condivisa, e permetta di evitare l'uso di invalidazioni e conseguenti copie di blocchi. Di seguito descriviamo il comportamento della comunicazione simmetrica con il protocollo Rdy-Ack e il descrittore di canale presentati, con la configurazione predefinita della coerenza della cache, quindi verrà presentata una versione ottimizzata che minimizza i messaggi di coerenza e le copie dei blocchi grazie alla configurazione dei PEs Home e al partizionamento del descrittore del canale nelle cache dei PEs.

La strategia predefinita di allocazione delle pagine di memoria nelle cache è *Hash-for-Home* [9]. La gestione delle informazioni di coerenza di cache, in particolare l'entrata della Directory, del blocco che contiene il descrittore di canale allocato viene affidata ad un generico processing element PE_k. Durante l'esecuzione del programma parallelo il processo mittente, eseguito in PE_i, e il processo destinatario, eseguito in PE_j, eseguono, per mezzo delle primitive di comunicazione, degli accessi al descrittore di canale. In generale $k \neq i$ e $k \neq j$ quindi si hanno tre copie del blocco contenente il descrittore di canale come mostrato in figura 3.3. La modifica di un campo del descrittore del canale effettuata da uno dei due processi comunicanti provoca l'invalidazione del bloc-

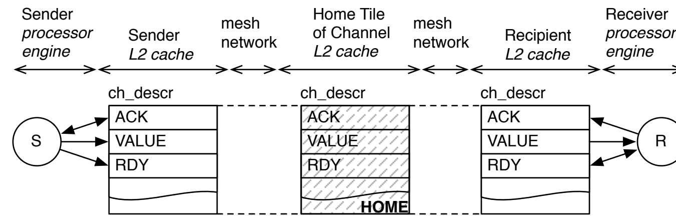


Figura 3.3: Rappresentazione di una comunicazione simmetrica tra un processo mittente S e uno destinatario R con uso della memoria condivisa e gestione predefinita della coerenza di cache (hash-for-home strategy)

co di L2 nel PE che esegue il processo partner. Quando il PE che contiene il blocco invalidato riferisce il descrittore di canale, l'intero blocco sarà trasferito dal PE Home, PE_k, al PE stesso. Nel caso migliore si ha una invalidazione e corrispondente trasferimento di blocco al termine di ogni esecuzione di una primitiva di comunicazione, quando viene notificato un evento di sincronizzazione al processo partner per mezzo di una scrittura nel descrittore di canale.

Questo offre lo spunto su come ottimizzare l'uso delle cache: la scrittura sul campo del descrittore di canale che implementa un segnale di sincronizzazione dovrebbe essere inoltrata direttamente alla cache del PE che attende il segnale, evitando il trasferimento di un blocco di L2 per la trasmissione di una sola parola. Questo comportamento è possibile in quanto:

- per ogni campo del descrittore esiste uno solo dei due processi che vi accede in lettura(vedi tabella 3.1),
- è possibile impostare il PE Home di un certo blocco di cache L2,
- come descritto nella sezione 2.3 un PE che è Home per un blocco detiene sempre la copia corretta del blocco stesso.

Il descrittore di canale viene partizionato in due parti, allocate in blocchi distinti e con diverse configurazioni di Homing:

la parte di output contiene tutti i campi acceduti in lettura del mittente (Ack), ed è allocata impostando come PE Home quello che esegue il processo mittente,

la parte di input contiene tutti i campi acceduti in lettura dal mittente (Rdy e Value), ed è allocata impostando come PE Home quello che esegue il processo destinatario.

	Rdy	Ack	Value
Sender	Write Only	Read and Write	Write Only
Receiver	Read and Write	Write Only	Read Only

Tabella 3.1: Modalità di accesso ai campi del descrittore di canale su memoria condivisa

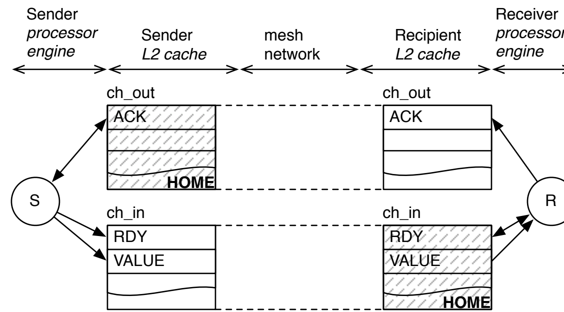


Figura 3.4: Rappresentazione di una comunicazione simmetrica tra i processi S e R con uso della memoria condivisa e gestione ottimizzata della coerenza di cache con strategia Remote Homing per il descrittore di canale

Con tale configurazione si incrementa la località delle informazioni: la coerenza dei dati e l'allocazione degli stessi avviene nella cache del PE che consuma tali dati, permettendo al PE produttore di effettuare scritture direttamente nella cache locale del consumatore tramite messaggi write-through. I processo consumatore dei dati trova le informazioni già aggiornate direttamente nella cache locale alla CPU che lo esegue.

Si osserva inoltre che, in tale computazione, il processo consumatore di una informazione esegue anche scritture sul blocco, ciò causa l'invio di messaggi di invalidazione del blocco alla cache L2 del PE che esegue il produttore. Tuttavia in futuro non si verificherà mai un trasferimento del blocco in quanto il produttore esegue *esclusivamente* scritture sul blocco, ciò causa l'allocazione del blocco nella L2 locale (quando il blocco è non valido), la modifica della singola parola scritta e l'invio del messaggio Write-Through alla cache L2 del PE consumatore.

In conclusione si fa presente che esiste un terzo blocco di cache, usato per il supporto del canale di comunicazione, che contiene i riferimenti alle due parti del descrittore di canale. Tale blocco è sempre acceduto in sola lettura, quindi non presenta particolari problemi per le prestazioni e può essere allocato in uno dei due PE comunicanti con strategia Remote Homing oppure in un terzo PE con strategia Hash-for-Home.

3.3.2 Comunicazione asimmetrica in ingresso

Il canale asimmetrico in ingresso è visto come costituito da tanti canali simmetrici quanti sono i mittenti. Al fine di ottimizzare le prestazioni non si usa direttamente l'implementazione del canale simmetrico ma si sfruttano comunque i risultati e l'analisi di tale meccanismo, in particolare per quanto riguarda l'allocazione degli oggetti condivisi in cache al fine di garantire la maggiore località possibile.

Siano n i processi mittenti del canale, dal punto di vista logico il canale asimmetrico usa n canali simmetrici implementati con il protocollo Rdy-Ack, e perciò caratterizzati dal valore della tripla $< Rdy,\ value,\ Ack >$. Come descritto nella sezione 3.3.1 i valori di *Rdy* e *value* di un canale simmetrico sono allocati in un blocco che ha per Home il PE che esegue il ricevente mentre *Ack* ha per Home il il PE che esegue il mittente. Per minimizzare il numero dei blocchi allocati in cache si raggruppa l'insieme dei Rdy e dei valori $\{< Rdy_i,\ value_i > \mid i \in \{1, \ldots, n\}\}$ allocandolo come vettore di n coppie $< Rdy,\ value >$ e impostando come Home il PE destinatario, con strategia Remote Homing. I valori di *Ack* sono invece allocati ciascuno in un blocco che ha per home il PE mittente corrispondente, questo consente la maggior località possibile.

Si fa presente che sebbene tale configurazione permetta la massima località delle informazioni ai PEs, presenta tuttavia l'allocazione nella cache L2 del destinatario di un numero di blocchi che è lineare rispetto al numero dei mittenti: si hanno $\frac{2 \cdot 4 \cdot n}{64}$ blocchi necessari a memorizzare l'array di coppie $< Rdy,\ value >$ e n blocchi allocati per le scritture sui flag *Ack* degli n mittenti. Considerando il caso della comunicazione con il massimo numero (63) di mittenti, si hanno 72 blocchi allocati nel PE destinatario, ovvero uno spazio di 4.5 KB su un totale di 64KB nella cache L2. Dato che siamo interessati a massimizzare le prestazioni si assume che l'uso di tale spazio nella L2 del destinatario non sia problematico per l'applicazione.

Si osserva che, soprattutto con un numero elevato di mittenti, la sveglia del destinatario sarà più lenta rispetto a quella nel canale simmetrico, infatti l'attesa è implementata eseguendo la scansione delle variabili *Rdy* perciò nel caso peggiore è necessario effettuare n accessi alla memoria dalla scrittura del valore 1 in un flag *Rdy* da parte del mittente.

Il descrittore di canale è costituito da n riferimenti ai valori di *Ack* allocati con strategia Remote Homing nei PEs mittenti, e il riferimento alla base del vettore delle coppie $< Rdy, value >$ allocato con strategia Remote Homing nel PE destinatario. Come descritto nella sezione 3.2 il protocollo di comunicazione rimane sostanzialmente invariato rispetto al caso simmetrico, rimane da specificare la politica adottata dal ricevente per selezionare il mittente da cui ricevere tra quelli che hanno inviato un nuovo messaggio. Il supporto della primitiva di ricezione adotta una scansione lineare dell'insieme dei flag *Rdy* alla ricerca di uno attivo. Ad ogni ricezione viene salvato l'indice del mittente da cui si è effettuata la lettura, in modo tale che alla prossima esecuzione della ricezione la scansione dei *Rdy* si avviata dal mittente successivo. La ricezione è quindi non deterministica ed è attuata un politica di fairness, sebbene la proprietà non sia formalmente soddisfatta.

Al fine di attuare il protocollo di comunicazione esiste una questione che l'implementazione del supporto della primitiva di invio deve risolvere: conoscere l'identificatore all'interno del canale asimmetrico del mittente che ha eseguito la

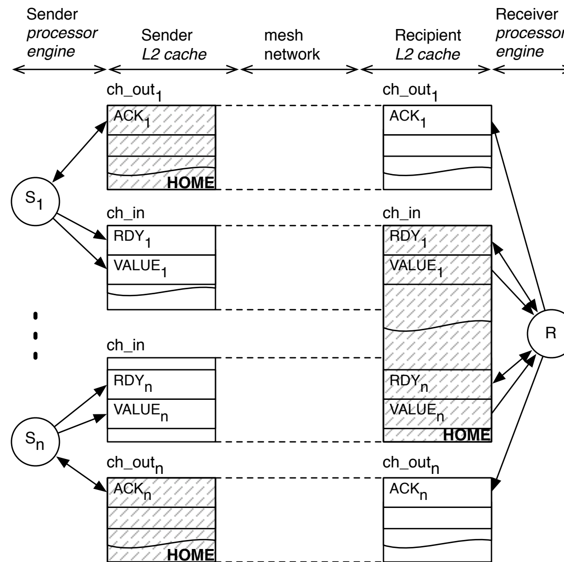


Figura 3.5: Rappresentazione di una comunicazione asimmetrica tra i processi $\{S_1, \ldots, S_n\}$ e R con uso della memoria condivisa e gestione ottimizzata della coerenza di cache con strategia Remote Homing per il descrittore di canale

primitiva. Tale informazione è ovviamente indispensabile al protocollo di invio, tra le possibili soluzioni si sono valutate le seguenti due:

- la primitiva di invio contiene un terzo parametro, l'identificatore del mittente nel canale, oltre al riferimento al descrittore di canale e al valore del messaggio,
- ogni mittente usa un proprio descrittore di canale che contiene l'identificatore all'interno del canale, per far ciò è necessaria una funzione di inizializzazione eseguite da ogni mittente prima dell'avvio della computazione.

Si è scelta la prima soluzione in quanto anche l'implementazione che sfrutta la UDN verifica lo stesso problema perciò è assicurata l'uniformità della primitiva di invio nelle due implementazioni del canale asimmetrico. È responsabilità dell'utente far si che ogni processo mittente del canale conosca il proprio identificatore univoco all'interno del canale e quindi poter invocare correttamente la primitiva di invio sul canale.

3.3.3 Comunicazione simmetrica senza barriera di memoria

Nella sezione 3.3.1 si è descritta l'implementazione del supporto alla comunicazione simmetrica per mezzo di strutture dati lock-free e, per garantire la correttezza, l'uso della barriera di memoria nel supporto della primitiva di invio. Si propone un'altra possibile implementazione della comunicazione simmetrica con un diverso protocollo che consente il corretto funzionamento con strutture dati lock-free senza l'uso di barriere di memoria. Il protocollo che si va a definire si

```
send(ch_descr, msg) ::
  wait until ch_descr->ack flag is equal to 1;
  reset ch_descr->ack flag to 0;
  copy msg into ch_descr->value entry;

receive(ch_descr) ::
  wait until ch_descr->value is not equal to NULL;
  read the ch_descr->value entry;
  set ch_descr->value to NULL;
  compiler_barrier();
  set ch_descr->ack flag to 1;
```

Codice 3.4: Descrizione astratta del protocollo di comunicazione Null-Ack su memoria condivisa

rifà a quello Rdy-Ack, in quanto sono mantenuti gli eventi di sincronizzazione "messaggio pronto" (Rdy) e "messaggio ricevuto" (Ack). In questo caso l'evento di Rdy non è segnalato esplicitamente ma risulta implicito nel cambiamento di valore del canale. Viene infatti definito il valore particolare α che non può essere assunto dai messaggi trasmessi nel canale e che indica la situazione in cui il canale è vuoto. Il canale assume valore α all'avvio dell'applicazione e al termine di ogni esecuzione della primitiva di ricezione. Il flag `Rdy` è eliminato dall'implementazione, il destinatario usa il valore del canale e il valore α per determinare la presenza di un nuovo messaggio, come descritto in codice 3.4.

Anzitutto si osserva che tale protocollo si applica bene alla nostra implementazione dei canali, in quanto si può definire, senza perdere di generalità, il valore particolare α come il valore di puntatore "nullo": `NULL = (void *) 0`. Passando all'analisi della correttezza, la barriera di memoria è stata eliminata nella primitiva di invio, in quanto esiste un unico oggetto (il campo `value`) che è scritto dal mittente ed è acceduto da altri processi. Nella primitiva di ricezione si hanno invece due oggetti, i campi `value` e `ack`, che sono scritti dal ricevente ed acceduti dal processo partner. Grazie alla specifica politica di homing di tali oggetti e alla modalità di accesso a questi del processo mittente, l'uso di una memory fence non è necessario. Risulta invece sufficiente che il PE che esegue il processo destinatario produca le due scritture nell'ordine specificato. Il campo `value` ha per Home il PE che esegue il processo ricevente ed è acceduto in scrittura dal processo mittente dopo aver letto la variazione di valore del campo `ack`. La prima scrittura è quindi locale alla L2 del ricevente, inoltre non ha importanza in che istante diventa visibile agli altri PEs, in quanto l'oggetto scritto (`value`) è acceduto in sola scrittura dal mittente. La seconda scrittura è una write-through alla L2 del mittente, il quale, una volta letto il nuovo valore può effettuare una nuova scrittura sul campo `value` che sicuramente è già stato modificato a `NULL`. Per imporre l'ordinamento delle istruzioni nel PE ricevente si usa l'istruzione `compiler_barrier()`, la quale è molto diversa da una barriera di memoria. Il comportamento delle scritture infatti resta asincrono, si è solo negato lo spostamento di codice da parte del compilatore, e quindi un ordine diverso di produzione delle due scritture.

3.4 Implementazione dei canali con UDN

L'uso della UDN permette sia la sincronizzazione dei processi che la trasmissione dei dati tra processi, con latenze molto basse e senza l'utilizzo della memoria condivisa e quindi del sottosistema di cache. A tal fine è possibile pensare ad una associazione tra uno o più canali firmware UDN e un canale software. Esistono però alcune limitazioni imposte da tale rete:

- sono disponibili quattro canali UDN (fisici), ciascuno dei quali è associato univocamente ad una coda firmware nella CPU, utilizzata in lettura;
- è fissata la massima dimensione dei pacchetti trasmessi nella rete, la quale non può superare la dimensione delle code firmware nelle CPU.

Ne segue che, volendo utilizzare solo la rete di interconnessione senza il supporto della memoria condivisa, il numero di canali software è limitato dal numero di canali firmware e che il grado di asincronia di un canale software possa essere limitato dalla dimensione delle code firmware. In altre parole non è possibile una implementazione dei canali di comunicazione tra processi che sfrutta esclusivamente UDN e che sia generica, non solo nel grado di asincronia, ma anche nel numero di canali disponibili per processo. Per ottenere tale genericità è necessario far uso di memoria condivisa.

3.4.1 Comunicazione simmetrica

Viene descritta l'implementazione della comunicazione simmetrica che sfrutta in modo ottimale la UDN. Tale implementazione pone alcuni limiti sull'utilizzo dei canali ma fornisce le migliori prestazioni possibli riguardo allo scambio dei messaggi sfruttando la UDN. Il protocollo Rdy-Ack per un canale simmetrico viene realizzato impiegando una coda UDN in entrambi i PE che eseguono i processi comunicanti. La coda firmware nel destinatario è usata per ricevere i messaggi, il segnale di Rdy è implicito con la ricezione di un nuovo messaggio, la coda firmware nel mittente è usata per i segnali di Ack, i quali sono esplicitati con l'invio di un messaggio di valore arbitrario di dimensione una parola dal destinatario a tale coda UDN nel mittente. Un possibile scenario è mostrato in figura 3.6 dove il canale di comunicazione è implementato tramite l'utilizzo della

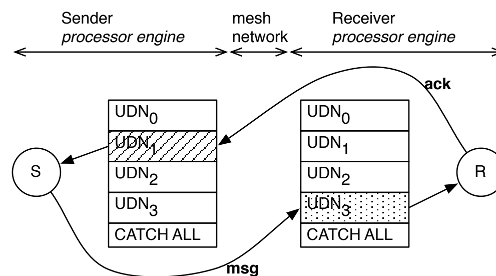


Figura 3.6: Rappresentazione di un possibile scenario di comunicazione simmetrica tra il processo mittente S e quello destinatario R, sfruttando la UDN

```
send(ch_descr, msg) ::
  leggi una parola dalla UDN Demux Queue ch_descr->dq_snd
  invia tramite UDN il valore msg alla UDN Demux Queue
    ch_descr->dq_rcv del PE ch_descr->cpu_rcv

receive(ch_descr) ::
  leggi una parola dalla UDN Demux Queue ch_descr->dq_rcv e
    assegna il valore alla variablie targa
  invia tramite UDN una parola arbitraria alla UDN Demux
    Queue ch_descr->dq_snd del PE ch_descr->cpu_snd
```

Codice 3.5: Descrizione astratta del protocollo di comunicazione Rdy-Ack su UDN per un canale di comunicazione simmetrico

seconda coda UDN nel processo mittente, per la ricezione dei segnali di Ack, e con la quarta coda UDN nel processo destinatario, per la ricezione dei messaggi; ne segue che in entrambi i PE, che eseguono esclusivamente il processo mittente o quello destinatario, rimangono disponibili altre tre code (o canali) UDN utilizzabili per altrettanti canali di comunicazione tra processi. Il protocollo di comunicazione segue quindi lo schema del codice 3.5: viene usata una struttura dati condivisa in memoria, il descrittore di canale, acceduta in sola lettura da entrambi i processi comunicanti, contenente l'identificatore delle cpu in cui i processi comunicanti sono eseguiti, e l'identificatore delle code UDN nei due PE, utilizzate dal canale per la comunicazione. La correttezza del protocollo si ottiene con due condizioni: *a*) l'inizializzazione di un canale di comunicazione Rdy-Ack prevede l'invio del segnale di Ack al mittente all'avvio dell'applicazione, in questo caso deve essere inviata una parola arbitraria alla coda UDN usata nel canale dal processo mittente, *b*) un corretto uso da parte dell'utente dei canali nell'assegnazione delle code UDN a diversi canali di comunicazione: se un canale di comunicazione fa uso di una certa coda UDN, tale coda UDN non deve essere utilizzata da altri canali o da altri meccanismi.

L'implementazione descritta limita perciò l'uso di questo tipo di canali ad un massimo di quattro canali per processo, l'implementazione bypassa completamente la memoria se non per accedere in sola lettura alle informazioni del canale, la trasmissione dei dati e la sincronizzazione dei processi/PE avviene a livello firmware grazie alla rete di interconnessione e alle primitive di accesso ad essa fornite dal sistema come letture e scritture su registri generali.

Con il grado di asincronia unitario non esistono problemi di deadlock in quanto, se verificate le condizioni di correttezza, il protocollo garantisce la corretta sincronizzazione e i dati scambiati nella rete sono lunghi una sola parola, quindi non causano l'overrun delle code UDN.

3.4.2 Comunicazione asimmetrica in ingresso

L'implementazione della comunicazione asimmetrica in ingresso deriva da quella simmetrica: un generico processo mittente del canale dispone di una coda UDN sulla quale legge gli eventi di Ack, il processo ricevente del canale usa una singola coda UDN sulla quale legge il messaggio di uno qualsiasi tra i processi mittenti. Un messaggio inviato da un mittente è costituito da una coppia di

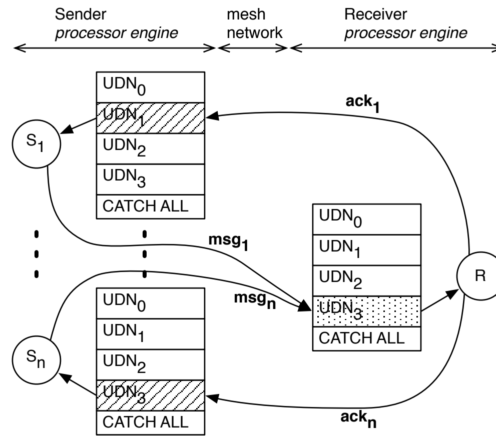


Figura 3.7: Rappresentazione di un possibile scenario di comunicazione asimmetrica in ingresso tra l'insieme di processi mittenti $\{S_1, \ldots, S_n\}$ e il processo destinatario R, sfruttando la UDN

valori: una parola di intestazione contenente l'identificatore del mittente all'interno del canale, e una seconda parola contente il valore del messaggio per il quale è stato richiesto l'invio. Il fatto che sia trasmesso anche una intestazione al messaggio è trasparente all'utente, in quanto l'esecuzione della ricezione nel processo destinatario rende visibile all'utente solo il valore del messaggio. Come mostrato in figura 3.7 diversi mittenti dello stesso canale possono avere come una coda UDN di ricezione qualsiasi, questo crea un problema simile a quello visto nell'implementazione che sfrutta la memoria condivisa nella sezione 3.3.2: durante l'esecuzione della primitiva di invio del canale, da parte di un generico mittente, deve essere noto l'identificatore del mittente all'interno del canale stesso, in modo da sapere quale coda UDN usare per l'evento Ack e che valore dare all'intestazione dei messaggi. Anche per questa implementazione del canale asimmetrico si usa un terzo parametro nella primitiva di invio: l'identificatore nel canale del mittente. Il comportamento del protocollo è descritto nel codice 3.6; come nel canale simmetrico, il supporto fa uso di un descrittore di canale contenente le informazioni sulle cpu e le code UDN usate dai processi comunicanti, nel caso dei processi mittenti tali informazioni sono organizzate in array accedute con l'identificatore del mittente all'interno del canale.

L'analisi della correttezza della comunicazione è simile a quella descritta nell'implementazione simmetrica: a) durante l'implementazione del canale devono essere inviati i messaggi di Ack a tutti i mittenti nelle relative code UDN, b) l'utente si fa carico di non usare in un certo processo, eseguito in modo esclusivo in un PE, la stessa coda UDN per più canali o compiti diversi. Si osserva inoltre che l'unità di routing di UDN è il pacchetto [7], quindi l'invio della coppia di parole $< sender_{rank},\ msg >$ da parte di un mittente non viene mai ricevuta intervallata da altre parole nella coda UDN del destinatario se l'invio è stato fatto impostando la coppia di parole come payload di un unico pacchetto. Al contrario l'invio di due pacchetti UDN per l'invio della coppia risulta non corretto per l'implementazione descritta.

```
send(ch_descr, msg, rank) ::
  leggi una parola dalla UDN Demux Queue
    ch_descr->dq_snd[rank]
  invia tramite UDN la sequenza di valori <rank, msg> alla
    UDN Demux Queue ch_descr->dq_rcv del PE
    ch_descr->cpu_rcv

receive(ch_descr) ::
  leggi una parola dalla UDN Demux Queue ch_descr->dq_rcv e
    assega il valore alla variablie sender
  leggi una parola dalla UDN Demux Queue ch_descr->dq_rcv e
    assegna il valore alla variablie targa
  invia tramite UDN una parola arbitraria alla UDN Demux
    Queue ch_descr->dq_snd[sender] del PE
    ch_descr->cpu_snd[sender]
```

Codice 3.6: Descrizione astratta del protocollo di comunicazione Rdy-Ack su UDN per un canale di comunicazione asimmetrico in ingresso

Si analizza infine la possibilità di situazioni di deadlock causate dall'uso della UDN, che come descritto nella sezione 2.1.1 è soggetta a tale problema. Anche in questo caso l'uso di un paradigma di comunicazione client-server, con un protocollo caratterizzato da grado di asincronia unitario nega il verificarsi di qualsiasi situazione di deadlock (con le ipotesi di correttezza precedenti). Il supporto della comunicazione è infatti caratterizzato da un comportamento client-server con interazione domanda-e-risposta, in cui il servente è il destinatario e i mittenti sono i clienti. Le richieste dei clienti sono i messaggi inviati dai clienti e le risposte sono i messaggi di ack inviati dal destinatario. In una computazione di questo tipo il deadlock avviene soltanto se i messaggi di risposta del servente riempono la coda di un cliente, ciò accade solo quando un cliente invia più richieste della capacità della coda UDN di memorizzare le risposte [9].

3.4.3 Comunicazioni con grado di asincronia maggiore di 1

Le due implementazioni dei canali simmetrici e asimmetrici che sfruttano UDN si prestano ad essere estese ad un grado di asincronia maggiore di uno, è infatti sufficiente inviare un numero $m > 1$ di messaggi di Ack al/i mettente/i in fase di inizializzazione del canale. In questo modo eseguendo lo stesso protocollo precedentemente definito si ha un comportamento asincrono di grado m.

Occorre porre attenzione al fatto che un grado di asincronia elevato può causare il deadlock dell'applicazione. In particolare un mittente non deve inviare più messaggi della dimensione della coda di demultiplexing UDN. Dato che in entrambe le forme di canale le risposte del destinatario hanno dimensione una parola allora il massimo grado di asincronia è la dimensione delle code UDN, per entrambi i canali.

3.4.4 Implementazione generica

Non è fornita una implementazione dei canali che sfrutti la UDN e che non limiti per ogni processo il numero di canali utilizzabili. Si descrive qui una possibile

implementazione. La limitazione fisica di un numero finito di canali UDN è superata utilizzando, con il paradigma precedente, per più canali "software", almeno una coda UDN nel mittente e nel destinatario. I diversi canali "software" sono riconosciuti mediante una intestazione ai dati scambiati contenente l'identificatore del canale. È necessario ricorrere alla memoria condivisa nel caso in cui il processo destinatario invochi la ricezione su un certo canale e i dati ricevuti dalla coda UDN siano appartenenti ad un altro canale "software", perciò si memorizzano tali dati per un utilizzo successivo e si continua a testare la coda UDN in attesa dei dati del canale usato.

Si suppone che una implementazione di questo tipo sia sempre usata insieme all'implementazione "specializzata su UDN", può essere quindi conveniente lasciare le quattro code UDN di demultiplexing a quest'ultima implementazione, e usare la coda *catch all* per l'implementazione "generica su UDN". Tale coda viene usata quando si inviano pacchetti UDN con tag di demultiplexing diverso dai quattro valori delle code firmware, la ricezione viene effettuata mediante la lettura di registri SPR.

L'uso di una implementazione "UDN generica" può essere una utile alternativa alla implementazione che sfrutta esclusivamente la memoria condivisa (descritta in 3.3) per applicazioni che necessitano di più di quattro canali per processo e basse latenze di comunicazione. Si osserva tuttavia che più sono usati molti canali di comunicazione in un singolo processo, più aumenta la probabilità di ricevere il messaggio di un canale diverso e quindi la necessità di eseguire una copia in memoria (overhead).

4 Esperimenti

Con lo scopo di stimare e confrontare le prestazioni delle due tipologie di implementazioni del supporto alle comunicazioni vengono proposti due diversi esperimenti. Il primo è volto a misurare la latenza di comunicazione dei canali, viene usato il classico metodo a scambio di messaggi, detto "ping-pong". Il secondo è la parallelizzazione di un modulo sequenziale su stream che prevede sia comunicazioni simmetriche, che almeno una comunicazione asimmetrica in ingresso; in questo caso il confronto delle due soluzioni si basa sul tempo di completamento dello stream e sulla scalabilità del sistema parallelo all'aumentare del numero dei processi. È significativa la scelta del tipo di computazione e di calcolo, in quanto le differenze prestazionali dei supporti forniti emergono con calcoli di grana fine, molto comuni in computazioni su stream. Calcoli di grana elevata, invece, non sono influenzati dalle comunicazioni: se è possibile la sovrapposizione delle comunicazioni al calcolo, le comunicazioni non hanno alcun impatto, altrimenti sono trascurabili rispetto al tempo di calcolo.

4.1 Misurazione della latenza di comunicazione

Si propone un primo confronto delle due tipologie di implementazioni del supporto alle comunicazioni effettuato sulla latenza di comunicazione in *assenza di conflitti* sia sulle reti di interconnessione che sulla memoria condivisa[1]. Per entrambe le forme di comunicazione vogliamo sapere quale è il tempo medio necessario ad eseguire completamente una comunicazione, ovvero l'intervallo di tempo medio tra l'inizio dell'esecuzione di una *send* e la copia del messaggio nella variabile targa del destinatario, avvenuta al termine dell'esecuzione della corrispondente *receive* nel destinatario. La latenza di un canale asimmetrico viene misurata con un unico mittente, così da poter essere confrontata con la latenza di comunicazione di un canale simmetrico.

Le implementazioni presentate nel Capitolo 3 sono riassunte in tabella 4.1 con i rispettivi nomi. Una misura di questo tipo è una prima risposta al quesito del lavoro di tirocinio: se esistono vantaggi prestazionali nell'uso di UDN per un supporto alle comunicazioni, e che tipo di miglioramenti si hanno rispetto ad una implementazione tradizionale. Questi risultati sono utili anche per un confronto interno alle diverse soluzioni che usano la memoria condivisa e che sono specifiche delle possibilità di configurazione della macchina TILE*Pro*64. La sezione 3.3 presenta alcune possibili scelte di gestione del sottosistema di memoria cache al fine di minimizzare gli overhead legati alla coerenza. Attraverso questa prima misura siamo quindi interessati a conoscere:

- quale è la degradazione dovuta alla gestione predefinita dell'homing delle strutture dati, senza sfruttare il paradigma consumatore-produttore,
- quale la degradazione legata all'uso della barriera di memoria necessaria con il protocollo Rdy-Ack e invece eliminata con un diverso protocollo.

[1]Non è possibile asserire l'assenza di conflitti in tali sottosistemi, tuttavia, se esistono, i conflitti sono minimi rispetto a quelli che si verificano in una applicazione reale, in quanto l'applicazione di misurazione fa uso di due soli processi che tramite il nostro supporto si sincronizzano a vicenda.

4.1.1 L'applicazione di misurazione

Al fine di valutare la latenza di comunicazione ($\mathrm{L_{com}}$) dei canali si misura il tempo di completamento ($\mathrm{T_C}$) di una applicazione di tipo "ping-pong" caratterizzato da due processi comunicanti mediante due canali con direzione una opposta a quella dell'altro. I due canali hanno stessa forma e stessa implementazione. Il comportamento di ciascun processo è definito dall'alternarsi di invio di un messaggio e ricezione di un messaggio rispettivamente sui due canali collegati al processo. Il processo che esegue per prima operazione la *send* è detto "white process", l'altro, che comincia eseguendo *receive*, è detto "black process". La sequenza temporale delle azioni eseguite dai due processi è mostrata in figura 4.1. Per $\mathrm{T_C}$ si intende il tempo impiegato per lo scambio di un numero n di messaggi, tale tempo è preso nel white process memorizzando il tempo di avvio, prima del primo invio, e il tempo di fine, dopo l'n-esima ricezione. Prima di eseguire gli n scambi i due processi effettuano uno o più scambi "non misurati" affinchè l'uso successivo dei canali trovi i blocchi del supporto già presenti in cache.

La latenza di comunicazione è approssimata come la metà del tempo medio di scambio:

$$\mathrm{T_{scambio}} = \frac{\mathrm{T_C}}{n} \qquad \mathrm{L_{com}} = \frac{\mathrm{T_{scambio}}}{2}$$

Il programma di misurazione effettua m scambi di messaggi. L'esecuzione e la corrispondente misurazione del tempo di completamento degli scambi è eseguita per diverse configurazioni di allocazione dei processi nei PE, a distanze diverse nella mesh. I valori di distanza, o *numero di hops*, considerati sono: il minimo (1), il massimo (14) e quello medio ($\sqrt{64} = 8$). Per ogni configurazione vengono eseguite n misure, ad ogni iterazione viene calcolato il valore medio della latenza di comunicazione per gli m scambi effettuati. Per ogni distanza viene presentata la media, il valore massimo e la devianza standard degli n valori medi della latenza di comunicazione.

Nella misurazione della comunicazione asimmetrica si prendono in considerazione solo le due principali implementazioni del supporto, ovvero quella che

<table>
<tr><th>Supporto architetturale</th><th>Nome dei canali di comunicazione</th><th>Descrizione</th></tr>
<tr><td rowspan="3">Memoria Condivisa</td><td><code>ch_sym_sm_rdyack_no</code></td><td>Utilizzo non ottimizzato della cache</td></tr>
<tr><td><code>ch_sym_sm_rdyack</code>
<code>ch_asymin_sm</code></td><td>Allocazione con massima località in cache</td></tr>
<tr><td><code>ch_sym_sm_nullack</code></td><td>Utilizzo del protocollo di comunicazione Null-Ack</td></tr>
<tr><td>UDN</td><td><code>ch_sym_udn</code>
<code>ch_asymin_udn</code></td><td>Uso esclusivo della rete di interconnessione</td></tr>
</table>

Tabella 4.1: Canali di comunicazione implementati con i due supporti architetturali

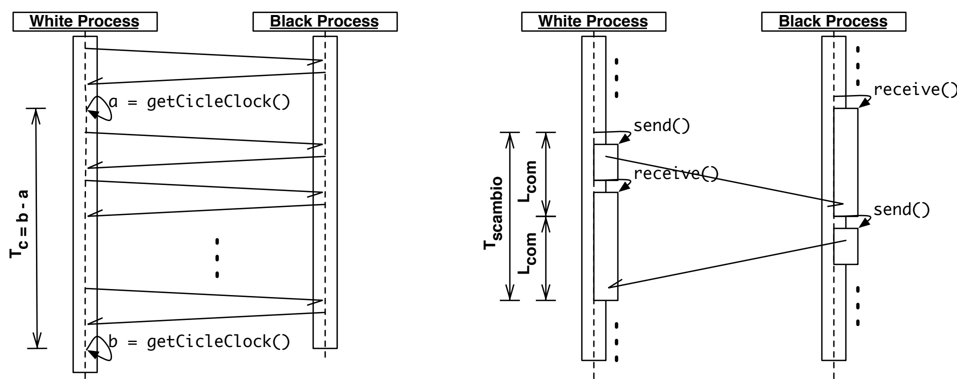


Figura 4.1: Rappresentazione della sequenza di azioni svolte dai due processi dell'applicazione di misurazione. A sinistra viene mostrato il comportamento complessivo, a destra il dettaglio di uno scambio di messaggi.

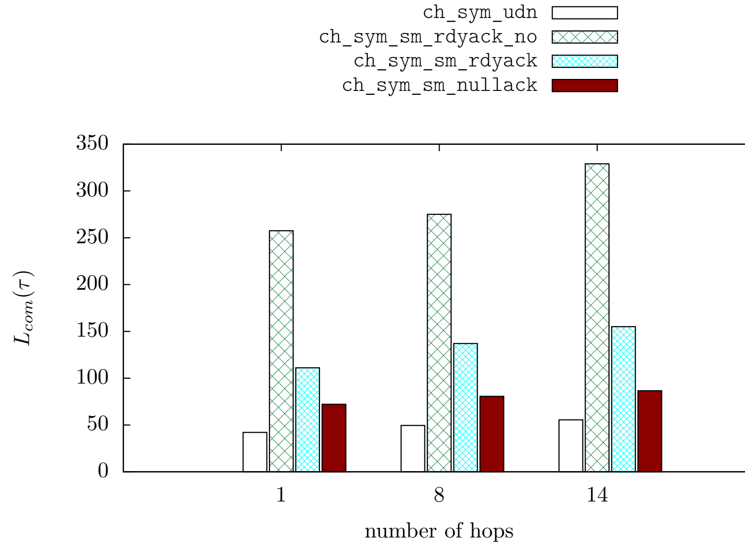
fa uso della UDN e quella che utilizza al meglio la memoria condivisa [2]. Inoltre, per la stessa forma di comunicazione e per quanto riguarda l'implementazione su memoria condivisa, viene fornita un'altra misura, chiamata `ch_asymin_sm_all`. Viene utilizzato un canale asimmetrico con il massimo numero di processi mittenti allocabili nella macchina; come in precedenza, la misura è effettuata con lo scambio di messaggi tra due soli processi. La misura è interessante in quanto la politica di ricezione del destinatario prevede la scansione del flag `Rdy` di tutti i processi mittenti. È pertanto auspicabile un aumento della latenza di comunicazione con l'aumento del numero dei mittenti, anche se solo uno di questi comunica con il destinatario. Si osserva che questo secondo modo di misurare la comunicazione asimmetrica non fornisce risultati significati per l'implementazione UDN, e per questo motivo non è mostrato nei risultati. Con l'uso di UDN, infatti, la politica di ricezione non deterministica da più mittenti è implementata nativamente dal firmware della macchina e non è soggetta a overhead (è la lettura da una coda FIFO di registri).

4.1.2 Risultati

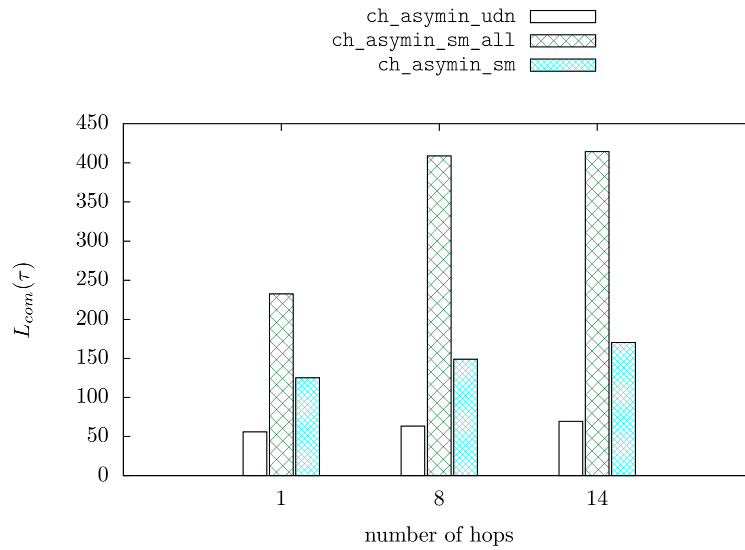
I risultati proposti in tabella 4.2 e graficamente in figura 4.2, sono relativi a 10 iterazioni del programma di misurazione con $m = 10^5$ scambi di messaggi tra i due processi.

I risultati ottenuti verificano i ragionamenti descritti nelle Sezioni 3.3, 3.4 che hanno guidato la realizzazione delle diverse implementazioni. La prima considerazione riguarda il confronto tra l'uso della UDN e della memoria condivisa: in entrambe le forme di comunicazione l'implementazione di UDN ha latenza inferiore a quella dell'implementazione con memoria condivisa più veloce. Raffiniamo questa osservazione valutando quale sia la migliore implementazione su memoria condivisa e quindi confrontiamola con l'implementazione UDN. La migliore implementazione su SM non è necessariamente quella più veloce: sebbene il protocollo Null-Ack fornisca la minor latenza, la sua implementazione fa uso

[2]Per determinare questa informazione si usano i risultati della misura delle implementazioni simmetriche, si veda la discussione nella sezione 4.1.2



(a) Latenza di comunicazione misurata delle implementazioni del canale simmetrico



(b) Latenza di comunicazione misurata delle implementazioni del canale asimmetrico in ingresso

Figura 4.2: Rappresentazione grafica dei risultati delle misurazioni della latenza di comunicazione dei canali. Il programma di misurazione è stato eseguito con un numero $m = 10^5$ di scambi e un numero $n = 10$ di iterazioni.

di un valore particolare dei messaggi e pertanto è meno generica delle altre implementazioni. Ad esempio se si richiedesse una implementazione di canali di tipo generico, l'applicazione di questo protocollo potrebbe essere problematica. L'implementazione Null-Ack è utile per conoscere quale è la degradazione introdotta dalla barriera di memoria, necessaria nel protocollo Rdy-Ack, e quindi quale è il vero overhead della nostra implementazione lock-free dei canali.

Passiamo a valutare la differenza tra le due implementazioni su SM che usano il protocollo Rdy-Ack e che differiscono per la gestione della gerarchia cache. Una gestione esplicita dell'allocazione in cache, che massimizzi la località dei dati, dimezza la latenza di comunicazione rispetto alla gestione predefinita dell'allocazione. Questo è un risultato generale, l'impostazione del PE consumatore

Implementazione	# *Hops*	L_{com}			
		Avg		*Std Dev*	*Max* (τ)
		τ	μsec		
`ch_sym_udn`	1	42.067	0.04866	0.09782	42.262
	8	49.561	0.05732	0.10419	49.770
	14	55.723	0.06445	0.10794	55.792
`ch_sym_sm_rdyack_no`	1	257.800	0.29818	0.00896	257.816
	8	275.320	0.31844	0.00669	275.328
	14	329.332	0.38091	0.01822	329.363
`ch_sym_rdyack_sm`	1	111.482	0.12894	0.18701	111.653
	8	137.378	0.15890	0.01633	137.396
	14	155.378	0.17971	0.01328	155.394
`ch_sym_sm_nullack`	1	72.360	0.08369	0.21942	72.550
	8	80.701	0.09334	0.12446	80.827
	14	86.759	0.10034	0.11820	86.826

(a) Misurazione della latenza di comunicazione simmetrica delle diverse implementazioni.

Implementazione	# *Hops*	L_{com}			
		Avg		*Std Dev*	*Max* (τ)
		τ	μsec		
`ch_asymin_udn`	1	56.0255	0.06480	0.00094	56.027
	8	63.525	0.07348	0.00098	63.527
	14	69.527	0.08042	0.00101	69.529
`ch_asymin_sm_all`	1	230.570	0.26668	0.02874	230.644
	8	408.779	0.47280	0.04786	408.847
	14	414.375	0.47928	0.02533	414.417
`ch_asymin_sm`	1	125.039	0.14462	0.00163	125.041
	8	149.041	0.17238	0.00194	149.045
	14	170.037	0.19667	0.00204	170.040

(b) Misurazione della latenza di comunicazione asimmetrica in ingresso delle diverse implementazioni.

Tabella 4.2: Misure della latenza di comunicazione rilevate iterando 10 volte il programma "ping-pong" con $m = 10^5$ scambi

come Home per il dato acceduto in computazioni *consumatore - produttore* risulta determinante per ottimizzare le prestazioni; considerazioni simili sono state valutate in contesti differenti [1]. Consideriamo quindi la `ch_sym_sm_rdyack` come implementazione candidata su memoria condivisa, che usa il protocollo Rdy-Ack. Osserviamo che il rapporto tra la latenza di questa soluzione e quella con il protocollo Null-Ack (sempre su SM) è di 1.68. Dato che le due implementazioni eseguono istruzioni simili, con la solo differenza della barriera di memoria, è ragionevole valutare la causa di questo aumento della latenza come l'esecuzione della memory fence.
Si osserva inoltre che l'implementazione Rdy-Ack è leggermente più sensibile alla variazione di distanza dei due processi, rispetto alla Null-Ack; quest'ultima ha un aumento della latenza pari a tanti cicli di clock quanto è l'aumento della distanza. Si nota che in questo aspetto la Null-Ack si comporta come la UDN.

Si considera infine il rapporto tra la soluzione su SM e quella su UDN. Il vantaggio di uso dell'implementazione UDN rispetto a quella SM è significativo: mediamente la latenza UDN è inferiore più della metà (1/2.737) di quella SM.

Il confronto dei canali asimmetrici con un singolo mittente ha risultato simile al confronto precedente. In questo caso il rapporto medio tra la latenze UDN e quella SM, 1/2.341, è leggermente maggiore. È significativo anche il confronto tra questi canali e i corrispondenti simmetrici, in quanto, nel programma di misurazione, i canali asimmetrici sono utilizzati per realizzare forme di comunicazione simmetrica. In entrambi i casi si ha un aumento della latenza di circa 10 cicli di clock. Ciò può essere spiegato dalle specifiche implementazioni parallele: con l'uso della UDN il mittente invia due parole, anzichè una del caso simmetrico; con l'uso della SM abbiamo l'uso di altre variabili, ad esempio per garantire la fairness, rispetto al caso simmetrico.
Si osserva infine che esiste una differenza sostanziale tra l'uso del canale asimmetrico su SM con un singolo mittente rispetto a quello con tutti i mittenti, ma uno solo attivo. La latenza nel secondo caso è mediamente il doppio di quella con un singolo mittente, ne segue un divario ancora più ampio con la latenza del canale su UDN.

4.2 Benchmark moltiplicazione matrice-vettore

In questa sezione viene proposto un secondo esperimento volto a verificare il comportamento delle implementazioni del supporto su SM e su UDN in una applicazione realistica, con un numero di processi parametrico e potenzialmente alto (fino al massimo numero di PE disponibili, visto l'uso del mapping esclusivo). In particolare è stata scelta una applicazione sintetizzata per mezzo di paradigmi di parallelismo che faccia uso sia di comunicazioni simmetriche e di almeno una comunicazione asimmetrica in ingresso. Per mostrare le differenze prestazionali delle due implementazioni la computazione scelta ha calcoli di grana fine e prevede l'elaborazione di dati su stream. Per lo stesso programma sono state eseguite due versioni che utilizzano rispettivamente: solo il supporto alle comunicazioni su SM, e solo il supporto alle comunicazioni su UDN. La principale metrica per il confronto di queste due esecuzioni è il tempo di completamento della computazione al variare della dimensione dei dati e del numero di processi usati.

4.2.1 Descrizione del problema

Il benchmark proposto si basa su un calcolo numerico di algebra lineare molto frequente nella risoluzione di problemi reali, non solo in ambito scientifico: il prodotto matrice per vettore. Sia $\mathbf{A} = (a_{ij})_{i=1,\dots,\mathrm{M},j=1,\dots,\mathrm{M}} \in \mathbb{Z}^{\mathrm{MxM}}$ una matrice di interi di dimensione MxM, e sia $\mathbf{b} = (b_1, \dots, b_{\mathrm{M}}) \in \mathbb{Z}^{\mathrm{M}}$ un vettore di M interi, il risultato dell'operazione prodotto matrice per vettore è un vettore $\mathbf{c} = (c_1, \dots, c_{\mathrm{M}}) = \mathbf{A} \cdot \mathbf{b} \in \mathbb{Z}^{\mathbb{M}}$ la cui componente i-esima è il prodotto scalare tra la riga i-esima di $\mathbf{A}$ e il vettore $\mathbf{b}$.

$$\forall\, i \in \{1, \dots, \mathrm{M}\} \ : \ c_i = \mathbf{a}_i \cdot \mathbf{b} = \sum_{j=1}^{\mathrm{M}} a_{ij} \cdot b_j$$

Dove $\mathbf{a}_i = (a_{i1}, \dots, a_{i\mathrm{M}})$ è la riga i-esima della matrice $\mathbf{A}$. Da questa definizione segue direttamente l'algoritmo sequenziale del calcolo, che è descritto dal codice 4.1.

La computazione considerata riguarda un sistema Σ modellato come un grafo aciclico contenente un modulo di elaborazione collegato a due stream, uno di ingresso e l'altro di uscita. Lo stream di ingresso è composto da m matrici di interi di dimensione fissata MxM e con un tempo medio di interarrivo $\mathrm{T_A}$, lo stream di uscita invece trasporta i risultati del calcolo eseguito dal modulo. Su ogni elemento dello stream il modulo calcola la moltiplicazione matrice per vettore utilizzando l'elemento stesso come primo operando e un vettore $\mathbf{b}$, costante per tutta la durata dell'applicazione, come secondo operando. Il vettore

```
int A[M][M], b[M], c[M];
for i:=0 to M-1 do
  c[i] := 0;
  for j:=0 to M-1 do
    c[i] := A[i][j]*b[j] + c[i];
```

Codice 4.1: Algoritmo sequenziale del calcolo matrice per vettore

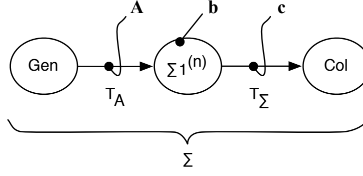


Figura 4.3: Rappresentazione compatta del grafo della computazione. L'elemento $\Sigma 1$ è un modulo sequenziale o un sottosistema parallelo che esegue la moltiplicazione per **b** su ogni elemento della matrice.

risultato di ogni operazione viene scritto nello stream di uscita. Ci poniamo nel caso in cui il tempo di calcolo di una moltiplicazione matrice per vettore, $\mathrm{T_{calc}}$, sia superiore al tempo di interarrivo $\mathrm{T_A}$. Dato che il tempo di servizio della computazione Σ è superiore a quello ideale (il tempo di interarrivo dello stream [3]), si richiede una trasformazione del modulo sequenziale in un sottosistema parallelo funzionalmente equivalente che permetta di eliminare o ridurre il collo di bottiglia nel sistema, determinato dal modulo stesso, e mantenga massima l'efficienza del sottosistema. Si indica con $\Sigma 1^{(n)}$ il sottosistema parallelo di calcolo, con grado di parallelismo n. Dal punto di vista teorico e indipendentemente dalla soluzione parallela scelta, si ha che il tempo di servizio *ideale* del sottosistema parallelo è il rapporto tra tempo di calcolo del programma sequenziale e il grado di parallelismo (equazione 1). Il tempo di servizio effettivo del sottosistema è invece il massimo valore tra il tempo medio di interarrivo dello stream e il tempo di servizio ideale del sottosistema (equazione 2). Si definisce *efficienza* del sottosistema il rapporto tra i tempi di servizio ideale ed effettivo (equazione 3). Ne segue che il sottosistema ha efficienza massima se e solo se nella computazione rimane il collo di bottiglia, ovvero il grado di parallelismo *non* è sufficientemente alto affinchè il tempo di servizio ideale sia minore o uguale del tempo di interarrivo. Altrimenti, se il collo di bottiglia è stato eliminato, l'efficienza non è massima e il suo valore corrisponde al *fattore di utilizzazione* del sottosistema, ovvero il rapporto tra il tempo di servizio ideale del sottosistema e il tempo di interarrivo. Da tale caratterizzazione ne segue il valore ottimo del grado di parallelismo, ovvero quel valore di n che consente di eliminare il collo di bottiglia e mantenere massima l'efficienza del sottosistema (equazione 4).

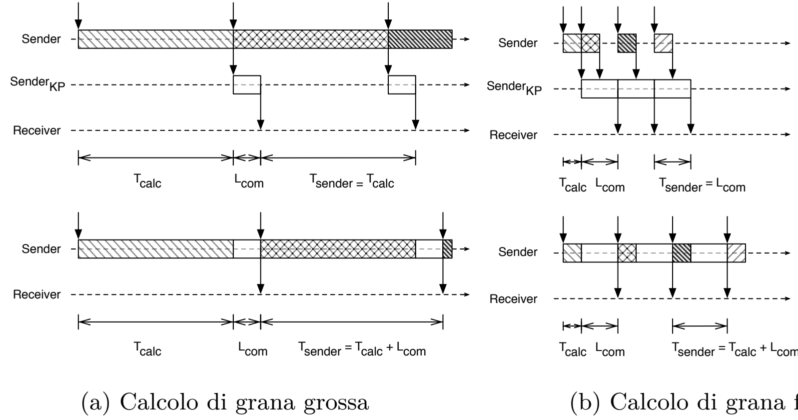
$$\mathrm{T_{\Sigma 1_id}}^{(n)} = \mathrm{T_S}^{(n)} = \frac{\mathrm{T_{calc}}}{n} \tag{1}$$

$$\mathrm{T_{\Sigma 1}}^{(n)} = \max(\{\mathrm{T_A},\ \mathrm{T_S}^{(n)}\}) \tag{2}$$

$$\varepsilon^{(n)} = \frac{\mathrm{T_{\Sigma 1_id}}^{(n)}}{\mathrm{T_{\Sigma 1}}^{(n)}} = \begin{cases} \frac{\mathrm{T_S}^{(n)}}{\mathrm{T_A}}, & \mathrm{T_S}^{(n)} < \mathrm{T_A} \\ 1, & \mathrm{T_S}^{(n)} \geq \mathrm{T_A} \end{cases} \in (0, \ldots, 1] \tag{3}$$

$$n_{\mathrm{opt}} = \min(\{n \in \mathbb{N} \mid \mathrm{T_S}^{(n)} \leq \mathrm{T_A}\}) = \left\lceil \frac{\mathrm{T_{calc}}}{\mathrm{T_A}} \right\rceil \tag{4}$$

[3] Per una trattazione formale della valutazione delle prestazioni di computazioni su stream in grafi aciclici si consulti [12]



(a) Calcolo di grana grossa

(b) Calcolo di grana fine

Figura 4.4: Rappresentazione grafica di due computazioni con diversa "grana" di calcolo, in presenza e in assenza di un processore di comunicazione

Altre definizioni utili per caratterizzare le prestazioni dell'applicazione sono le seguenti:

Tempo di completamento dello stream è definita come il tempo medio impiegato per completare l'esecuzione del calcolo su tutti gli elementi dello stream. Se la lunghezza dello stream m è molto superiore al grado di parallelismo n è possibile approssimarlo come m volte il tempo medio di servizio del sistema

$$m >> n \quad \Rightarrow \quad \mathrm{T_C} = m \cdot \mathrm{T}_\Sigma \tag{5}$$

Scalabilità esprime il risparmio relativo del tempo medio di servizio che può essere ottenuto usando l'implementazione parallela con n processi rispetto all'implementazione sequenziale. Può essere espressa anche in termini del tempo di completamento.

$$s^{(n)} = \frac{\mathrm{T_{calc}}}{\mathrm{T}_\Sigma} \tag{6}$$

$$s_{\mathrm{id}}{}^{(n)} = \frac{\mathrm{T_{calc}}}{\mathrm{T}_{\Sigma_\mathrm{id}}} = \frac{\mathrm{T_{calc}}}{\frac{\mathrm{T_{calc}}}{n}} = n \tag{7}$$

4.2.2 Sul problema scelto

È importante soffermarci e motivare il tipo di applicazione scelta: il supporto alle comunicazioni fornito è specifico per un dominio applicativo caratterizzato da grana di calcolo fine, se non finissima. Solo in computazioni di questo tipo siamo interessati a ridurre la latenza di comunicazione, e si osservano quindi delle differenze nelle prestazioni globali dell'applicazione. A scopo esemplificativo si considera una parte di una applicazione su stream riguardante due moduli collegati in pipeline: si assume che il tempo di calcolo del primo modulo sia opportunamente dimensionato come il tempo di interarrivo dello stream. Le

figure 4.4a, 4.4b mostrano due situazioni con diversa grana di calcolo del primo modulo. Se il tempo di calcolo è di diversi ordini di grandezza superiore alla latenza di comunicazione, l'ottimizzazione di qualche centinaio di cicli di clock della comunicazione non consegue alcun vantaggio sul tempo di servizio effettivo del modulo; la comunicazione avrà sempre impatto nullo o trascurabile nel tempo di servizio effettivo del modulo. Se è possibile la sovrapposizione del calcolo infatti la latenza di comunicazione è completamente mascherata dal tempo di calcolo, altrimenti, la latenza di comunicazione si va a sommare al tempo di calcolo, risultando trascurabile. L'ottimizzazione delle comunicazioni è invece apprezzabile quando la latenza di queste è dello stesso ordine di grandezza del tempo di calcolo.

Computazioni di grana fine sono molto frequenti nel dominio delle applicazioni su stream di elementi, si pensi a forme di deep packet inspection nel campo della computer networking, nelle quali vengono esaminati specifiche parti di pacchetti, che passano da un punto di ispezione di una rete di computer, alla ricerca di non conformità, o per collezionare statistiche sulle informazioni trasmesse. Computazioni su dato singolo, in special modo quelle che seguono il paradigma Data Parallel sono caratterizzate da grana fine delle computazioni, in seguito al partizionamento del dato. Anche in queste applicazioni è utile disporre di un supporto efficiente alle comunicazioni, sia per le forme di comunicazione collettiva, che per le comunicazioni tra moduli in caso di dipendenze sui dati (forme Stencil).

Il Benchmark è stato scelto nell'ottica di apprezzare le differenze tra le due realizzazioni del supporto e di verificare il loro comportamento in una applicazione reale. Le due implementazioni del supporto non differiscono solo dal conseguimento di una diversa latenza di comunicazione, come mostrato nell'esperimento precedente. La versione UDN permette un disaccoppiamento tra la comunicazione dei processi e l'accesso ai dati in memoria. La comunicazione tra due processi è realizzata interamente con l'uso della UDN e senza l'impiego della memoria condivisa. Ciò ha come effetto un numero di richieste alla memoria condivisa minore rispetto all'implementazione dei canali con la memoria. Ci aspettiamo che questo fatto abbia ripercussioni positive sulle prestazioni globali dell'applicazione, non solo per quanto riguarda le comunicazioni, ma in più in generale, relativamente al tempo di risposta delle richieste alla memoria.

4.2.3 Il metodo di parallelizzazione scelto

La parallelizzazione del modulo sequenziale è stata scelta tra un insieme ristretto (per il tipo di computazione) di paradigmi la cui semantica e modello dei costi sono noti [12], tra cui:

Farm è caratterizzata dalla replicazione della funzione di calcolo sequenziale in n identiche unità di elaborazione (sottoinsieme delle unità worker). È applicabile solo a computazioni su stream, viene usata una unità emettitore per la distribuzione di un elemento dello stream ad un worker, e una unità collettore, collegata ad ogni worker, per ricevere i risultati;

Data Parallel sono forme di parallelismo che richiedono una discreta conoscenza del calcolo sequenziale ma che per questo motivo risultano flessibili alla specifica computazione. Si basano sul partizionamento dei dati e sulla replicazione della funzione di calcolo nelle unità worker, possono

essere applicate sia a singoli dati che a stream di dati; in quest'ultimo caso il partizionamento è applicato ad ogni elemento dello stream. I gradi di libertà forniti riguardano la strategia di partizionamento dei dati, l'eventuale replicazione di dati, l'organizzazione delle unità worker (indipendenti o interagenti) e il collezionamento dei risultati parziali. Vengono usate forme di comunicazione collettiva per distribuire le partizioni del dato (*scatter*), per collezionare i risultati parziali (*gather*) e/o per costruire il risultato (ad esempio l'operazione di *reduce*). A seconda dell'organizzazione dei workers si distinguono due famiglie di forme Data Parallel: *Map* se ogni worker esegue un calcolo completamente indipendente da quello degli altri worker, *Stencil-based* se esiste un'interazione tra i workers durante l'esecuzione del calcolo.

Si è deciso di adottare un paradigma di tipo Data Parallel così da poter applicare il supporto alle comunicazioni per la realizzazione delle comunicazioni collettive coinvolte, in modo da confrontare le prestazioni dei due tipi di supporto forniti anche relativamente all'implementazione di tali comunicazioni. In particolare, il collezionamento dei risultati sarà realizzato per mezzo di un canale asimmetrico in ingresso che ha come mittenti l'insieme dei processi worker dell'implementazione Data Parallel, e come destinatario un processo collettore. Come spiegato in seguito si rende necessaria una distribuzione di ciascun elemento dello stream ai processi worker, piuttosto che una distribuzione delle partizioni di ogni elemento. L'implementazione di questa operazione fa uso dei canali simmetrici ed è caratterizzata da una struttura ad albero mappata sull'insieme dei worker.

Analizzando le dipendenze sui dati del programma sequenziale (codice 4.1) si osserva che una qualsiasi istruzione di una certa iterazione del ciclo esterno è indipendente da tutte le istruzioni di un'altra qualsiasi iterazione dello stesso ciclo. Al contrario ogni istruzione di una iterazione del ciclo interno dipende (*Read-After-Write*) dall'istruzione precedente della stessa iterazione.

```
...
{ c[i]:=0; c[i]:=A[i][0]*b[0]+c[i]; c[i]:=A[i][1]*b[1]+c[i];
      ... c[i]:=A[i][M-1]*b[M-1]+c[i]; }
{ c[i+1]:=0; c[i+1]:=A[i+1][0]*b[0]+c[i+1]; ... c[i+1]:=A[i
    +1][M-1]*b[M-1]+c[i+1]; }
...
```

Codice 4.2: Rappresentazione delle istruzioni di due cicli esterni contigui.

Ne segue che può essere esplicitato del parallelismo sul ciclo esterno, eseguendo iterazioni diverse del ciclo esterno in processi diversi, ottenendo in tal modo un grado massimo di parallelismo $n = \mathrm{M}$ e una complessità di esecuzione $O(n)$ contro $O(n^2)$ del calcolo sequenziale. Da ciò deriva direttamente il partizionamento delle matrici, che è per righe. In tale situazione si ha la "grana" minima sia per quanto riguarda le partizioni dei dati, che per il tempo di calcolo. In generale, un'implementazione effettiva fa uso di un grado di parallelismo n minore di M, dipendentemente dal valore medio del tempo di interarrivo dello stream. In tale situazione le matrici sono sempre partizionate per riga, con grana $g = \lceil M/n \rceil$ righe, il calcolo dei processi worker consiste di g prodotti scalari tra ogni riga della partizione associata al worker e il vettore $\mathbf{b}$.

Si adotta la replicazione del vettore $\mathbf{b}$ nei processi worker, ciò è possibile in quanto tale oggetto viene acceduto in sola lettura. Di conseguenza l'implemen-

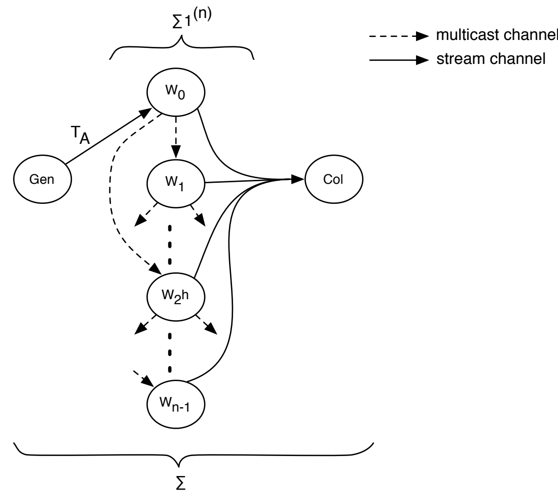


Figura 4.5: Grafo della computazione Map ($\Sigma 1^{(n)}$) che è collegata allo stream e che fa uso della multicast strutturata ad albero binario mappato sull'insieme di processi worker

tazione Data Parallel descritta è di tipo *Map*.

Rispetto a quanto detto finora il sottosistema Map è caratterizzato da una comunicazione di *multicast* piuttosto che dalla scatter; infatti si utilizza il supporto alle comunicazioni descritto nella sezione 3 per realizzare gli archi del grafo, conseguentemente abbiamo il sottografo Map operante su uno stream di riferimenti. Lo stream di ingresso trasporterà i riferimenti alle matrici, la distribuzione di un elemento dello stream ai moduli della Map è quindi l'invio dello stesso riferimento agli n moduli; sarà poi dovere di ogni worker eseguire il calcolo sulla propria partizione dell'oggetto riferito dal puntatore ricevuto in ingresso.

L'implementazione più semplice della distribuzione multicast fa uso di un singolo processo distributore che in modo sequenziale esegue l'invio del dato ad ogni processo worker. Questa soluzione è critica per comunicazioni di grana fine, in quanto può diventare rapidamente un collo di bottiglia all'aumentare del grado di parallelismo. Considerato l'ambito di applicazioni in si pone il supporto è lecito aspettarsi stream caratterizzati da elevata banda, quindi la necessità di disporre di comunicazioni collettive (tra cui la multicast) efficienti, che non siano collo di bottiglia per l'applicazione. Implementazioni della multicast adeguate a questo contesto hanno tempo di servizio costante, ad esempio implementano il processo distributore come un sottosistema parallelo strutturato ad albero. In questo caso, grazie all'effetto-pipeline, il tempo di servizio è pari alla latenza di comunicazione di un canale simmetrico per l'arietà dell'albero. Ciò richiedere tuttavia nodi aggiuntivi a quelli del sottosistema parallelo. Una soluzione alternativa che mantenga lo stesso tempo di servizio senza moduli addizionali consiste nell'implementare l'albero in modo distribuito direttamente nell'insieme dei moduli worker. In questo modo, un worker, prima di avviare il proprio

calcolo su un elemento dello stream prende parte alla comunicazione multicast dell'elemento stesso realizzata in modo distribuito sull'insieme dei worker. Questa soluzione (per un tempo di interarrivo superiore a 2 volte il tempo di servizio della multicast) permette di risparmiare nodi di elaborazione rispetto ad una soluzione con un sottosistema di nodi specializzato nella distribuzione multicast. Di contro, una realizzazione di questo tipo fa pagare il proprio tempo di servizio all'interno del tempo del sottosistema di calcolo nel caso in cui non sia possibile sovrapporre le comunicazioni al calcolo, come accade in TILE*Pro*64 che non dispone dei supporti architetturali necessari nei PEs per l'esecuzione delle comunicazioni[4].

La formulazione del tempo di servizio ideale e del grado di parallelismo ottimo del sottosistema Map differisce da quella definita precedentemente, in quanto occorre considerare che le comunicazioni non sono sovrapposte al calcolo. Di conseguenza nel tempo di servizio del sottosistema di calcolo si pagano completamente i tempi di servizio della multicast e della gather (equazione 8), chiamiamo con Δ_{com}la somma di tali tempi. Anche il grado ottimo di parallelismo è rivalutato nella equazione 9.

$$\begin{aligned} \mathrm{T}_{\Sigma 1_\mathrm{id}}{}^{(n)} &= \mathrm{T}_{\mathrm{S}}{}^{(n)} = \mathrm{T}_{\mathrm{multicast}} + \frac{\mathrm{T}_{\mathrm{calc}}}{n} + \mathrm{T}_{\mathrm{gather}} \\ &= 2 \cdot \mathrm{T}_{\mathrm{sym_send}} + \frac{\mathrm{T}_{\mathrm{calc}}}{n} + \mathrm{T}_{\mathrm{asymin_send}} = \Delta_{\mathrm{com}} + \frac{\mathrm{T}_{\mathrm{calc}}}{n} \qquad (8) \\ n_{\mathrm{opt}} &= \left\lceil \frac{\mathrm{T}_{\mathrm{calc}}}{\mathrm{T}_{\mathrm{A}} - \Delta_{\mathrm{com}}} \right\rceil \qquad (9) \end{aligned}$$

Come ulteriore conseguenza, applicando la definizione di scalabilità al tempo di servizio, non è possibile ottenere il valore ideale pari al grado di parallelismo usato.

4.2.4 Sulla latenza di accesso alla memoria

I valori di $\mathrm{T}_{\mathrm{calc}}$, $\Delta_{\mathrm{com}}(\mathrm{SM})$ e $\Delta_{\mathrm{com}}(\mathrm{UDN})$ non sono costanti, ma variano con il grado di parallelismo usato. All'aumentare del numero di processi coinvolti nell'applicazione aumentano il numero di richieste alla memoria condivisa e quindi aumenta l'uso delle reti di interconnessione e della gerarchia di memoria, con possibili congestioni, e in generale un aumento del tempo di risposta. In particolare sia $\mathrm{T}_{\mathrm{calc}}$ che $\Delta_{\mathrm{com}}(\mathrm{SM})$ dipendono dalla latenza di accesso alla memoria e al sottosistema di cache, in quanto, in caso di *fault* dei dati nella cache locale o nella cache home, è necessario trasferire il blocco corrispondente da un'altra cache o da un controllore di memoria, rispettivamente. Formalmente il problema può essere modellando come un sistema *client-server*: un controllore di memoria, o una L2 cache, è considerato il modulo servente di un certo insieme di moduli clienti: il sottoinsieme dei PE che producono le richieste di accesso alla memoria a quel preciso modulo. Il tempo medio di risposta, R_{Q}, del servente è quindi caratterizzato dal periodo di tempo che la richiesta trascorre nella coda del servente, $\mathrm{W}_{\mathrm{Q}}(\rho)$, più la latenza di elaborazione del servente. Essendo la computazione *domanda e risposta* il fattore di utilizzazione ρ del servente

[4]Una forma di sovrapposizione esiste quando si fa uso del supporto alle comunicazioni che usa UDN, è infatti possibile che il trasferimento del messaggio sia eseguito in modo parzialmente sovrapposto all'esecuzione del calcolo del mittente.

è sempre inferiore ad 1. Al variare del fattore di utilizzazione del servente il tempo di risposta R_Q si mantiene pressocchè costante; dopo un certo valore, detto critico, di ρ, R_Q aumenta asintoticamente ad infinito con ρ che tende ad 1 (congestione). Il tempo di servizio effettivo di ogni cliente può essere caratterizzato dalla somma del tempo medio di calcolo più il R_Q. L'aumento della banda di richieste al servente, quindi l'aumento del fattore di utilizzazione ρ, e quindi l'aumento del tempo di risposta R_Q può essere causato dalla riduzione del tempo (grana) di calcolo nei clienti, oppure dall'aumento del numero di clienti. In [11] è presentata una analisi della latenza di accesso alla memoria con e senza conflitti, e relativi modelli dei consti.

4.2.5 Descrizione dell'implementazione

L'applicazione considerata è fittizia, ovvero non esiste un dispositivo, o processing element, che produce lo stream, come non esiste quello che lo riceve. Al fine di poter eseguire l'applicazione sono stati realizzati due processi, eseguiti in modo esclusivo su due PE della macchina, con il compito di produrre e consumare gli stream di matrici e di vettori rispettivamente. Questi processi sono collegati al sottosistema parallelo per mezzo dei canali messi a disposizione dal nostro supporto. Ne segue che il massimo parallelismo esplicitabile dalla macchina per la realizzazione della Map è $N = 59$, occorre infatti riservare un altro PE all'esecuzione del processo `main` dell'applicazione, e due PE sono riservati per l'esecuzione di funzionalità del sistema operativo.

Il processo generatore dello stream è inizializzato con un tempo di interarrivo parametrico. La temporizzazione dello stream viene realizzata da questo processo invocando la primitiva di conteggio dei cicli di clock ed effettuando una attesa attiva sul valore del tempo trascorso, fin tanto che è minore al tempo di interarrivo specificato. Il processo generatore è collegato al primo processo worker del Map che costituisce il processo radice dell'albero che realizza la multicast. È compito di tale worker avviare la multicast distribuita sull'insieme dei worker per mezzo dei canali che costituiscono la struttura ad albero della comunicazione. Ogni processo worker è quindi collegato al processo collettore per mezzo di un canale asimmetrico in ingresso.

Il mapping dei processi nella macchina adottato consiste nell'eseguire i processi generatore e collettore nei primi due PE, i processi worker sono mappati in modo consecutivo a partire dal terzo PE, il processo main è mappato sull'ultimo PE disponibile. La topologia della comunicazione multicast si basa sull'applicazione della strategia *depth-first* applicata al mapping di un albero binario nella sequenza lineare dei processi worker.

L'applicazione è eseguita in modo parametrico nelle seguenti variabili:

- il tempo di interarrivo (T_A) in cicli di clock,
- la dimensione della matrice (il numero di righe M),
- il grado di parallelismo del sottosistema Map (n),
- la lunghezza dello stream (m),
- l'implementazione adottata dai canali simmetrici e dai canali asimmetrici in ingresso.

Si sono eseguite le misurazioni per due diverse configurazioni di implementazioni del supporto alle comunicazioni: utilizzando solo UDN, oppure solo la memoria condivisa. In tutte le esecuzioni si è adottata una lunghezza dello stream $m = 500$ sufficientemente maggiore al massimo grado di parallelismo esplicitabile (circa 60) affinchè sia possibile usare l'approssimazione dell'equazione 5.

Le misure proposte nel seguito non sono mai relative ad una singola esecuzione, ma sono valori medi delle misure effettuate in un certo numero di esecuzioni sullo stesso tipo di computazione.

4.2.6 Descrizione del metodo di misurazione

Le misure prese in considerazione sono il tempo di completamento dello stream e il tempo di servizio del sottosistema Map. Tutti i processi dell'applicazione si sincronizzano su un oggetto di tipo barriera, il tempo di completamento viene avviato dopo che il processo generatore ha passato questa barriera e fermato al termine dell'esecuzione del processo collettore. Il tempo di servizio della Map è misurato nel processo worker alla radice dell'albero multicast ed è il risultato della media di tutti i tempi impiegati da tale processo tra la ricezione di due elementi contigui nello stream.

Durante l'analisi saremo interessati a caratterizzare le presentazioni del sistema sotto altri punti di vista:

- il tempo di calcolo di un singolo prodotto scalare eseguito da un processo worker: per stimare questo valore viene misurato il tempo di calcolo di ogni singolo worker per ogni elemento dello stream, il tempo medio di calcolo viene quindi diviso per la dimensione della partizione (numero di righe). La misura è avviata in ogni worker dopo aver concluso la partecipazione alla multicast e viene conclusa prima di inviare il risultato al collettore.

- il tempo di servizio della multicast: è stimato come media delle misure del tempo impiegato dal worker alla radice dell'albero mulitcast per eseguire l'invio ai due sotto-alberi.

4.2.7 Risultati delle misurazioni

Considerato che siamo interessati a computazioni di grana fine si sono considerare matrici con un numero di righe relativamente basso: $\mathrm{M} \in \{ 56, 168, 280 \}$. Con il massimo grado di parallelismo, infatti, abbiamo partizioni di dimensione $g \in \{ 1, 3, 5 \}$ righe, rispettivamente; conseguentemente, il calcolo svolto nei worker in tale situazione prevede $g \cdot \mathrm{M}$ moltiplicazioni e altrettante somme. Per ogni dimensione di matrice si sono eseguite istanze del programma di benchmark con grado di parallelismo della Map variabile. In ogni caso si è scelto n divisore di M.

Dato che non è imposto il tempo di interarrivo, si vorrebbe stimare il tempo di servizio della Map con il massimo numero di nodi disponibili, in modo tale da conoscere quale è la banda massima dello stream in cui il nostro sottosistema parallelo effettua il calcolo, producendo i risultati con la stessa velocità. L'equazione da usare è quella mostrata precedentemente in formula 8. Se invece fosse noto il tempo di interarrivo dello stream, si userebbe l'equazione di formula 9 per ricavare il minimo numero di nodi worker necessari per eliminare il collo di bottiglia. Una previsione effettiva del comportamento del sistema stima i

valori di T_{calc} e Δ_{com}per mezzo di simulazioni o applicando con modelli di costo formali che caratterizzano vari aspetti dell'architettura.

L'applicazione di modelli di costi all'architettura TILE*Pro*64 non è lo scopo del tirocinio, si è quindi usato un approccio molto più semplice: è stimato in modo rudimentale il tempo di servizio ideale sostituendo nella formula 8, al T_{calc} la misura del tempo di completamento effettuata sul programma sequenziale eseguito nella CPU di un singolo PE, e al Δ_{com} la misura della latenza di comunicazione descritta nell'esperimento precedente (sezione 4.1.2). Quello che otteniamo è una valutazione molto elementare del tempo di servizio che difficilmente sarà riscontrata nell'esecuzione del programma, perché, ad esempio, non si è tenuto conto dell'aumento della latenza di accesso alla memoria con l'incremento del grado di parallelismo. Nelle tabelle 4.3a, 4.3b sono riassunte le misure dei due parametri; in tabella 4.3c sono presentati i valori calcolati del tempo di servizio con il massimo grado di parallelismo ($N = 59$). Il tempo di servizio effettivo del sistema viene mostrato nella sezione seguente.

Dimensione matrice	T_{calc}	
	τ	μsec
56x56	82934.927	95.924692
168x128	735319.293	850.489412
280x280	2040430.561	2360.015038

(a) Misurazioni del tempi di elaborazione del programma di calcolo sequenziale.

	Implementazione usata		Δ_{com}	
	canale simmetrico	*canale asimmetrico*	τ	μsec
UDN	`ch_sym_udn`	`ch_asymin_udn`	180.95523	0.20929
	`ch_sym_sm`	`ch_asymin_sm`	481.04016	0.55638
SM	`ch_sym_sm`	`ch_asymin_sm_all`	724.91273	0.83845

(b) Stima di Δ_{com} per mezzo delle misure della latenza dei canali effettuate in sezione 4.1.2 con i due processi distanti 14 hops.

M	${T_{\Sigma 1_id}}^{(n)}$(UDN)		${T_{\Sigma 1}}^{(n)}$(SM)	
	τ	μsec	τ	μsec
56	1586.632	1.835	2130.589	2.464
168	12643.994	14.624	13187.952	15.254
280	34764.524	40.210	35308.482	40.839

(c) Valori del tempo di servizio ideali della Map con grado di parallelismo massimo ($N = 59$). Sono ottenuti sostituendo i valori delle tabelle 4.3a, 4.3b alla formula 8.

Tabella 4.3

Misura dei tempi di completamento e di servizio al variare del tempo di ingresso

Si sono sperimentate più esecuzioni dell'applicazione con diversi valori del tempo di interarrivo dello stream:

1. viene considerata una banda di arrivi per la quale non si riesce ad eliminare il collo di bottiglia; in particolare questo esperimento ci mostra qual'è il tempo di servizio effettivo che assume il sottosistema;
2. Una volta noto il tempo di servizio effettivo si considerano velocità degli arrivi superiori a questo valore, per i quali deve essere eliminato il collo di bottiglia. Scopo di questi esperimenti è verificare che il collo di bottiglia sia effettivamente eliminato, e con che grado di parallelismo n si riesce ad eliminare, in particolare se il valore di questo n è vicino a quello stimato con la formula 9.

I risultati degli esperimenti fatti sono mostrati in figura 4.6 per quanto riguarda il tempo di servizio della Map, e in figura 4.7 per il tempo di completamento dello stream. La scalabilità è calcolata con la formula 6 sostituendo al tempo di calcolo, quello misurato sul programma sequenziale e presentato in tabella 4.3a. I grafici di scalabilità sono quindi mostrati in figura 4.8. Si ricorda che il tempo di servizio effettivo della Map è il massimo tra il tempo di interarrivo e il tempo di servizio ideale, costituito, quest'ultimo, dal tempo di calcolo sulla partizione della matrice, più Δ_{com} il tempo speso nelle comunicazioni. Per un generico tempo di interarrivo abbiamo due possibili soluzioni:

la Map è collo di bottiglia: è interessante verificare quale sia il tempo di interarrivo effettivo della Map, e di conseguenza la massima scalabilità effettiva della Map. In precedenza si è infatti osservato che le formule usate per stimare il tempo di servizio della Map non applicano alcun modello dei costi relativo ad aspetti dell'architettura, ma si limitano ad usare le misure su computazioni particolari. In seguito ai conflitti sulle reti di interconnessione e sui sottosistemi della gerarchia di memoria si prevede un aumento del tempo di calcolo nei worker, rispetto a $\frac{T_{\text{calc}}}{n}$, e un aumento della latenza di comunicazione, rispetto a quella misurata. Applicando la formula del n_{opt} ai valori effettivi del tempo di calcolo e della latenza di comunicazione si ottiene un valore maggiore rispetto a quello stimato inizialmente. La tabella 4.4 mostra i tempi di servizio effettivi, per le varie dimensioni delle matrici. Questi sono molto superiori ai tempi di servizio ideali relativi al massimo n esplicitabile (59), mostrati in tabella 4.3c.

la Map non è collo di bottiglia: esiste un certo grado di parallelismo, esplicitabile dalla macchina, per cui il tempo di servizio ideale della Map è inferiore al tempo di interarrivo. Indipendentemente da un ulteriore aumento di n, il tempo di servizio effettivo del sistema rimane costante al tempo di interarrivo e la scalabilità rimane sempre uguale al grado di parallelismo ottimo. È possibile che con l'aumentare del grado di parallelismo si creino delle degradazioni per cui il tempo di interarrivo e la scalabilità diminuiscono leggermente rispetto a quelli dell'n ottimo.

Le misure effettuate mostrano che il tempo di servizio effettivo del sistema raggiunge il tempo di interarrivo con un valore di n "vicino a quello ottimo". Stesso ragionamento si applica alla scalabilità, la quale raggiunge il valore ottimo di n con un grado di parallelismo vicino a questo valore. I valori misurati per alcuni tempi di interarrivo sono mostrati nella tabella 4.5. Il comportamento del sistema è quindi quello atteso. Si osserva inoltre che la stima del grado di parallelismo ottimo è tanto migliore quanto la dimensione della matrice è più grande.

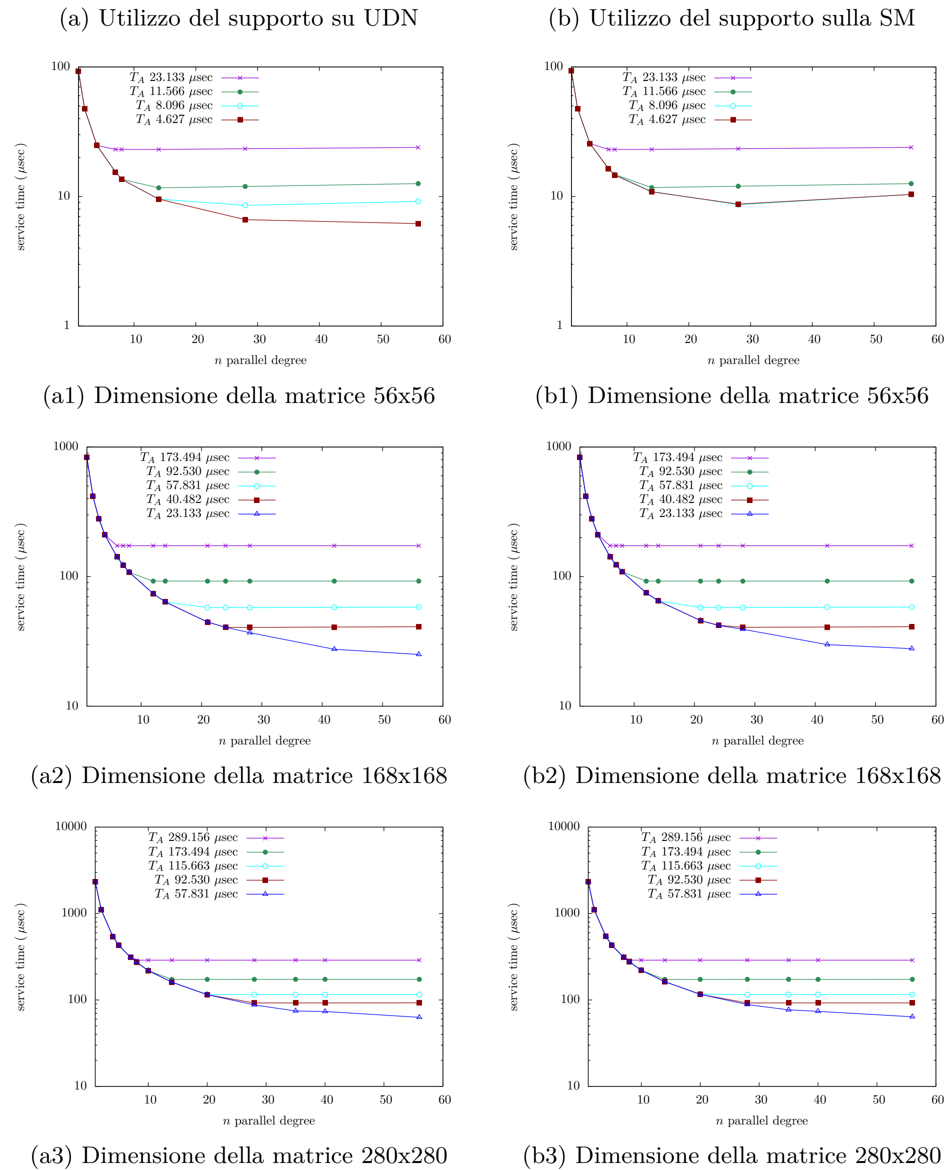
M	T_A	*Impl*	n_{opt}	n	$s^{(n)}$	${T_{\Sigma 1}}^{(n)}(\tau)$
56	$4 \cdot 10^3\ \tau = 4.626\ \mu$sec	UDN	21.716	56	15.529	5340.578
		SM	23.568	28	10.982	7552.140
168	$20 \cdot 10^3\ \tau = 23.1325\ \mu$sec	UDN	37.102	56	33.894	21694.473
		SM	37.672	56	30.582	24044.160
280	$50 \cdot 10^3\ \tau = 57.831\ \mu$sec	UDN	40.957	56	37.495	54419.290
		SM	41.205	56	36.937	55241.196

Tabella 4.4: Tempi di servizio effettivo e scalabilità della Map *migliori* quando la Map è collo di bottiglia.

M	T_A	*Impl*	n_{opt}	n	$s^{(n)}$	${T_{\Sigma 1}}^{(n)}(\tau)$
56	$10 \cdot 10^3\ \tau = 11.566\ \mu$sec	UDN	8.446	8	7.074	11723.431
				14	8.220	10089.545
		SM	8.713	8	6.491	12776.486
				14	8.182	10136.235
168	$50 \cdot 10^3\ \tau = 57.831\ \mu$sec	UDN	14.718	14	13.336	55139.259
				21	14.739	49888.153
		SM	14.807	14	13.095	56151.886
				21	14.735	49902.936
280	$10 \cdot 10^4\ \tau = 115.662\ \mu$sec	UDN	20.441	20	20.433	99859.243
				28	20.448	99787.616
		SM	20.503	20	20.163	101196.684
				28	20.450	99778.438

Tabella 4.5: Tempi di servizio effettivo e scalabilità della Map quando la Map non è collo di bottiglia. Vengono mostrati i valori relativi al grado di parallelismo vicino al valore ottimo calcolato.

Figura 4.6: Grafici del tempo di servizio del sottosistema Map al variare del tempo di interarrivo



(a) Utilizzo del supporto su UDN

(b) Utilizzo del supporto sulla SM

(a1) Dimensione della matrice 56x56

(b1) Dimensione della matrice 56x56

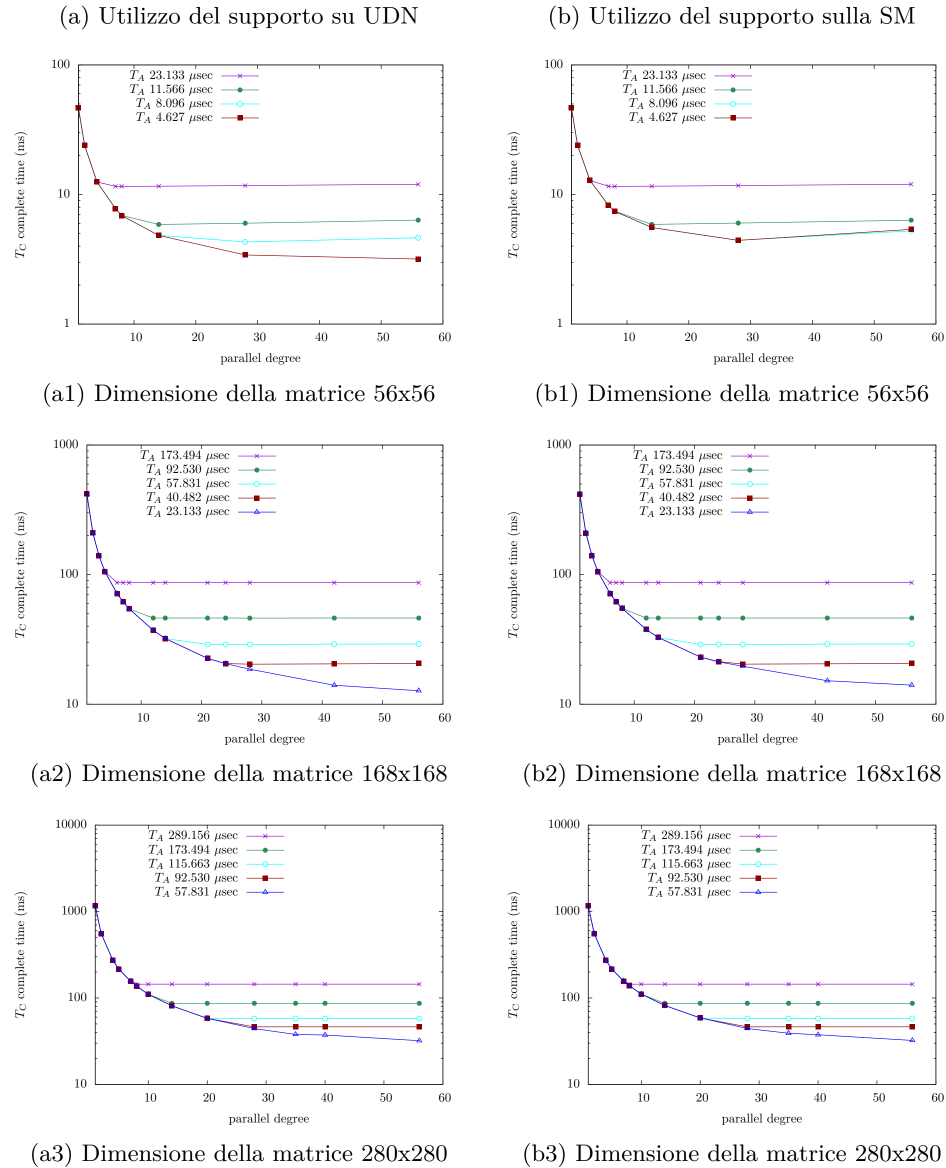
(a2) Dimensione della matrice 168x168

(b2) Dimensione della matrice 168x168

(a3) Dimensione della matrice 280x280

(b3) Dimensione della matrice 280x280

Figura 4.7: Grafici del tempo di completamento dello stream al variare del tempo di interarrivo



(a1) Dimensione della matrice 56x56

(b1) Dimensione della matrice 56x56

(a2) Dimensione della matrice 168x168

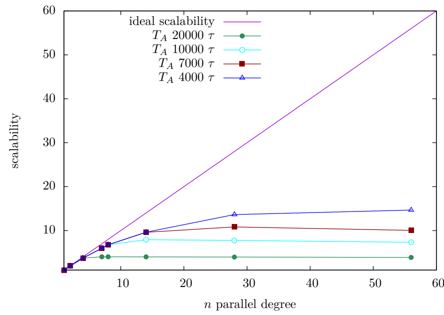
(b2) Dimensione della matrice 168x168

(a3) Dimensione della matrice 280x280

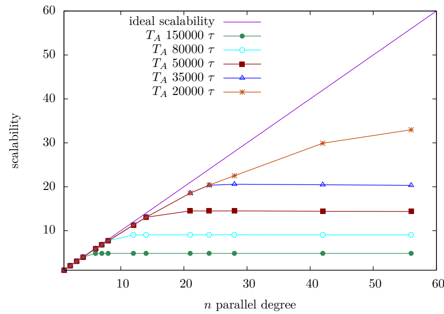
(b3) Dimensione della matrice 280x280

Figura 4.8: Grafici di scalabilità del tempo di completamento dello stream al variare del tempo di interarrivo

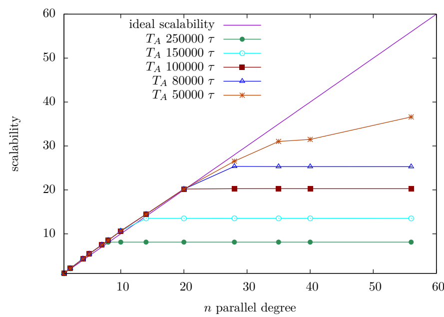
(a) Utilizzo del supporto su UDN



(a1) Dimensione della matrice 56x56

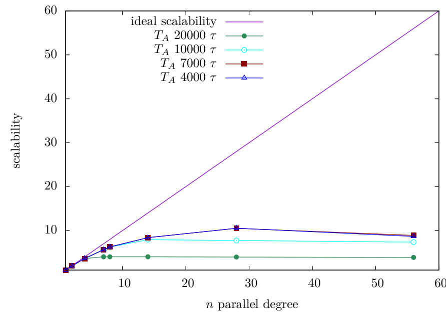


(a2) Dimensione della matrice 168x168

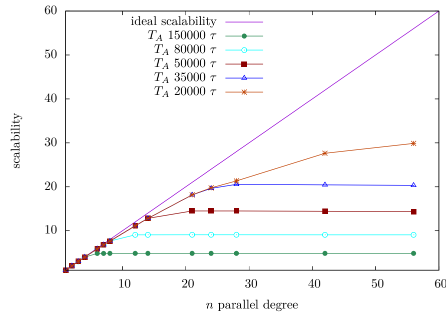


(a3) Dimensione della matrice 280x280

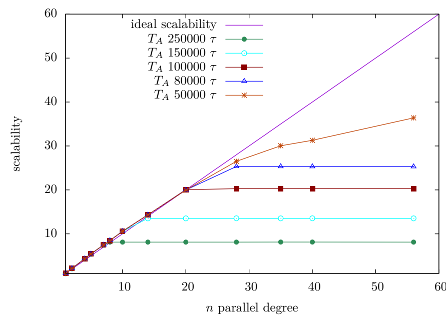
(b) Utilizzo del supporto sulla SM



(b1) Dimensione della matrice 56x56



(b2) Dimensione della matrice 168x168



(b3) Dimensione della matrice 280x280

Analisi delle possibili degradazioni

Si è indagato su quali siano le cause della cattiva scalabilità che caratterizzano le esecuzioni, in particolar modo quella con dimensione minore dei dati. Esiste una parte, o più parti, del sistema che non si comporta come descritto dalla formula applicata per calcolare la scalabilità. Il sistema può essere visto come costituito da tre fasi collegate in pipeline: multicast, calcolo, gather. Il programma è stato rieseguito misurando il tempo di servizio di ciascuna fase, al fine di verificare il comportamento effettivo del sistema. Da tali misurazioni non risulta problematica la fase di gathering, mentre sono le altre due fasi che aumentano il proprio tempo di servizio con il crescere di n.

fase di calcolo le misure del tempo di calcolo sono proposte in maniera leggermente elaborata nelle figura 4.9a: viene esposto il rapporto tra i tempi di calcolo di un singolo prodotto scalare nel programma sequenziale e nel calcolo di un worker del sistema parallelo di grado n. Tale rapporto dovrebbe essere costante, indipendentemente dal grado di parallelismo; si osserva tuttavia un aumento di tale rapporto al crescere di n. Questa degradazione è tanto maggiore quanto più piccole sono le dimensioni dei dati. È ragionevole ipotizzare la causa di questo comportamento come l'aumento del tempo di risposta alla memoria indotto dall'incremento del numero di richieste alla memoria prodotto da valori crescenti di n. Come spiegato precedentemente, l'aumento del numero di clienti in una computazione *client-server domanda-e-risposta* produce un aumento del tempo di risposta medio del servente. Inoltre il fatto che la degradazione sia maggiore con dimensione più piccola della matrice porta a convincersi di tale ragionamento, in quanto la diminuzione della dimensione dei dati porta ad un calo della grana di computazione dei worker, e quindi ad un aumento della frequenza di accessi alla memoria (controllore o L2 cache home) dovuti al verificarsi di un fault nella cache locale. Per convincersi di tale ragionamento, e grazie ad una particolare caratteristica architetturale, è stato effettuato un ulteriore esperimento, mostrato in figura 4.9b: la stessa misurazione è presa per dati in virgola mobile. I core di TILE*Pro*64 non dispongono di unità di elaborazione firmware per i calcoli in virgola mobile, pertanto tali conti vengono interpretati a livello assembler. Ne segue che un calcolo aritmetico in virgola mobile viene interpretato come molte istruzioni assembler. Rispetto allo stesso calcolo con tipi interi, si ha una frequenza di accesso alla memoria inferiore, con conseguente riduzione del tempo di risposta. Il comportamento mostrato dall'esperimento con dati float ha infatti il tempo di calcolo del singolo prodotto scalare costante (o quasi) per tutte le dimensioni della matrice.

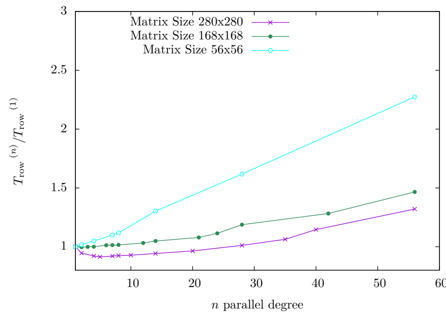
fase di multicast dato che la multicast non è realizzata da un sottosistema distinto da quello di calcolo, ma è mappata nei worker, si sono considerati i tempi di servizio della multicast relativi alle situazioni in cui il sistema non è collo di bottiglia. Le misurazioni sono mostrate in figura 4.10. I valori effettivi delle comunicazioni, in una applicazione *reale*, risultano maggiori rispetto a quelli misurati con l'applicazione ping-pong. Si osserva inoltre un aumento del tempo di servizio con l'aumentare di n, soprattutto con dimensioni piccole dei dati. Anche in questo caso, tale tempo di servizio sarebbe dovuto essere costante. Per il supporto che usa la memoria

condivisa possono essere portate le stesse argomentazioni della fase di calcolo.

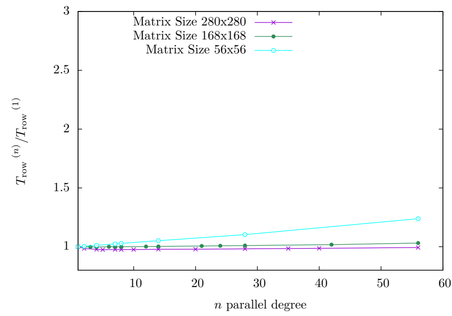
Figura 4.9: Fattore di variazione del tempo di calcolo di un singolo prodotto scalare: rapporto tra il tempo medio di calcolo di un singolo prodotto scalare nel calcolo di un worker della Map rispetto allo stesso tempo nel programma sequenziale. La Map è collo di bottiglia: $T_A = 4.626\mu$sec.

(a) Utilizzo di dati di tipo Intero

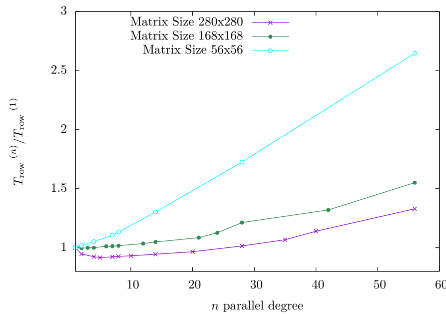
(b) Utilizzo di dati di tipo Floating Point



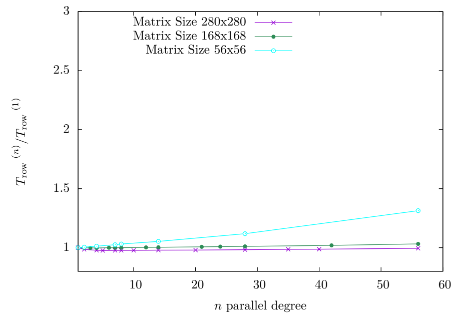
(a1) Utilizzo del supporto su UDN



(b1) Utilizzo del supporto su UDN

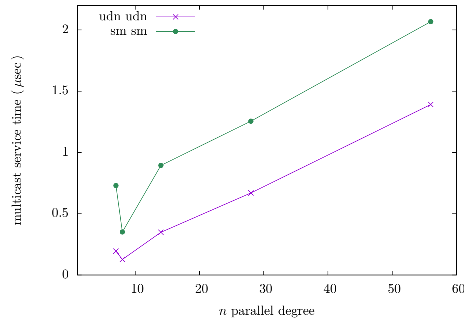


(a2) Utilizzo del supporto su SM

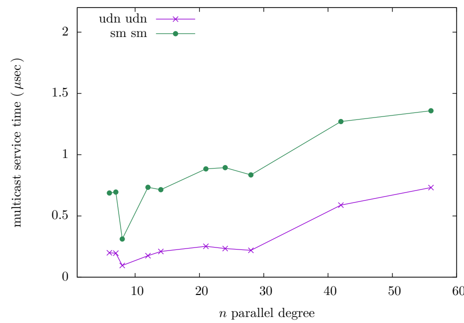


(b2) Utilizzo del supporto su SM

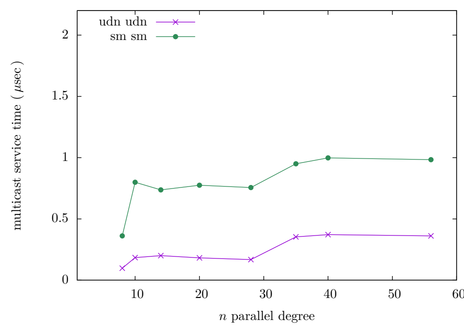
Figura 4.10: Confronto di uso dei due supporti nel tempo di servizio della multicast misurata sul primo worker quando la Map non è collo di bottiglia



(a) Dimensione della matrice 56x56, $T_A = 20 \cdot 10^3\, \tau = 23.133\, \mu$sec



(b) Dimensione della matrice 168x168, $T_A = 150 \cdot 10^3\, \tau = 173.494\, \mu$sec



(c) Dimensione della matrice 280x280, $T_A = 250 \cdot 10^3\, \tau = 289.157\, \mu$sec

Confronto tra le due implementazioni

Concludiamo infine con il vero scopo del benchmark, ovvero il confronto tra le due implementazioni. La figura 4.11 offre il confronto delle due implementazioni nella scalabilità e nel tempo di servizio, con il sistema che è collo di bottiglia. Come atteso, le differenze tra le due versioni si osservano sulla grana più fine del calcolo nei worker, quindi sulla dimensione più piccola della matrice. All'aumentare della dimensione della matrice cresce la grana di calcolo e le differenze si assottigliano.

Al fine di confrontare puntualmente le due realizzazioni del benchmark è utile la tabella 4.4 proposta precedentemente. Considerando la dimensione delle matrici 56x56: l'implementazione su UDN ha un fattore di scalabilità di 15 con 56 processi, contro una scalabilità di 10 con 28 processi, della versione SM; il tempo di servizio effettivo della UDN è 5340.6 $\tau = 6.177$ μsec con 56 processi, contro i 7552.140 $\tau = 8.735$ μsec sempre con 28 processi, della SM.

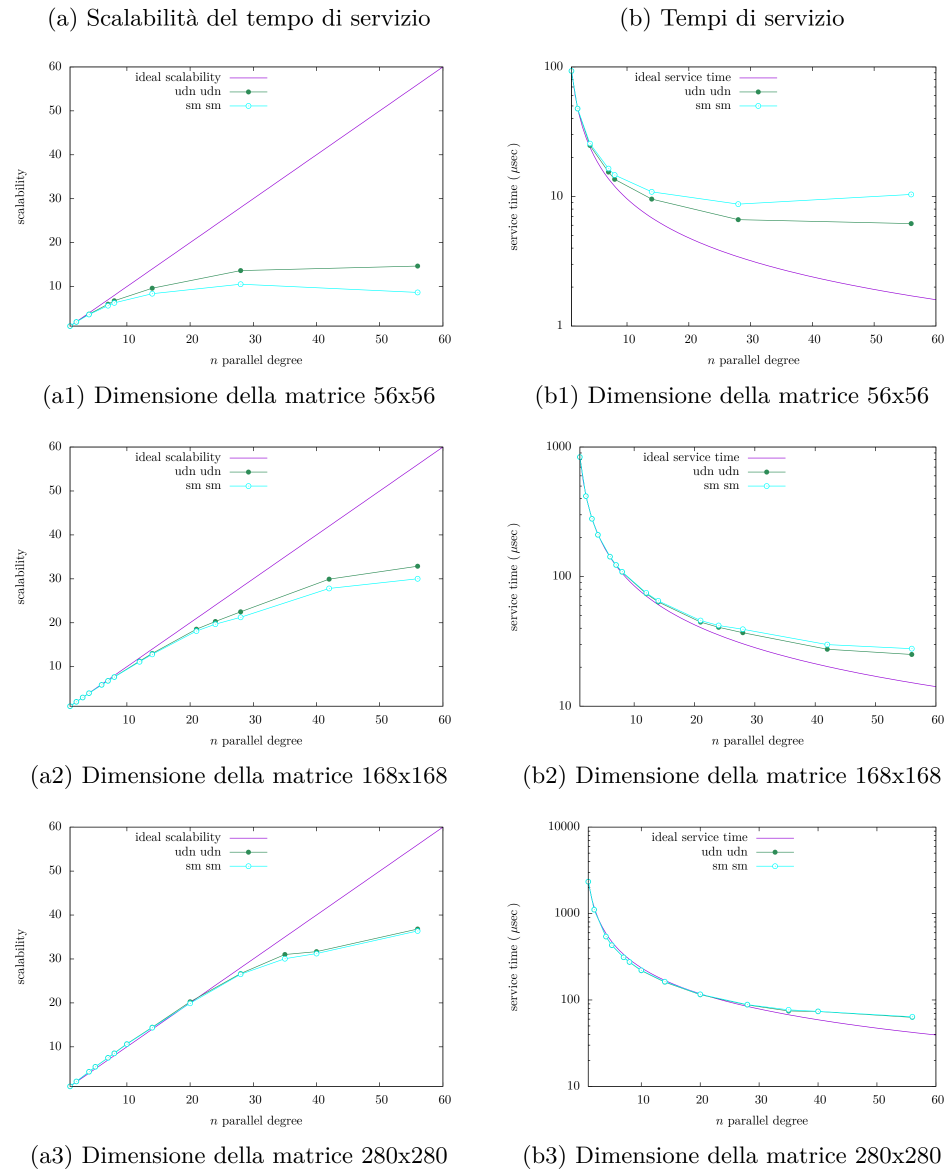
Il benchmark e le misure fatte offrono spunti di riflessione sulle due versioni del supporto alle comunicazioni realizzati:

tempo di calcolo La misura sul tempo di calcolo di un singolo prodotto scalare ci permette di confrontare l'aumento che sia ha dei tale tempo con l'incremento dei n nelle due implementazioni: graficamente ciò è osservabile confrontando le figure 4.9a1, 4.9a2. L'implementazione su SM ha un tempo calcolo superiore a quello della versione su UDN, ciò è visibile soprattutto con la dimensione più piccola dei dati. Questo è significativo, in quanto l'uso di un supporto diverso e indipendente dalla memoria condivisa porta effettivamente ad una minore congestione del sottosistema della memoria e quindi a una minore degradazione del tempo di calcolo. Si osserva che il guadagno ottenuto non è elevato, ovvero, esiste anche in UDN una degradazione preponderante, tuttavia la differenza rispetto all'implementazione SM è visibile con tale grana del calcolo. Occorre considerare che la versione UDN del supporto fa uso della rete di interconnessione solo per la trasmissione dei puntatori, mentre la trasmissione vera e propria dei dati è affidata al sottosistema di cache e di memoria condivisa. Ciò fa ben sperare che una implementazione che sfrutti la UDN anche per la comunicazione tipi di dato arbitrari abbia influenza ancor più positiva nelle prestazioni globali rispetto alla comunicazione di riferimenti. In questo caso caso infatti anche la trasmissione delle strutture dati "passerebbe" per la UDN, riducendo così il grado di utilizzo della gerarchia di memoria condivisa.

Ciò fa sperare che una implementazione delle comunicazioni che sfrutti la UDN anche per la trasmissione dei dati abbia impatto molto maggiore nelle prestazioni complessive.

tempo della multicast La lettura dei grafici del tempo di multicast (figura 4.10) quando il sistema non è collo di bottiglia, ci conferma che la latenza di comunicazione del supporto UDN si mantiene ben inferiore a quella della versione SM anche in una applicazione reale, e non solo nell'applicazione ping-pong.

Figura 4.11: Confronto della scalabilità e del tempo di servizio tra le due implementazioni con tempo di interarrivo $T_A = 4 \cdot 10^3\, \tau = 4.6265\, \mu$sec



(a) Scalabilità del tempo di servizio

(b) Tempi di servizio

(a1) Dimensione della matrice 56x56

(b1) Dimensione della matrice 56x56

(a2) Dimensione della matrice 168x168

(b2) Dimensione della matrice 168x168

(a3) Dimensione della matrice 280x280

(b3) Dimensione della matrice 280x280

5 Conclusioni

Si è svolto un primo esperimento finalizzato all'utilizzo di una rete di interconnessione tra processing element di una macchina multi-core per la realizzazione di un supporto alle comunicazioni tra processi. L'implementazione vuole far fronte a problemi di grana molto fine, e per questo motivo il supporto permette lo scambio di messaggi di tipo riferimento. Il problema è stato studiato su una macchina ben precisa, il processore Tilera TILE*Pro*64, il quale mette a disposizione dell'utente una rete mesh bidimensionale UDN, che ha buone caratteristiche di scalabilità e che implementa a firmware quattro diversi flussi. È stata proposta una implementazione che sfrutta al meglio ciò che la macchina offre, seppure con una limitazione: un processo non può usare più di quattro canali. Da un lato non si ritiene rilevante una implementazione più generica, in quanto molte forme di parallelismo sono implementabili con questo vincolo. Dall'altra parte, se vi è necessità di utilizzare più canali, è sempre possibile impiegare canali implementati sulla memoria condivisa. Del supporto è infatti fornita anche una versione che utilizza l'approccio classico, servendosi della memoria condivisa. La versione su memoria condivisa fa uso di aspetti avanzati, quali la configurazione della coerenza delle cache, resi disponibili dalla specifica macchina utilizzata.

Al fine di confrontare le prestazioni dei due tipi di implementazione, si sono svolti due esperimenti: il primo è la misura della latenza di comunicazione, in una applicazione specifica per tale scopo, senza l'esecuzione di alcun calcolo; il secondo è l'utilizzo del supporto in una applicazione reale, significativa per i nostri scopi. Le misure del primo esperimento confermano quanto era stato intuito: l'implementazione che usa la rete di interconnessione ha meno della metà della latenza di comunicazione della versione che usa la memoria condivisa. Le misure fatte sulla seconda applicazione enfatizzano questo primo risultato: in una applicazione reale si continua a osservare una latenza di comunicazione del supporto UDN inferiore alla metà di quella misurata con l'uso della memoria condivisa. Il secondo esperimento inoltre ha mostrato una diminuzione della latenza di accesso alla memoria con l'uso del supporto su UDN, rispetto all'impiego delle comunicazioni su memoria condivisa. Ciò è conseguenza della diminuzione del numero di accessi alla gerarchia di memoria. Si conclude che tale caratteristica del supporto su UDN ha benefici sulle prestazioni globali dell'applicazione.

L'utilizzo di un supporto alle comunicazioni basato su UDN, piuttosto che su memoria condivisa, è risultato significativo per il tipo di comunicazione studiato. Computazioni su stream con grana di calcolo fine richiedono un supporto alle comunicazioni efficiente, soprattutto nel caso in cui l'architettura, come quella del TILE*Pro*64, non disponga dei supporti architetturali per sovrapporre la comunicazione al calcolo. L'utilizzo della UDN per le comunicazioni non solo offre latenze di comunicazione inferiori a quelle del supporto con memoria condivisa, ma è caratterizzato da altre caratteristiche importati per le prestazioni di una applicazione di grana fine: i conflitti alla memoria sono ridotti, ed esiste una forma ridotta di sovrapposizione della comunicazione al calcolo.

5.1 Sviluppi futuri

Lo sviluppo più interessante del lavoro fatto è l'implementazione, per mezzo della rete di interconnessione UDN, di canali di tipo arbitrario. Questo permette la realizzazione della comunicazione di messaggi di dimensione arbitraria, per mezzo di una rete di interconnessione e senza l'utilizzo di strutture dati allocate in memoria. Un supporto alle comunicazioni di questo tipo permette un disaccoppiamento completo tra gli strumenti architetturali usati per realizzare le comunicazioni e quelli impiegati per realizzare l'accesso ai dati condivisi in memoria. Per questo motivo, e per le caratteristiche di scalabilità e banda della UDN, si pensa che una realizzazione di questo tipo conduca a risultati ancora più apprezzabili di quelli ottenuti in questo primo esperimento. Per questo tipo di supporto sussistono alcuni problemi progettuali tipici nelle comunicazioni a scambio di messaggi di processi allocati in un grafo di nodi.

Una possibile espansione consiste nel fornire le stesse forme di comunicazione con grado di parallelismo più alto. Ciò può essere comodo in molti casi per compensare una frequenza media di interarrivo con alta variabilità. L'implementazione di questo aspetto su UDN è in gran parte fornito in modo primitivo dalla rete di interconnessione, per cui la realizzazione è banale. Nella versione che usa la memoria condivisa è richiesta invece una riprogettazione del supporto, valutando se adottare strutture dati lock-free, più complesse rispetto alla realizzazione con buffer di messaggi condiviso.

Infine, è pensabile una realizzazione del supporto su UDN che non limiti il numero di canali per processo. A tal fine è richiesto l'uso di una memoria tampone nella quale copiare messaggi o segnali ricevuti dalla UDN ma che appartengono a un canale differente da quello su cui è stato invocata la funzionalità del supporto.

Riferimenti bibliografici

[1] Daniele Buono, Marco Danelutto, Silvia Lametti, and Massimo Torquati. Parallel patterns for general purpose many-core. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pages 131–139. IEEE, 2013.

[2] Murray Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel computing*, 30(3):389–406, 2004.

[3] Tilera Corporation. Tilepro processor family, `http://www.tilera.com/products/processors/TILEPro_Family`, 2012.

[4] Horacio González-Vélez and Mario Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Software: Practice and Experience*, 40(12):1135–1160, 2010.

[5] Timothy G Mattson, Beverly A Sanders, and Berna L Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, 2004.

[6] G. Mencagli T. De Matteis, F. Luporini and M. Vanneschi. Evaluation of architectural supports for fine-grained synchronization mechanisms. In *11th IASTED International Conference on Parallel Distributed Computing and Networks, Innsbruck, Austria*, 2013.

[7] Tilera. *UG101 - Tile Processor User Architecture Manual*.

[8] Tilera. *UG120 - Tile Processor Architecture Overview for the Tilepro Series*.

[9] Tilera. *UG205 - Programming the Tile Processor*.

[10] M. Vanneschi. *Architettura degli elaboratori*. Didattica e ricerca: Manuali. Plus, 2009.

[11] M. Vanneschi. *Parallel Architectures*. Course notes of High Performance Computing in Computer Science and Networking, University of Pisa. 2012.

[12] M. Vanneschi. *Structuring and Design Methodology for Parallel Computations*. Course notes of High Performance Computing in Computer Science and Networking, University of Pisa. 2012.

[13] D. Wentzlaff, P. Griffin, H. Hoffmann, Liewei Bao, B. Edwards, C. Ramey, M. Mattina, Chyi-Chang Miao, J.F. Brown, and A. Agarwal. On-chip interconnection architecture of the tile processor. *Micro, IEEE*, 27(5):15–31, 2007.