Indice

1 Introduzione

Nella programmazione parallela è consolidato l'uso di paradigmi di parallelismo 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, soprattuto 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 evuluzione e ha prodotto numerosi risultati [2], 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. È preso in esame il Network Processor Tilera TILE*Pro*64, il quale integra, in un singolo chip, 64 unità di processamento interconnesse da sei 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 processori.

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 [1].

Il principale vantaggio nell'utilizzo di strutture di interconnessione tra processori 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: la gestione delle comunicazioni distinta dall'accesso ai dati in due diversi sottosistemi architetturali riduce i conflitti al sottosistema di memoria principale e di cache e alle strutture di interconnessione che le gestiscono con una conseguente diminuzione del tempo di accesso; 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 dei processori di comunicazione, infatti, 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 diponibili dal supporto sono quelle di uso più comune nei paradigmi di programmazione parallela. In particolare, le scelte fatte nell'implementazione di questo supporto sono state guidate 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. Questa concezzione ha derivato alcune scelte progettuali, ad esempio: il grado di parallelismo 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.

Le forme di comunicazione considerate sono quelle di canali simmetrico unidirezionale e asimmetrico unidirezionale in ingresso. Entrambi i canali 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 dimesione arbitraria facilita l'implementazione del supporto con UDN. Questo tipo di implementazione dei canali è frequente in processori di networking dove i messaggi scambiati tra i core dello stesso processore 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.

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 UDN;

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 captiolo si descrivono gli esperimenti che sono stati realizzati al fine di valutare le due diverse tipologie di implementazione del supporto alle comunicazioni. Sono presentate le misure della latenza delle comunicazioni ottenute per mezzo di una applicazione "ping-pong".

2 Caratteristiche dell'architettura complessiva

Il Chip multicore TILE*Pro*64 è costituite da 64 unità di processamento (tile) connesse attraverso una struttura di interconnessione on-chip Tilera iMesh. Ogni processore è costituito da una unità di processamento strutturata come un pipline 3-way VLIW, dall'unità di switching che collega il processore alla rete di interconnessione iMesh e dal sottosistema di cache costituito da due livelli: il primo livello costituito dalle parti istruzioni (L1i, 16KB) e dati (L1d, 8KB), contiene una tabella associativa che realizza il Translation Lookaside Buffer, 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.

2.1 Reti di interconnessione

La Tilera iMesh [6] è una struttura di interconnessione dei tile composta da 6 reti mesh bidimensionali identiche e indipendenti, ciascuna delle quali realizza il trasporto di un ben preciso tipo di traffico. Cinque di queste reti sono dinamiche, ovvero è possibile richiedere l'invio di messaggi a un qualsiasi altro tile. Esistono tre reti dinamiche che si occupano del trasporto di dati della memoria:

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

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

CDN Coherence Dinamic 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 efficente di più flussi di dati realizzata attraverso una interfaccia a livello di registri FIFO e con hardware che supporta il direzionamento dei flussi in distinte FIFO.

IDN I/O Dynamic Network, principalmente è usata per il trasferimento tra tiles e dispositivi 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, accedibile da applicazioni eseguite al livello utente, permette la comunicazione tra tile 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 [].

L'unità di switching di ogni tile realizza per ogni rete un crossbar completo, tutti gli ingressi sono bufferizzati (il grado di asincronia è tre). L'ampiezza dei collegamenti, detta *flit*, è la parola della macchina, 32-bit. Le reti di TILE*Pro*64 operano alla stessa velocità dei processor cores, ne segue che la latenza per leggere un flit da un buffer di ingresso, attraversare il crossbar e memorizzare lo stesso flit nel buffer di un tile vicino è pari ad un singolo ciclo di clock. L'unità di routing è il pacchetto, il routing è di tipo statico e viene implementato attreversando prima la direzione X poi quella Y. Un pacchetto delle reti di memoria è costituito da un flit di intestazione, inserito dal tile mittente, usato dalle unità

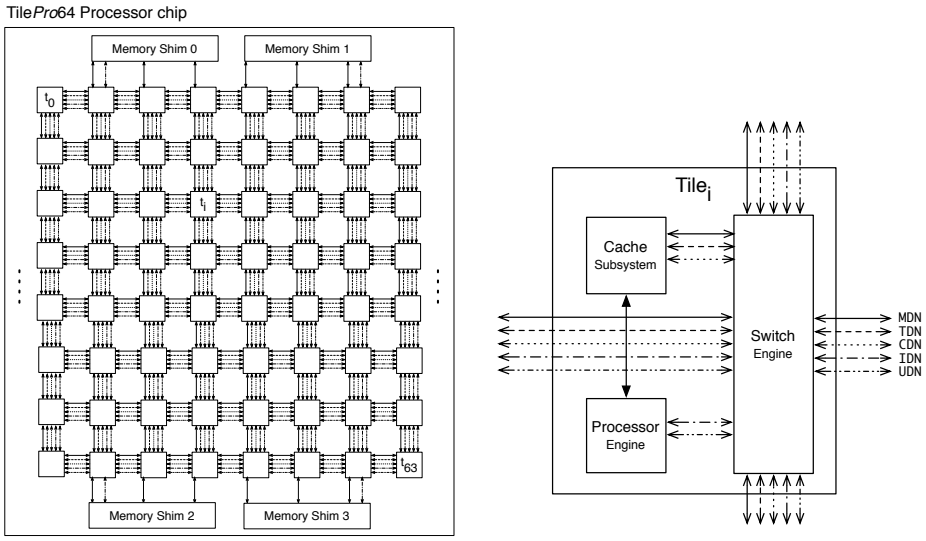


Figura 1: Rappresentazione parziale del processore TILE*Pro*64 con il dettaglio dell'architettura di un signolo tile

di switching intermedie per il routing e trasparente al tile destinatario e uno o più flit per il valore di cui è richiesta la trasmissione nella rete. La massima dimensione del pacchetto è 129 parole, l'intestazione contiene l'informazione del tile destinatario, quello mittente e la lunghezza del pacchetto. Le reti di messaggistica (UDN e IDN) contengono un secondo flit di intestazione, inserito anchesso dal tile mittente, necessario all'unità di switching del tile destinatario per direzionare il pacchetto nella FIFO corrispondente al flusso di cui fa parte il pacchetto. Tutte le reti hanno controllo di flusso di tipo *wormhole*, ovvero i flits del pacchetto sono trasmessi in modo pipeline nella rete diminuendo la latenza di trasmissione dei pacchetti rispetto al metodo store-and-forward. È garantita l'atomicità di trasmissione di un pacchetto: tramite il flit di intestazione, l'unità di switching determina la porta di uscita in cui deve essere trasmesso il pacchetto, una volta concesso l'uso di tale porta (potrebbe essere in esecuzione la trasmissione di un altro pacchetto su tale porta) blocca tale porta di uscita fin tanto che l'ultimo flit del pacchetto non ha attraversato con successo lo switch.

2.1.1 User Dynamic Network

La UDN è collegata direttamente tra unità di processamento e unità di switching di un tile. Sono disponibili quattro flussi UDN, nel modulo di processamento esistono altrettante code verso le quali sono demultiplexati 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 agli stadi di esecuzione ni pipeline dell'unità. L'accesso ai flussi UDN avviene tramite letture o scritture a 4 registri generali riservati alla UDN, si dice che la UDN è register-mapped. Il programmatore ha possibilità

```
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 1: Istruzioni assembler per accedere alla UDN

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 1.

L'accesso alla UDN è completamente interlocked, nel senso che, nelle situazioni in cui è verificata l'attesa nell'invio o nella ricezione l'esecuzione interna del modulo di processamento viene bloccata fin tanto che non si verifica la condizione di sveglia. Tale attesa è caratterizzata da un 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. Ne segue che un uso improprio di tale servizio può causare il deadlock dell'applicazione. Per scongiurare tale eventualità è necessario adottare protocolli di comunicazione adeguati e/o una dimensione dei pacchetti che non saturi le code di demultiplexing (circa 118 parole) [4].

In conclusione la rete UDN offre molti vantaggi (bassissima latenza nella comunicazione tile-to-tile, zero latenza nella sveglia, bassi consumi) e presenta alcune limitazioni (l'hardware supporta 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, per ottimizzare l'implementazione di operazioni critiche per le prestazioni, mentre è raccomandato l'uso della memoria condivisa per il più delle comunicazioni. Nel caso di computazioni di grana fine, i meccanismi chiave la cui ottimizzazione fornisce miglioramenti notevoli alle prestazioni sono quelli di sincronizzazione e comunicazione tra processi.

2.2 Architettura della memoria principale

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

- l'ordine delle istruzioni originale nel programma può essere modificato, così che accessi ad una pagina condivisa in memoria, effettuati da un certo processore, possono essere riordinati e diventare visibili con un altro ordine agli altri core,
- le operazioni di store eseguite da un processore appaiono visibili simul-

taneamente a tutti gli altri processori, ma possono diventare visibili al processore che ha emesso la scrittura prima che avvenga la visibilità globale.

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

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, l'allocazione dei blocchi avviene solo con read miss, la seconda ha dimensione 8KB con blocchi di 16B, l'allocazione avviene solo con load miss, le scritture sono write through. La cache L2 ha dimensione 64KB, blocchi di 64B e l'allocazione dei blocchi avviene sia con read miss che write miss, la politica di scritture è write back.

La coerenza della memoria cache è garantita automaticamente dall'hardware. È possibile configurare la macchina in diverse modalità:

- memoria coerente nelle caches,
- memoria non coerente nelle caches, 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 nella tile che ha eseguito la primitiva stessa),
- coerenza garantita dal non uso della gerarchia cache.

L'implementazione della coerenza automatica si basa su invalidazione, ed è implementata mediante la tabella direcotory distribuita nei processor cores della macchina. Per un certo blocco di cache b viene usato il termine Home Tile o Home Node con il consueto significato nei protocolli di coerenza cache: il processor core detiene e gestisce le informazioni di coerenza della cache relative al blocco b, e invia messaggi di invalidazione quando necessario. Un nodo gestisce la coerenza, e contiene le entrate della directory, dei soli blocchi di cui è home. Nell'architettura Tilera un Nodo 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 nodo verrebbe chiamato Owner di b oltre che Home). Questo consente di poter vedere un terzo livello di cache distribuito sulle L2: se un nodo esegue una lettura da memoria che risulta miss sia nella L1d che nella L2 locali, tale richiesta viene inoltrata al tile che è home del blocco corrispondente, pittosto che essere inviata direttamente alla memoria. È onere della L2 di tale tile 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 tile richiedente. Ne segue una caratterizzazione dei due livelli di cache: L1 è una cache privata, contenente solo le informazioni richieste dal unità di processamento locale, la L2 è una cache condivisa in quanto può contenere sia blocchi richiesti dal processore locale, che blocchi di cui è home e per i quali ha ricevuto una richiesta da un altro tile.

Lo schema con cui i blocchi di memoria vengono associati ai tile 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 tile. 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 homed nel tile 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 tile. Ciò è utile per dati privati in quanto non trarrebbero vantaggio dall'uso della cache di una altro tile 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 tile della macchina con granularità di un blocco L2 nei tile. Tale dispersione delle pagine di memoria nella cache dei tile consente una distribuzione uniforme del traffico nelle reti del chip e un effettivo bilanciamento del traffico di richieste alla memoria tra l'insieme dei tiles.

Remote Homing per una generica pagina di memoria viene impostato il corrispondente Home Tile. Tale tile è anche chiamato L3 per la pagina, infatti le richieste di accesso ad un qualsiasi blocco interno alla pagina, da parte di altre tile, vengono inoltrate alla tile home. Questa 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 accedute con uno schema produttori-consumatore, infatti configurare la pagina dell'oggetto in questione come homed nella tile del consumatore aumenta la località degli acceessi di tale processo, il quale si trova la struttura modoficata dai produttori direttamente nella propria L2, in quanto le scritture effettuate dai produttori sono write through nella cache del tile 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".
Di seguito 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 processori.

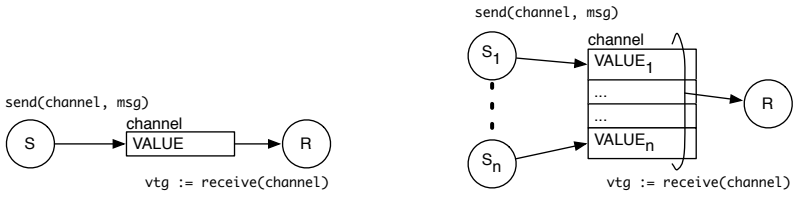
Si precisa che quando si parla di *processore* si intende l'elemento, o nodo, interno al chip multicore costituito dalle unità di processamento, di cache e di switching. Con il termine processore ci riferiamo quindi a ciò che viene comunemente chiamato *core* (o *tile* nella nomenclatura Tilera). Nel seguito si usa il termine *processo* per riferirsi all'unità di parallelismo dell'applicazione, la quale può essere un processo linux come un thread posix a seconda dell'implementazione. Si precisa inoltre che si considera solo un *mapping di tipo esclusivo* dell'applicazione nell'architettura, ovvero esiste una corrispondenza biunivoca tra ogni processo dell'applicazione e il processore della macchina che lo esegue.

3.1 Specifica dei canali di comunicazione

Un canale di comunicazione *simmetrico* è un oggetto che mette in relazione un processo mittente con un processo destinatario, consentendo l'invio di messaggi dal processo mittente al processo destinatario. Un canale di comunicazione *asimmetrico in ingresso* è un oggetto che mette in relazione più processi mittenti con un singolo processo destinatario, consentendo a ciascun mittente l'invio di messaggi al processo destinatario, il quale 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 uniforme per le implementazioni diverse:

```
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)
```



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

Figura 2: Rappresentazione astratta di canali con grado di asincronia unitario

Ad ogni primitiva viene quindi passato il descrittore di canale come riferimento ad una struttura dati allocata in memoria condivisa. 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, quindi non viene utilizzata la tecnica zero-copy, 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 *rank* 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. Si è quindi adottato il protocollo per la comunicazione di unità firmware per la comunicazione tra processi. Tale protocollo di comunicazione è caratterizzato dall'uso di due messaggi di sincronizzazione, detti Ready (Rdy) e Acknowledgement (Ack) e per questo motivo è chiamato nel seguito protocollo 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.

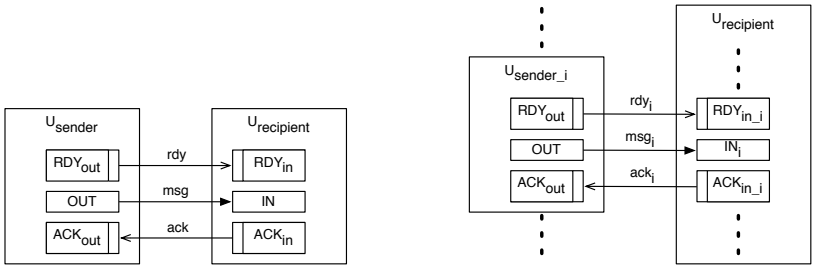
```
send ::
  wait until acknowledgement signal is received
  send the message to recipient and
    reset acknowledgement flag and
    signal ready to the recipient

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 2: Descrizione astrata 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 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 messagio 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.

3.2 Descrizione dei canali al livello firmware

Prima di descrivere i protocolli di comunicazione e la relativa implementazione adottata con i due supporti architetturali si descrive brevemente il protocollo tipico per realizzare la comunicazione di unità di elaborazione firmware in modo indipendente dal tempo [5]. Le idee alla base di questo protocollo sono utilizzate per realizzare dei meccanismi efficenti e ottimizzati sull'architettura di TILE*Pro*64.



(a) Comunicazione simmetrica (b) Comunicazione asimmetrica

Figura 3: Le componenti di un canale di comunicazione firmware

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 $U_{recipient}$ collegate dai collegamenti *rdy(1)*, *msg(L)*, *ack(1)* come in figura 3a, il protocollo di comunicazione è perciò definito come segue:

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

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

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 3b. 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 esgue il protocollo del caso simmetrico, leggendo il valore, resettando il Rdy e inviando l'ack per il canale scelto.

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: 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 pattern 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 add 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 un processore, 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), 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 eliminta 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 al processore, in quanto servite dalla cache locale. Quando il processo pattern 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 nodo 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, consentendo invece l'attuazione di un modello di programmazione di tipo *remote storing*. 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 nodi home e al partizionamento del descrittore del canale nelle cache dei processori.

La strategia predefinita di allocazione delle pagine di memoria nelle cache è *Hash-for-Home* [4]. 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 nodo N_k. Durante l'esecuzione del programma parallelo il processo mittente, eseguito nel nodo N_i, e il processo destinatario, eseguito nel nodo N_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 4. La modifica di un campo del descrittore del canale effettuata da

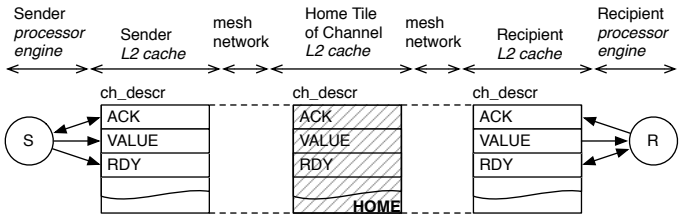


Figura 4: Rappresentazione di una comunicazione simmetrica tra un processo mittente S e uno destinatario R con uso della memoria condivisa e gestione predefinita della coerenza di cache (hash-for-home strategy)

uno dei due processi comunicanti provoca l'invalidazione del blocco di L2 nel nodo che esegue il processo pattern, l'intero blocco sarà trasferito dal nodo home N_k al nodo invalidato al prossimo acceso al descrittore di canale effettuato nel nodo 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 pattern 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 nodo che attende il segnale, evitando il trasferimento di un blocco di L2 per la trasmissione di una sola parola. Tale comportamento è simile al modello di Remote Store Programming nel quale un processo può scrivere nella memoria locale di un altro processo. Dato che per ciascun campo del descrittore di canale esiste un unico consumatore (vedi Tabella 1), è possibile applicare alle cache L2 il modello RSP utilizzando la strategia di allocazione Remote Homing del TILE*Pro*64. Il descrittore di canale viene partizionato in due parti: *1*) la parte di output contine tutti i campi acceduti in lettura del mittente (Ack), ed è allocata impostando come nodo home quello che esegue il processo mittente, *2*) la parte di input contiene tutti i campi acceduti in lettura dal mittente (Rdy e Value), ed è allocata impostando come nodo home quello che esegue il processo destinatario.

Con tale configurazione si incrementa la località delle informazioni: la coerenza dei dati e l'allocazione degli stessi avviene nella cache del nodo che consuma tali dati, permettendo al nodo 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 propria cache.

Si osserva inoltre che in tale computazione il nodo consumatore di una informazione (ad esempio il nodo mittente per la parte di output del canale) esegue

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

Tabella 1: Modalità di accesso ai campi del descrittore di canale

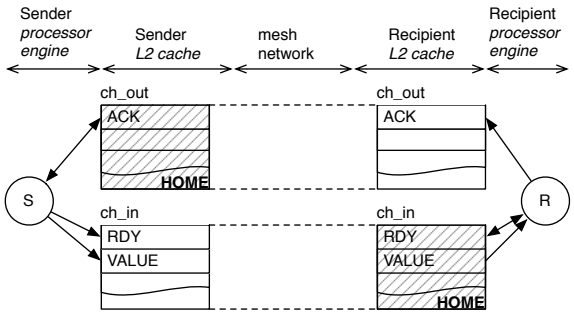


Figura 5: Rappresentazione di una comunicazione simmetrica tra i processi S e R con uso della memoria condivisa e gestione ottimizzata della coerenza di cache con strategia Remote Homing per il descrittore di canale

anche scritture sul blocco, ciò causa l'invio di messaggi di invalidazione del blocco alla cache L2 del nodo 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 nodo 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 nodi comunicanti con strategia Remote Homing oppure in un terzo nodo 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'analsi 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 nodo home quello ricevente mentre *Ack* ha per home il nodo 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 nodo destinatario, con strategia Remote Homing. I valori di *Ack* sono invece allocati ciascuno in un blocco che ha per home il nodo mittente corrispondente, questo consente la maggior località possibile.

Si fa presente che sebbene tale configurazione permetta la massima località delle informazioni ai processori, 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 nodo 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 nodi mittenti, e il riferimento alla base del vettore delle coppie $< Rdy, value >$ allocato con strategia Remote Homing nel nodo destinatario. Come descritto nella Sezione 3.1 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 soddifsatta.

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

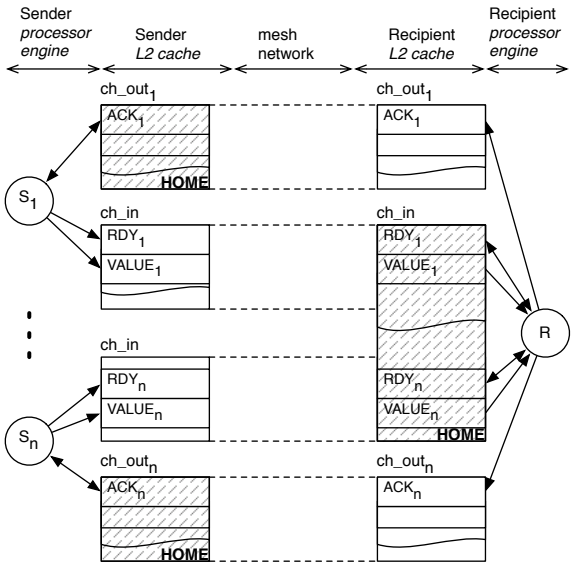


Figura 6: Rappresentazione di una comunicazione asimmetrica tra i processi $\{S_1, \ldots, S_n\}$ e R con uso della memoria condivisa e gestione ottimizzata della coerenza di cache con strategia Remote Homing per il descrittore di canale

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 User Dynamic Network 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 con protocollo alternativo

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 rifà a quello Rdy-Ack, in quanto sono mantenuti gli eventi di sincronizzazione

```
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 4: Descrizione astratta del protocollo di comunicazione Null-Ack su memoria condivisa

"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 ul 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 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 processore che esegue il processo destinatario produca le due scritture nell'ordine spcificato. Il campo `value` ha per home il nodo 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 processori, in quanto l'oggetto scritto (`value`) è acceduto in sola scrittura dal mittente. La seconda scrittura è una write-through alla L2 del mitttente, 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 tile 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 User Dynamic Network 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 sottositema 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: *a*) sono disponibili quattro canali UDN (fisici), ciascuno dei quali è associato univocamente ad una coda firmware nel processore destinatario, *b*) è fissata la massima dimensione dei pacchetti trasmessi nella rete, la quale non può superare la dimensione delle code firmware nei processori. 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 firware. 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, ovvero non si fa uso della memoria condivisa ne per la sincronizzazione ne per la trasmissione. 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 processori che esguono 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 sengali 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 7 dove il canale di comunicazione è implementato tramite l'utilizzo della seconda coda UDN nel processo mittente, per la ricezione dei segnali di Ack, e con la quarta coda UDN nel processo destinatario, per la ricezioe

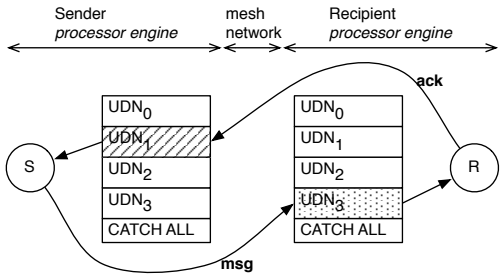


Figura 7: Rappresentazione di un possibile scenario di comunicazione simmetrica tra il processo mittente S e quello destinatario R, sfruttando la User Dynamic Network

```
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_rcp del processore ch_descr->cpu_rcp

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

Codice 5: Descrizione astratta del protocollo di comunicazione Rdy-Ack su User Dynamic Network per un canale di comunicazione simmetrico

dei messaggi; ne segue che in entrambi i processori, 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 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 processori, utilizzate dal canale per la comunicazione. La correttezza del protocollo si ottine 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/processori 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 messaggo di uno qualsiasi tra i processi mittenti. Un messaggio inviato da un mittente è costituito da una coppia di 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

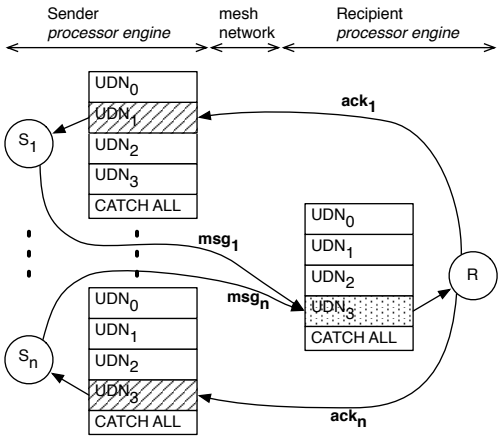


Figura 8: Rappresentazione di un possibile scenario di comunicazione asimmetrica in ingresso tra l'insieme di processi mittenti $\{S_1, \ldots, S_n\}$ e il processo destinatario R, sfruttando la User Dynamic Network

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 8 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 mittete, 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 paramentro nella primitiva di invio: l'identificatore nel canale del mittente. Il comportamento del protocollo è descritto nel Codice 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 processore, la stessa coda UDN per più canali o compiti diversi. Si osserva inoltre che l'unità di routing di UDN è il pacchetto [3], 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.

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. An-

```
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_rcp del processore
    ch_descr->cpu_rcp

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

Codice 6: Descrizione astratta del protocollo di comunicazione Rdy-Ack su User Dynamic Network per un canale di comunicazione asimmetrico in ingresso

che 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 soltato se i messaggi di risposta del servente riempono la coda di un cliente, ciò accade solo quando un cliente invia più richeste della capacità della coda UDN di memorizzare le risposte [4].

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 impelmentazione 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 (overheads).

4 Esperimenti

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 2 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.

Supporto architetturale	*Nome dei canali di comunicazione*	*Descrizione*
Memoria Condivisa	`ch_sym_sm_no`	Utilizzo non ottimizzato della cache
	`ch_sym_sm` `ch_asymin_sm`	Allocazione con massima località in cache
	`ch_sym_sm_nullack`	Utilizzo del protocollo di comunicazione Null-Ack
UDN	`ch_sym_udn` `ch_asymin_udn`	Uso esclusivo della rete di interconnessione

Tabella 2: Canali di comunicazione implementati i due supporti architetturali

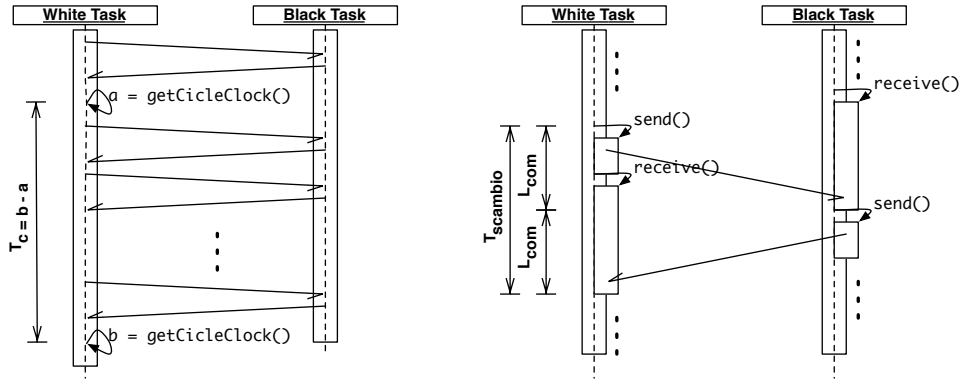


Figura 9: Rappresentazione della sequenza di azioni svolte dai due processi dell'applicazione di misurazione. A sinistra viene mostrato il comportamento complessivo, a destra il dettaglio di uno scambio di messaggi.

4.1.1 L'applicazione di misurazione

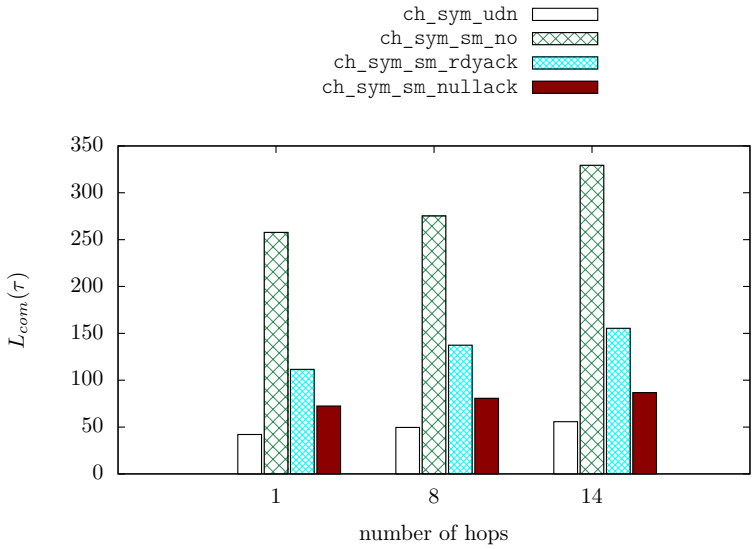
Al fine di valutare la latenza di comunicazione (L_{com}) dei canali si misura il tempo di completamento (T_C) di una applicazione di tipo "ping-pong" caratterizzato da due processi comunicanti mediante due canali con direzione una opposta a quella dell'altro. I due canali hanno stessa forma e stessa 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 9. Per 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:

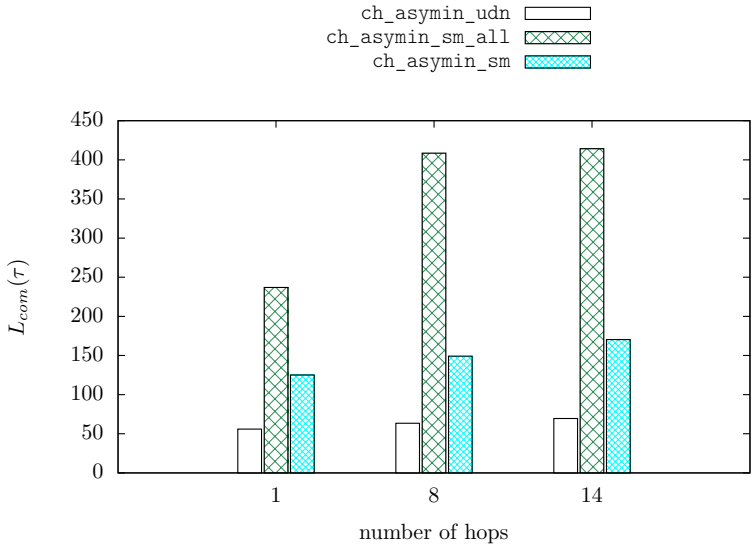
$$T_{scambio} = \frac{T_C}{n} \qquad L_{com} = \frac{T_{scambio}}{2}$$

4.1.2 Risultati

Il programma di misurazione effettua m scambi di messaggi, l'eseuzione e la corrispondente misurazione del tempo di completamento degli scambi è stata eseguita per diverse configurazioni di allocazione dei processi nei processori, a distanze diverse nella mesh. 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.



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



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

Figura 10: 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.

Implemenatation	*#Hops*	L_{com} (τ)		
		Avg	*Std Dev*	*Max*
`ch_sym_udn`	1	42.066550	0.097822	42.262000
	8	49.561120	0.104196	49.769500
	14	55.722760	0.107943	55.792050
`ch_sym_sm_no`	1	257.800490	0.008955	257.815900
	8	275.320380	0.006685	275.327650
	14	329.332180	0.018220	329.363400
`ch_sym_sm`	1	111.482450	0.187050	111.653200
	8	137.378440	0.016294	137.395550
	14	155.377520	0.013281	155.393950
`ch_sym_sm_nullack`	1	72.359500	0.219423	72.549500
	8	80.700840	0.124463	80.827350
	14	86.759230	0.118203	86.826250

Tabella 3: Risultati numerici, in cicli di clock, della misurazione della latenza di comunicazione asimmetrica in ingresso delle diverse implementazioni.

Implemenatation	*#Hops*	L_{com} (τ)		
		Avg	*Std Dev*	*Max*
`ch_asymin_udn`	1	56.025529	0.000941	56.026905
	8	63.525349	0.000979	63.526575
	14	69.527121	0.001011	69.528720
`ch_asymin_sm_all`	1	230.570268	0.028743	230.643940
	8	408.779285	0.047859	408.846645
	14	414.374837	0.025326	414.417200
`ch_asymin_sm`	1	125.039207	0.001632	125.041220
	8	149.041103	0.001936	149.045050
	14	170.037299	0.002036	170.040285

Tabella 4: Risultati numerici, in cicli di clock, della misurazione della latenza di comunicazione asimmetrica in ingresso delle diverse implementazioni.

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 processori 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 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 7.

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 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

```
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 7: Algoritmo sequenziale del calcolo matrice per vettore

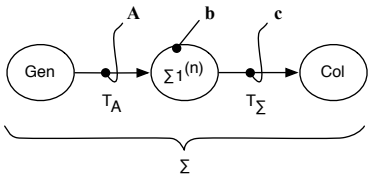


Figura 11: Rappresentazione compatta del grafo della computazione. L'elemento Σ1 è un modulo sequenziale o un sottosistema parallelo che esegue la moltiplicazione per **b** su ogni elemento della matrice.

al tempo di interarrivo T_A. Dato che il tempo di servizio della computazione Σ è superiore a quello ideale (il tempo di interarrivo dello stream), 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 *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 sottositema 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).

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

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

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

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

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

Tempo di completamento dello stream è definita come il tempo medio impegato 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 usanto l'implementazione parallela con n processi rispetto all'implementazione sequenziale. Può essere espressa anche in termini del tempo di completamento.

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

$$s_{\mathrm{id}}{}^{(n)} = \frac{\mathrm{T_{calc}}}{\mathrm{T_{\Sigma\_id}}} = \frac{\mathrm{T_{calc}}}{\frac{\mathrm{T_{calc}}}{n}} = n \tag{7}$$

4.2.2 Il metodo di parallelizzazione scelto

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 distibuzione di ciascun elemento dello stream ai processi worker, piuttosto che una distibuzione 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 7) 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 8: 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 escuzione $O(n)$ 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 **b**.

Si adotta la replicazione del vettore **b** nei processi worker, ciò è possibile in quanto tale oggetto viene acceduto in sola lettura. Di conseguenza l'implementazione 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. Al fine di ottimizzare le prestazioni complessive della computazione è stata adottata una strutturazione ad albero dell'operazione di multicast; questa è caratterizzata da un tempo di servizio costante pari alla latenza di comunicazione del canale simmetrico per l'arietà dell'albero, e da una latenza proporzionale al logaritmo del numero di nodi destinatari. L'implementazione scelta è un albero binario mappato sull'insieme dei moduli worker della Map; ciò significa che i moduli che eseguono il calcolo della Map sono collegati tra loro in un sottografo ad albero binario. Prima di avviare il proprio calcolo su un elemento dello stream prendono parte alla comunicazione multicast dell'elemento stesso, che quindi è realizzata in modo distribuito su tale insieme di moduli. 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

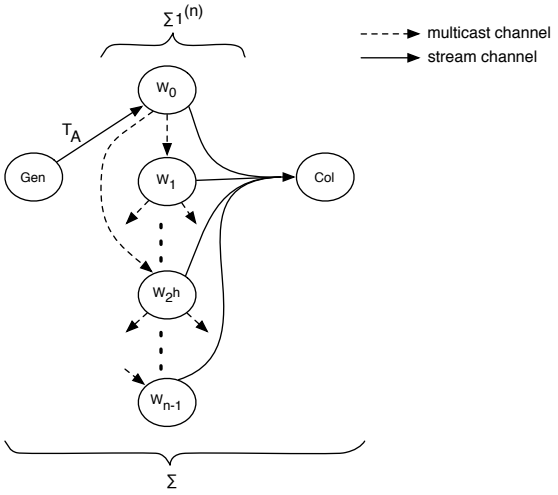


Figura 12: Grafo della computazione Map ($\Sigma 1^{(n)}$) che è collegata allo stream e che fa uso della multicast strutturata ad albero binario mappato sui moduli worker

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 tiles per l'esecuzione delle comunicazioni[2].

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.

$$\begin{aligned} {T_{\Sigma 1\_\text{id}}}^{(n)} &= {T_S}^{(n)} = T_{\text{multicast}} + \frac{T_{\text{calc}}}{n} + T_{\text{gather}} \\ &= 2 \cdot T_{\text{sym\_send}} + \frac{T_{\text{calc}}}{n} + T_{\text{asymin\_send}} = \Delta_{\text{com}} + \frac{T_{\text{calc}}}{n} \qquad (8) \\ n_{\text{opt}} &= \left\lceil \frac{T_{\text{calc}}}{T_A - \Delta_{\text{com}}} \right\rceil \qquad (9) \end{aligned}$$

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

[2]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.

4.2.3 Descrizione dell'implementazinoe

L'applicazione considerata è fittizia, ovvero non esiste un dispositivo, o nodo di elaborazione, che produce lo stream, ne esiste quello che lo riceve. Al fine di poter eseguire l'applicazione sono stati realizzati due processi, eseguiti in modo esclusivo su due tile 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 processore all'esecuzione del processo `main` dell'applicazione, e due processori 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 nodo 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 tile, i processi worker sono mappati in modo consecutivo a partire dal terzo tile, il processo main è mappato sull'ultimo tile 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 nodi 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 implementazini 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.

4.2.4 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.

Dato che siamo interessati a computazioni di grana fine si sono considerare matrici con un numero di righe relativamente basso: $\mathrm{M} \in \{\, 56, 168, 280 \,\}$. 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.

A differenza di una applicazione reale, non è imposto il tempo di interarrivo. Il nostro scopo non è quindi trovare il grado di parallelismo ottimo per una certa computazione su stream; piuttosto si vuole capire quale sia la scalabilità dell'applicazione all'aumentare di n, confrontare il tempo di servizio con quello atteso e confrontare queste misure tra le esecuzioni che usano le due configurazioni di supporti.

- misuro il tempo di servizio dimensionando il tempo di interarrivo con il valore del tempo di servizio ideale. La stima del tempo di servizio ideale usa il valore di Tcalc e di Δ_{com}misurati con uno o due processi, e diversi dal rispettivo valore con un grado di parallelismo diverso (non sono costanti ma dipendono da n). Da questa misurazioe ne ricavo anche la scalabilità, e osservo che il sistema rimane collo di bottiglia, soprattutto per grana fine.

- Posso fare i confronti dei tempi di servizio tra le due impelmentazioni del supporto.

- Devo spiegare come mai non si raggiunge il tempo di servizio ideale, il sistema resta un collo di bottiglia quando non dovrebbe esserlo, dai calcoli fatti: ipotizzo che il tempo di calcolo e il tempo di multicast non siano costanti al variare di n; dimostro ciò misurando questi tempi:

  - misuro il tempo di calcolo di un singolo prodotto scalare, facci vedere

Matrix Size	T_{calc} (μs)	T_{calc} (τ)
56x56	85.997340	74351.900000
168x128	848.096504	733250.424000
280x280	2360.060404	2040469.784000

(a) Misurazione del tempi di elaborazione del programma di calcolo sequenziale al variare della dimensione della matrice.

Canali usati	$\mathrm{T}_{\Delta_{\mathrm{com}}}(\tau)$
UDN only	180.95523
SM only	481.04016
SM only, with all senders	724.91273

(b) Valori ideali della somma dei tempi impiegati nelle comunicazioni dai processi worker (misure con #hops = 14).

Tabella 5: Valori ideali dei tempi di calcolo e di comunicazione

che per matrici di dimensione piccola tale tempo aumenta notevolmente con il grado di parallelismo.

    * per essere ancora più esaustivo mostro le misure con datai float: la grana è più grande e il tempo di calcolo di un singolo prodotto scalare non aumenta molto al variare di n

  – allo stesso modo prendo il tempo di multicast nel primo worker quando la Map non è collo di bottiglia, e faccio vedere che tale tempo aumenta con l'aumentare di n.

    * Quale tempo di interarrivo scelgo per le diverse dimensioni di M?
    * si osserva che con tempi di interarrivo più stretti il tempo di multicast ha valori più alti.

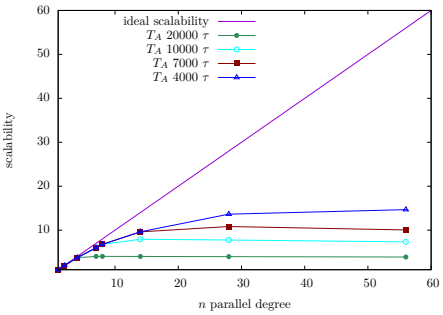
- Un'altra osservazione sul grafico del tempo di calcolo di un singolo prodotto scalare: confrontando le due implementazioni noto che l'aumento di quella UDN è inferiore all'aumento di quella SM, questo mi dice che UDN riduce i conflitti alla memoria condivisa, ed è anche per questo che funziona meglio rispetto alla SM, con grana fine.

4.2.5 Risultati delle misurazioni

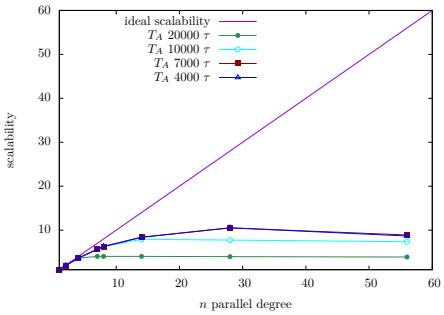
Figura 13: Grafici di scalabilità del tempo di completamento dello stream al variare del tempo di interarrivo

(a) Implementazione con solo UDN

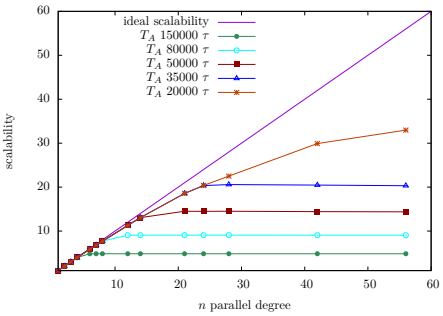
(b) Implementazione con solo SM



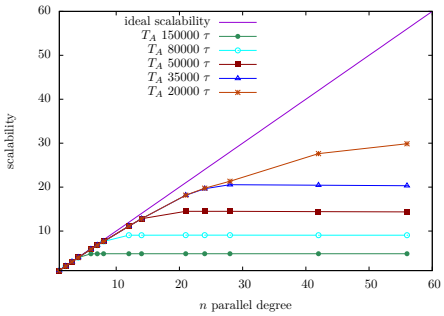
(a1) Scalabilita dell implementazione UDN con M=56 al variare del tempo di interarrivo



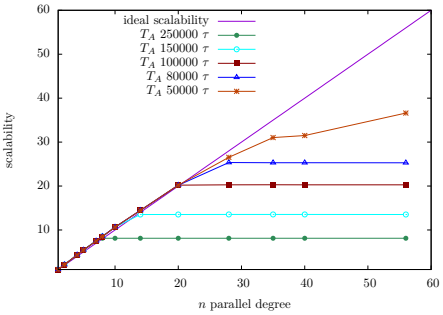
(b1) Scalabilita dell implementazione SM con M=56 al variare del tempo di interarrivo



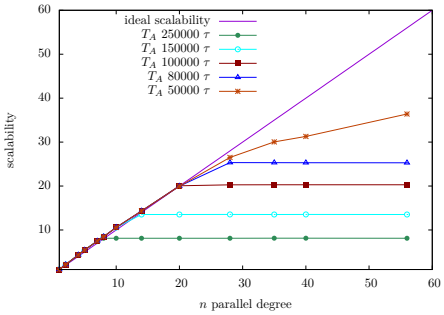
(a2) Scalabilita dell implementazione UDN con M=168 al variare del tempo di interarrivo



(b2) Scalabilita dell implementazione SM con M=168 al variare del tempo di interarrivo



(a3) Scalabilita dell implementazione UDN con M=280 al variare del tempo di interarrivo

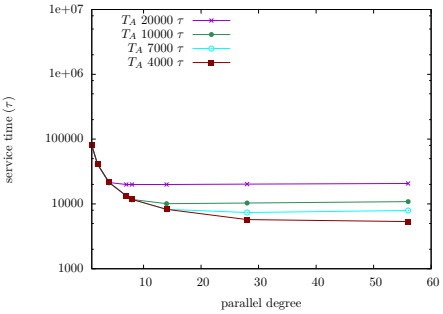


(b3) Scalabilita dell implementazione SM con M=280 al variare del tempo di interarrivo

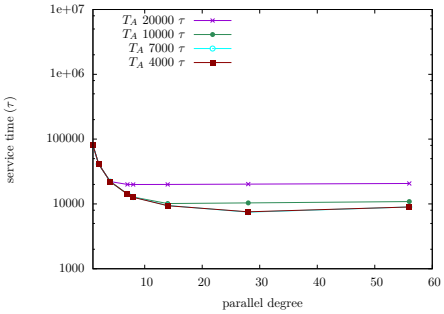
Figura 14: Grafici del tempo di servizio al variare del tempo di interarrivo

(a) Implementazione con solo UDN

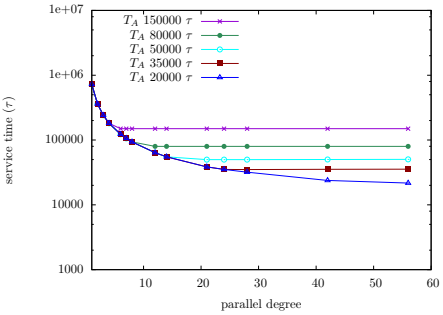
(b) Implementazione con solo SM



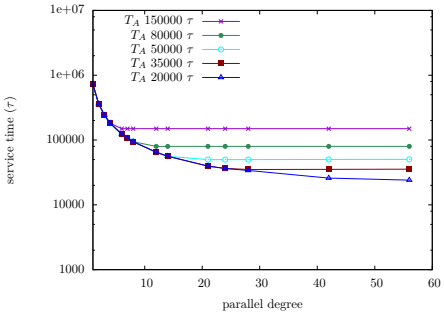
(a1) Tempo di servizio dell'implementazione UDN con M=56 al variare del tempo di interarrivo



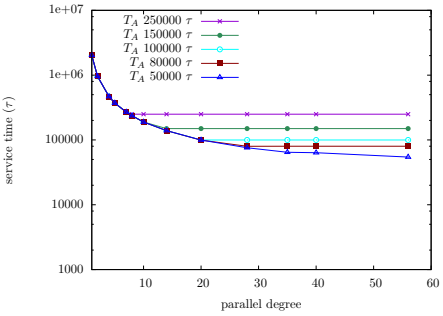
(b1) Tempo di servizio dell'implementazione SM con M=56 al variare del tempo di interarrivo



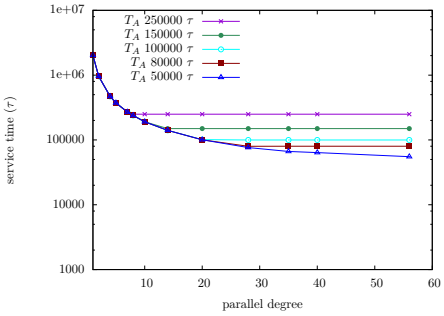
(a2) Tempo di servizio dell'implementazione UDN con M=168 al variare del tempo di interarrivo



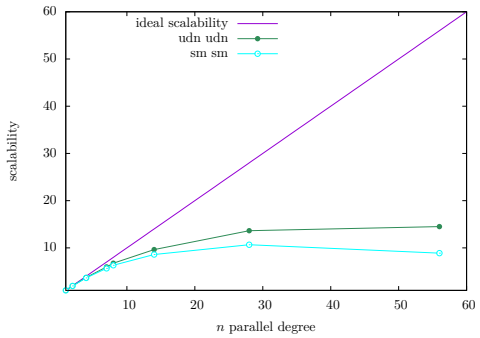
(b2) Tempo di servizio dell'implementazione SM con M=168 al variare del tempo di interarrivo



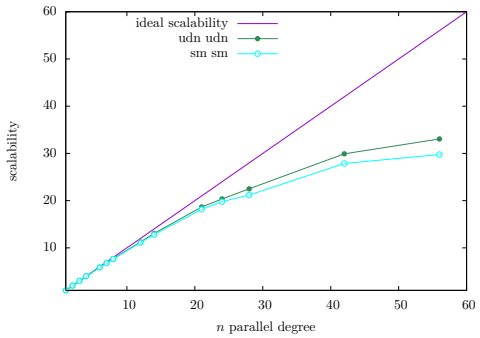
(a3) Tempo di servizio dell'implementazione UDN con M=280 al variare del tempo di interarrivo



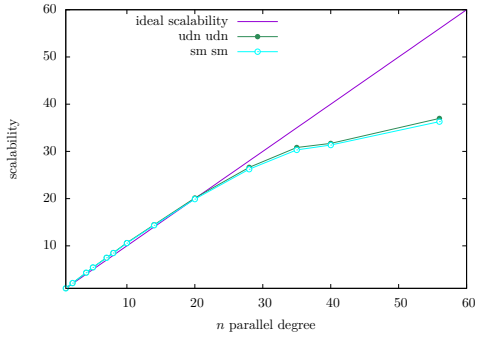
(b3) Tempo di servizio dell'implementazione SM con M=280 al variare del tempo di interarrivo



(a) Confronto della scalabilita nelle diverse implementazioni, Ta=181, M=56

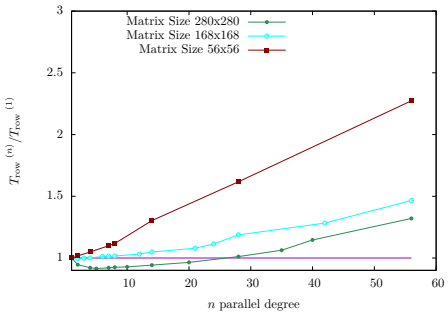


(b) Confronto della scalabilita nelle diverse implementazioni, Ta=181, M=168

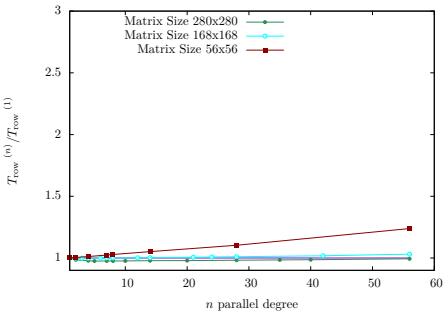


(c) Confronto della scalabilita nelle diverse implementazioni, Ta=181, M=280

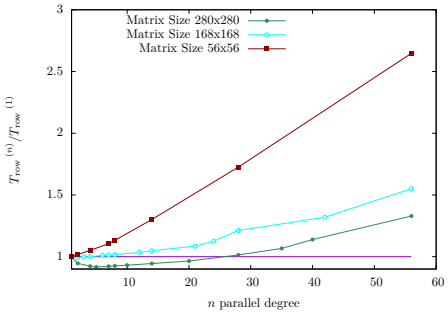
Figura 15: Rapporto tra i tempi di calcolo di un singolo prodotto scalare



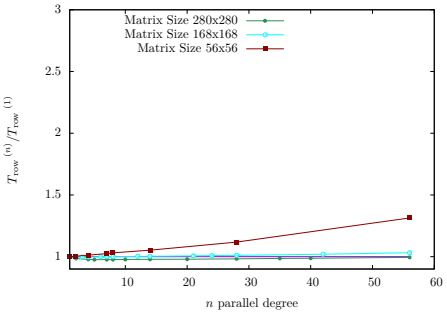
(d) Tempi di calcolo di una singola Row · Col con Ta=4000, canali UDN e dati Int



(e) Tempi di calcolo di una singola Row · Col con Ta=4000, canali UDN e dati Float



(f) Tempi di calcolo di una singola Row · Col con Ta=4000, canali SM e dati Int

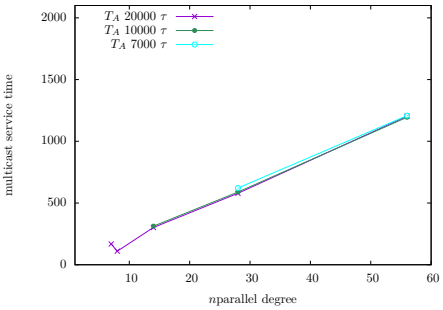


(g) Tempi di calcolo di una singola Row · Col con Ta=4000, canali SM e dati Float

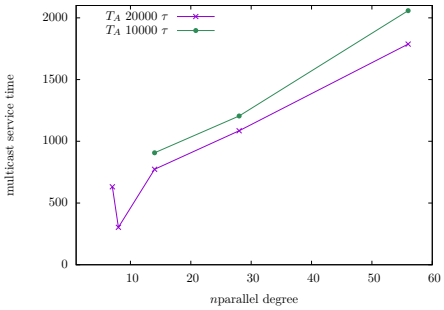
Figura 16: Grafici del tempo di multicast al variare del tempo di interarrivo

(a) Implementazione con solo UDN

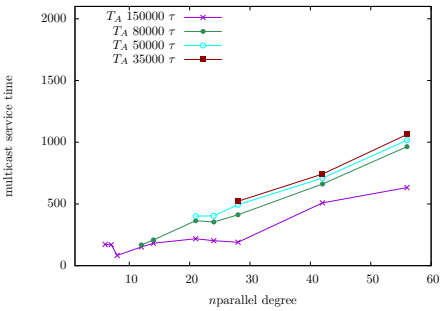
(b) Implementazione con solo SM



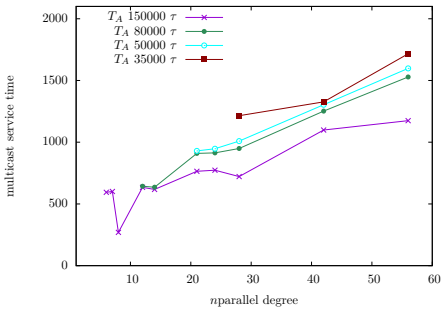
(a1) tempo di multicast dell implementazione UDN con M=56 al variare del tempo di interarrivo



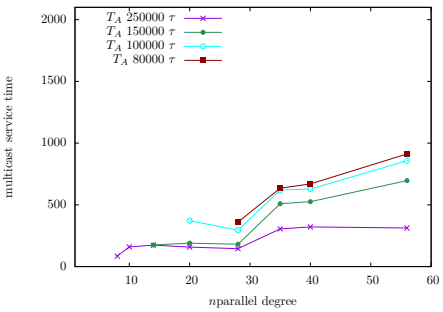
(b1) tempo di multicast dell implementazione SM con M=56 al variare del tempo di interarrivo



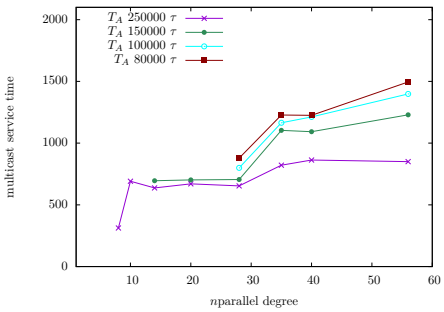
(a2) tempo di multicast dell implementazione UDN con M=168 al variare del tempo di interarrivo



(b2) tempo di multicast dell implementazione SM con M=168 al variare del tempo di interarrivo



(a3) tempo di multicast dell implementazione UDN con M=280 al variare del tempo di interarrivo



(b3) tempo di multicast dell implementazione SM con M=280 al variare del tempo di interarrivo

5 Conclusioni

References

[1] 11th IASTED International Conference on Parallel Distributed Computing and Networks, Innsbruck, Austria. Evaluation of Architectural Supports for Fine-Grained Synchronization Mechanisms, 2013.

[2] 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.

[3] Tilera. UG101 - Tile Processor User Architecture Manual.

[4] Tilera. UG205 - Programming the Tile Processor.

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

[6] 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.