Modelling Communications in Cache Coherent Systems

Sabela Ramos Garea
Torsten Hoefler

March 8, 2013

Contents

A Communication Model for Cache-Coherent Systems

The aim of this work is to provide an analytical model to simplify the parallel algorithm design in cache coherent systems. The current trend in actual processors is to increase the number of cores per chip, motivated by the stop of frequency scaling, which makes it crucial to design efficient parallel solutions for shared memory. Multi-core processors are standard in current commodity machines and many-core accelerators are becoming increasingly popular. On the one hand, the GPUs (Graphical Processing Units) are being widely used to accelerate general purpose applications. However, the use of GPUs forces a change in the programming paradigm from latency-optimized to stream-optimized computing. On the other hand, projects like Intel MIC (Many Integrated Core) Architecture intend to provide an Intel `x86` based accelerator that allows the use of traditional techniques in order to take advantage of many-core parallelism. The latter group of accelerators usually provide cache coherency protocols to increase programmability, as multi-core processors do, and, generally, the only way to communicate cores in these architectures is relying on loads and stores to common memory locations and the cache coherency protocol.

In this scenario, we have developed a communication model to make it possible to design optimized algorithms in cache-coherent systems, and with this purpose, we have made a preliminary study of current multi-core architectures (Intel Sandy Bridge) and the recently released Intel Xeon Phi, which is the most scalable among the `x86` based architectures. We provide one-line communication models for these architectures and, then, we focus on the features of the Intel Xeon Phi to develop a comprehensive communication model and prove that it allows to design complex exchange algorithms in a 60-core environment.

Chapter 1

Development of Simple Cache Models

1.1 Selected Architectures

The architectures described in this Section are one of the last Intel multi-core system (Sandy Bridge) and the most scalable Intel `x86` based coprocessor (Intel Xeon Phi).

1.1.1 Intel Xeon Phi

The Intel Xeon Phi coprocessor is a many-core system based on the Intel MIC (Many Integrated Core) architecture. The current commercial Xeon Phi (5110P) has 60 simplified Intel CPU cores running at 1056 MHz and supports 4 threads per core with hyperthreading (thus, 240 threads in the die). Each core has a vector unit with 64 byte registers featuring a new vector instruction set known as Intel Initial Many Core Instructions (IMCI). The cache memory in each core is arranged in a 32 kb L1 data cache, 32 kb L1 instruction cache, and a private 512 kb L2 unified cache which is kept coherent by a distributed tag directory system (DTDs). Cores are arranged on a bidirectional ring bus that provides high scalability to which other components like memory controllers or tag directories are also connected.

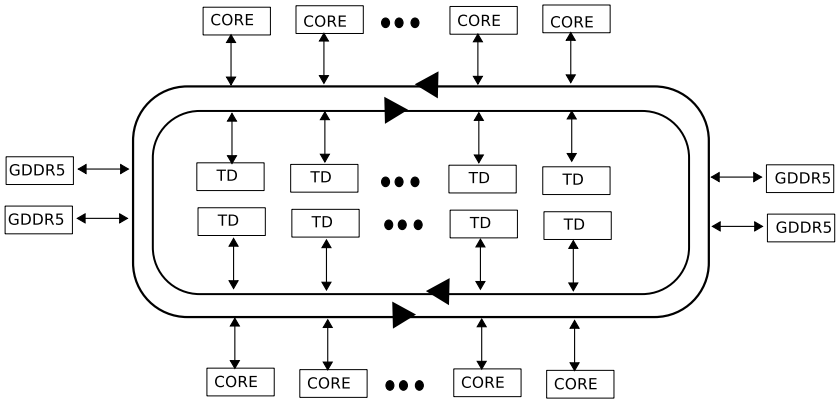


Figure 1.1: Architecture of the Intel Xeon Phi coprocessor

Figure 1.1 represents the basic architecture of the Xeon Phi including cores, bus, memory controllers and tag directories. The bidirectional ring has three independent rings in each direction [4]: a data block ring (64 bytes wide), an address ring (send/write commands and memory addresses) and an acknowledgment ring (flow control and coherency messages). There are 64 tag directories connected to the ring and the address-mapping to the tag directories is based on hash functions over memory addresses, leading to an even distribution around the ring. The memory controllers, also connected

to the ring, provide access to the GDDR5 memory (8 GB of global memory). The coprocessor runs a simplified Linux-based OS in one of the cores.

The main advantage of the Xeon Phi over other accelerators or coprocessors is that it provides the well-known `x86` ISA and memory model, hence the programming effort is just focused on how to better exploit performance, but it can be done with known techniques and languages such as OpenMP or MPI. Furthermore, Xeon Phi can be used as a mere coprocessor in which the host offloads code to be accelerated, or as an independent unit that runs a whole application or that communicates in a symmetric manner with the host [10, §6]

1.1.2 Intel Sandy Bridge

The multi-core architecture analyzed is an Intel Sandy Bridge Xeon E5-2670, at 2.60GHz, with a dual-socket configuration of eight-core processors with HyperThreading (HT) activated (see Figure 1.2) and connected by QPI (Quick Path Interconnect, 8 GT/s). L1 (32 kb data and 32 kb instructions) and L2 (256 kb unified) caches are private to each core. Each processor has a shared L3 cache (or LLC) of 20MB which is divided in eight slices. The internal components of the chip, including the LLC slices, are connected via a ring bus composed by a data ring, a request ring, an acknowledge ring and a snoop ring. Any core can use any of the cache slices, thus having access to data stored in any of them.

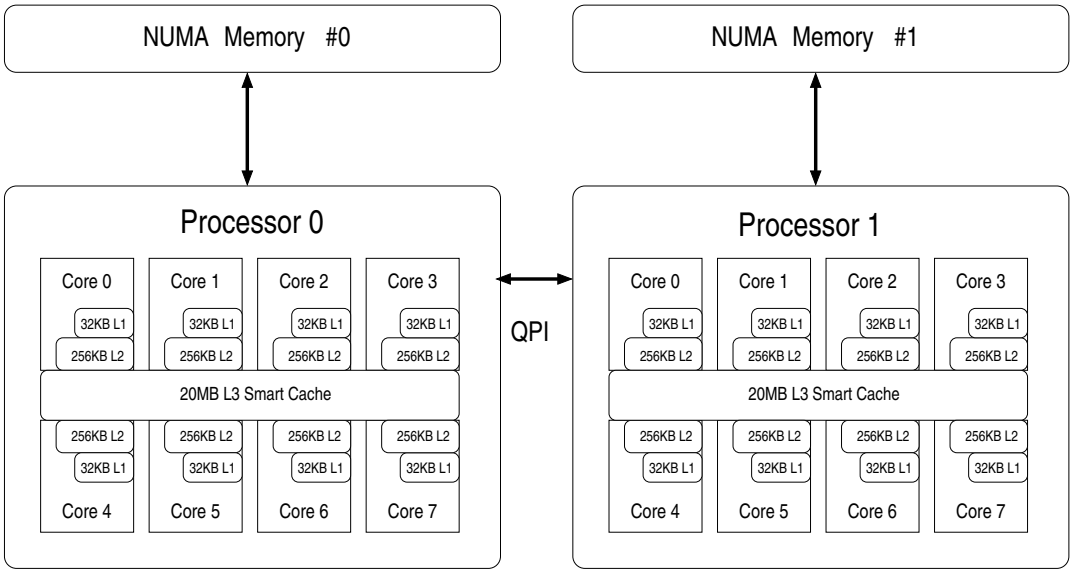


Figure 1.2: Xeon E5-2670 Sandy Bridge

1.2 Cache Coherency Protocols

Both Xeon Phi and Sandy Bridge use variations of the traditional MESI protocol [7], summarized in table 1.1.

Table 1.1: MESI protocol states

M	Modified	Only this core owns the line	It has been modified regarding memory
E	Exclusive	Only this core owns the line	It has not been modified regarding memory
S	Shared	Several cores can have the line	It has not been modified regarding memory
I	Invalid	The core does not own the line	

1.2.1 Extended MESI (Intel Xeon Phi)

The Xeon Phi's cache coherency protocol is a directory protocol based on MESI that uses GOLS (Globally Owned Locally Shared) to simulate a Owned state, thus allowing the share of a modified line. The goal is to avoid writebacks to memory when another core wants to read a modified line. Hence, in this protocol, the Shared state does not imply that the line has not been modified. Each core's cache maintains the MESI state of the lines that it holds (Table 1.2) and the Distributed Tag Directorys (DTDs) will hold the global GOLS coherency state of each line (Table 1.3). Lines are assigned to each DTD regarding the line address instead of the core that is holding or requesting the line.

Table 1.2: MESI protocol states

M	Modified	Only this core owns the line	It has been modified regarding memory
E	Exclusive	Only this core owns the line	It has not been modified regarding memory
S	Shared	Several cores can have the line	It may or may not have been modified regarding memory
I	Invalid	The core does not own the line	

Table 1.3: GOLS protocol states

GOLS	Globally Owned Locally Shared	Several cores can have the line	It has been modified regarding memory
GE/GM	Globally Exclusive/Modified	Only this core owns the line	It may or may not have been modified regarding memory (the core will have the line in M or E)
GS	Globally Shared	Several cores can have the line	It has not been modified regarding memory
GI	Globally Invalid	No core holds the line	

Each time a core has a cache miss, it will request the line to the correspondent DTD. This DTD will answer depending on the GOLS state of the line and will request memory or the core which owns the line to answer with the data. If another core has the line, it will acknowledge the DTD and send the data to the requester core, which will also acknowledge to the DTD that it has received the data. Then, the DTD will update the line state. Any eviction will also have to request the DTD for allowance before effectively evicting the line.

1.2.2 MESIF (Sandy Bridge)

MESIF is the snooping cache coherence protocol that is implemented in Intel Xeon Sandy Bridge systems [15]. It is based on the classical MESI (Modified-Exclusive-Shared-Invalidated) adding a Forwarding state to prevent multiple caches holding a shared line to answer when this line is requested by another core's cache (Table 1.4). When a line is in shared state, one of the cores will have this line in F and it will be the one that will answer the request. The last core requesting this line will become the new $forwarder$,and the last $forwarder$ will keep the line as S, preventing the eviction of the F line, since the last core requesting the line is most likely to maintain it.

Table 1.4: MESIF protocol states

M	Modified	Only this core owns the line	It has been modified regarding memory
E	Exclusive	Only this core owns the line	It has no modifications regarding memory
S	Shared	Several cores can have the line	It has no modifications regarding memory
I	Invalid	The core does not own the line	
F	Forward	Other cores can have the line in S state	It has no modifications regarding memory

1.3 Modeling cache transfers

This section analyzes the latency of cache line transfers among cores. First, we present the communication graph that results of applying the cache coherency protocol to a single cache line that is being read by two threads, T_0 and T_1 located in two different cores. The vertex represent the state of the cache line in each core's cache, and the edges represent the transitions between vertex with the operation that caused the transition and cost associated. To parametrize the costs, it is necessary to measure the raw latency of moving cache lines and we have used the memory benchmarks from the BenchIT suite [15]. Once we had obtained the latency results, we used them to simplify the modeling of communications in the cache coherency protocols of the architectures selected.

1.3.1 Cache Line Transfers Benchmark

The memory benchmarks included in the BenchIT suite have been used by Molka et al. in [15] to measure the time of transfers among caches in a Nehalem architecture. It is based on the existence of two running threads, T_0 and T_1. T_0 reads a line owned by T_1 and the benchmark is run varying the location of T_1, and the initial coherence state of the cache line. Hence, these benchmarks are able to provide a measurement of the cost of transferring one cache line depending on the coherence state and the cache-line location. The benchmark uses pseudo-random addresses within different pages of a buffer and then, 32 reads are performed using assembly code. The time is measured using the `RDTSC` counter and the time of a line transfer is estimated as the average of these 32 accesses. This benchmark is run several times, using the average of the runs as the final timing.

We have introduce small variations to be able to make a statistical analysis of the results (e.g. to obtain the standard deviation of the measurements). Moreover several inline assembly instructions were modified to be compatible with the Xeon Phi instruction set:

- *mfence* or memory fences are not supported in Xeon Phi, so we have used *lock; addl $0,(%%rsp)* instead.
- *clfush* is also not supported and it had to be replaced by *clevict* for both L1 and L2 caches.

The S and F states use an extra thread (T_2) that shares the line. It is located in a core where T_0 and T_1 are not running.

1.3.2 Notation

The operations that cause transitions to a cache line state are *read*, *RFO* (Read For Ownership, when the thread intends to write the line) and *evict*. While the first two represent fetches of lines, and can be modeled as read operations (R), the third one represents eviction of lines usually caused by external agents (an *RFO* from another thread or replacement of cache lines) and are not modelled. Writes to memory are only performed on eviction.

We will model each operation as $R_{\mathcal{L},\mathcal{S}}$, indicating the location from where the line is read ($\mathcal{L}$) and the current state of the line ($\mathcal{S}$). The location can take the values local (L), when the line is in the reader's cache, and remote (R) for an extra-core location. Moreover, on Sandy Bridge, when specifying if the remote location is within the same processor, it can take the value P, and if it is another processor but within the same node, N is used to indicate location.

1.3.3 Communication Model for Intel Xeon Phi

Figure 1.3 represents the costs of the transfer of a cache line between two threads (T_0 and T_1) that are running in different cores. As explained before, each vertex represents the state of the line in each of the two cores, while the edges represent the transitions.

The possible states of a line are M (modified), E (exclusive), S (shared) and I (invalid). Each vertex would be composed by the combination of two of these states to represent the cache line in each of the cores involved. Some vertex were discarded as impossible combinations of states: (E,M), (E,E), (E,S), (M,M), (M,E), (M,S), (S,M) and (S,E).

Each transition is labelled with the required action (in the form $T_i, action$) and the cost associated. Dotted edges represent external actions by a third thread (T_2) or evictions, thus, this edges do not

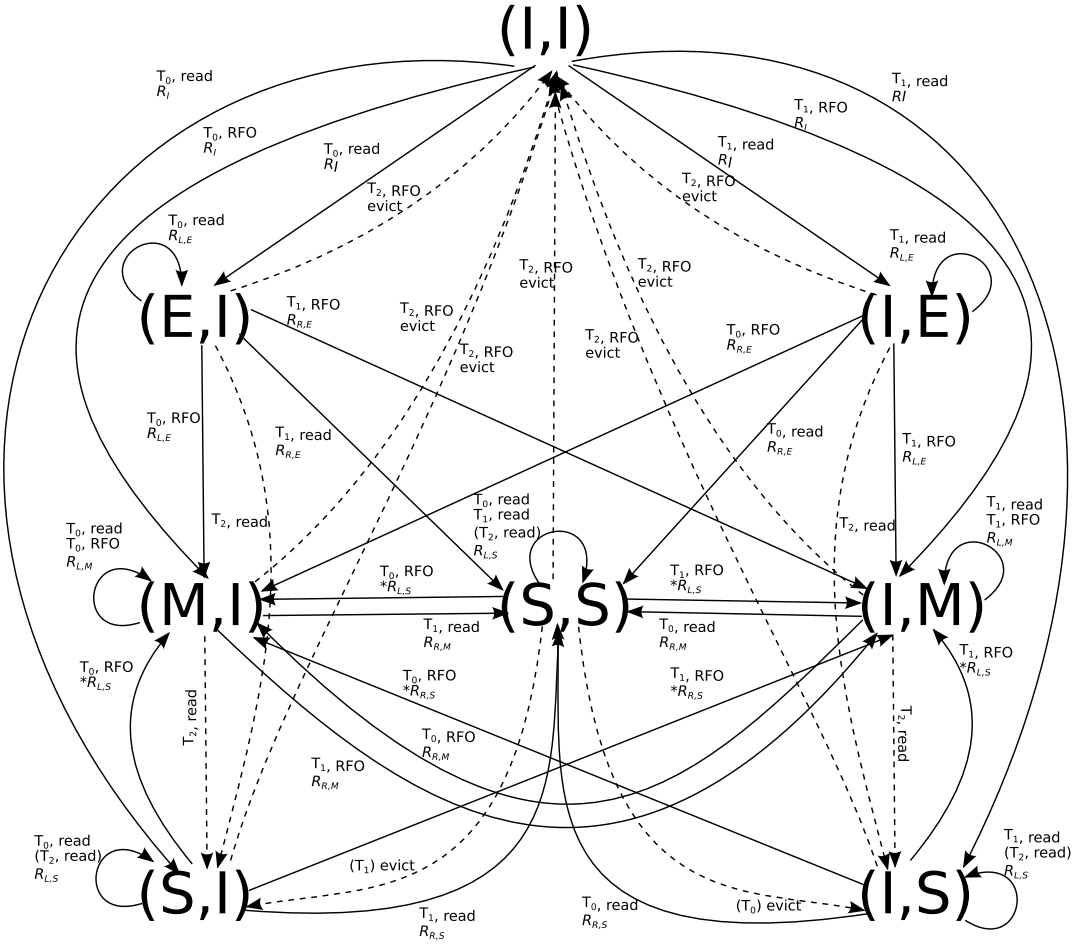


Figure 1.3: Graph of the MESI transitions of a line within two cores

Table 1.5: Cost in nanoseconds of reading a line within a core and between two cores

(a) Full model

Label	**Cost**
$R_{L,M}$	8.6
$R_{L,E}$	8.6
$R_{L,S}$	8.7
$R_{R,M}$	234.7
$R_{R,E}$	235.8
$R_{R,S}$	233.4
R_I	277.7

(b) Simplified model

Label	**Cost**
$R_{L,*} = R_L$	8.6
$R_{R,*} = R_R$	235.8
R_I	277.7

Table 1.6: Results of the BenchIT memory latency test on Intel Xeon Phi (nanoseconds)

	Same Core		Adjacent cores		Middle distance		Largest distance	
	avg	stdev	avg	stdev	avg	stdev	avg	stdev
M	8.600	0.227	241.218	21.718	234.702	25.617	240.098	10.377
E	8.603	0.246	227.412	20.637	235.796	25.486	237.370	27.731
S	8.662	0.883	232.002	10.190	233.363	34.959	233.371	22.523
I	277.650	34.016	274.345	25.222	278.818	34.439	284.503	29.582

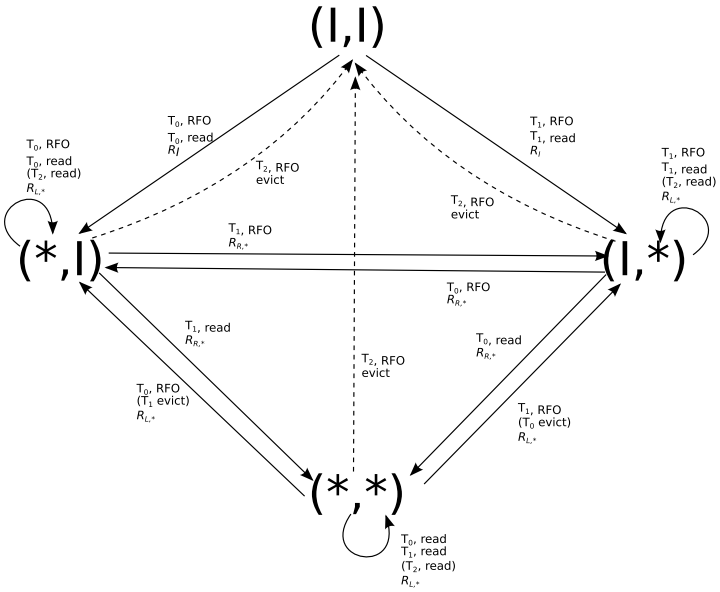


Figure 1.4: Graph of the simplified MESI transitions of a line within two cores

have any associated cost. The actions can be *read*, *RFO* (when the thread intends to write the line) and *evict*. When several actions appear next to one edge, it indicates that any of them can be taken to reach the next vertex. All the costs are estimated by the latency of a read and represented by $R_{\mathcal{L},\mathcal{S}}$, where $\mathcal{L}$ is the location of the line (Local for the own cache of the thread performing the action, and Remote if the line is in other core's cache), and $\mathcal{S}$ is the state of the line before the transition. When $\mathcal{S} = I$, the location is not relevant since the line has to be always fetched from memory.

The symbol $*$ preceding some of the costs indicate that there can be an overhead as a consequence of the need to invalidate the line in other caches. This is a consequence of estimating every cost by a read and occurs on an RFO of a shared line. With a modified or exclusive line the target of invalidation is well-defined (the current owner), but with a shared line there can be multiple cores holding the line.

The values of the transition costs are included in Table 1.5a and were extracted from Table 1.6. For the reads between different cores, the "Middle Distant" column has been selected. For the reads of invalid lines, it is used the value of the column "Same Core". The selection has been arbitrarily made since there is no real difference between the latency among different cores or the latency of fetching a line from memory depending on the core that owns the line.

The full model in Table 1.5a indicates that some of the states in Figure 1.3 can be collapsed due to nearly identical transition costs. Hence, we can form three groups of operations, local reads, remote reads, and reads of invalid lines. Table 1.5b and Figure 1.4 summarize the simplified model for Intel Xeon Phi.

1.3.4 Communication Model for MESIF (Sandy Bridge)

In the MESI protocol (Figure 1.5), when a core requests a line that is shared, every core that has the line answers to the request. The MESIF protocol extends traditional MESI adding the Forward state to indicate which one of the cores holding the shared line is in charge of answering the requests performed by other cores. The last core requesting the shared line is the one that will have it in F, taking advantage of temporal locality and assuming that this core has less probabilities of evicting it.

Thus, in the MESIF protocol, states (S, S), (S, I) and (I, S) are complemented with (F, S), (S, F), (I, F) and (F, I). Figure 1.6 represents the transitions including this combinations but without representing all the transitions for clarity purposes (they are depicted in Figure 1.5).

The main difference regarding the extended MESI from Xeon Phi is that the S or F state represent lines that are consistent with memory, that is, with no modifications. Thus, modified lines can not be shared. Within a processor, a modified line can be transferred among cores using the L3 cache, and

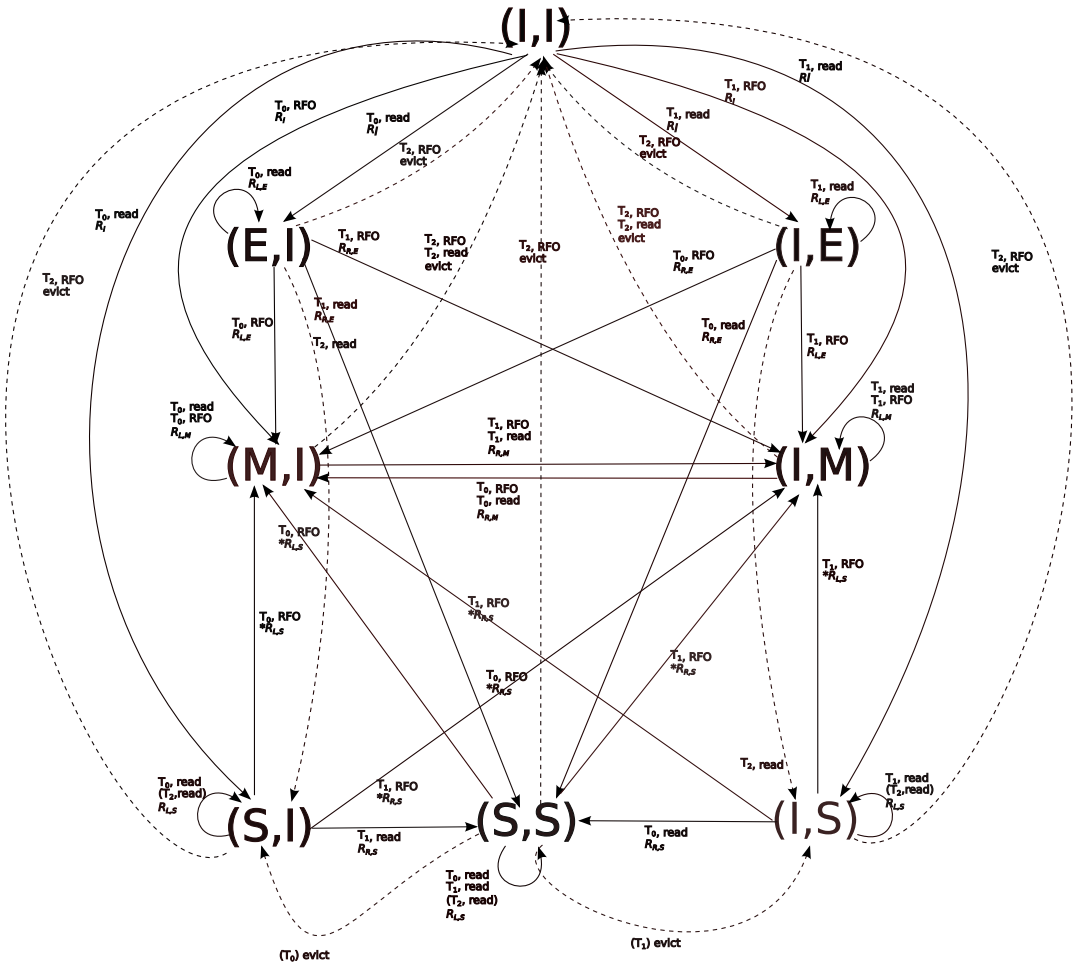


Figure 1.5: Graph of the MESI protocol transitions of a line within two cores

it could be transferred also through QPI to another processor but, eventually, a write-back will have to be performed. The other great difference is the F state. One remarkable issue is that when the F line is evicted, there is no core responsible for answering a request, so the line must be fetched from memory again. However, this is highly unlikely to happen since the core that has it in F is the last one that has requested it.

The results on Sandy Bridge using BenchIT, in nanoseconds, are shown in Table 1.7. For each possible state of a cache line, results include the average and the standard deviation of the latency of a transfer inside one core, between two cores within the same processor and between cores from different processors.

Table 1.7: Results of the BenchIT memory latency test on Sandy Bridge (nanoseconds)

	Same Core		Same processor		Diff. processors	
	avg	stdev	avg	stdev	avg	stdev
M	2.3	0.07	41.6	3.63	115.4	5.15
E	2.3	0.04	32.8	2.83	75.7	3.72
S	2.8	0.87	15.8	2.13	17.3	2.67
I	69.3	6.98	70.9	4.11	109.3	5.05
F	2.3	0.02	15.1	0.93	16.0	2.58

With the results obtained we can derive the following clusterized table (Table 1.8) using the same notation as in Section 1.3.3, but introducing new locations: $R_{P,\mathcal{S}}$ for another core within the same

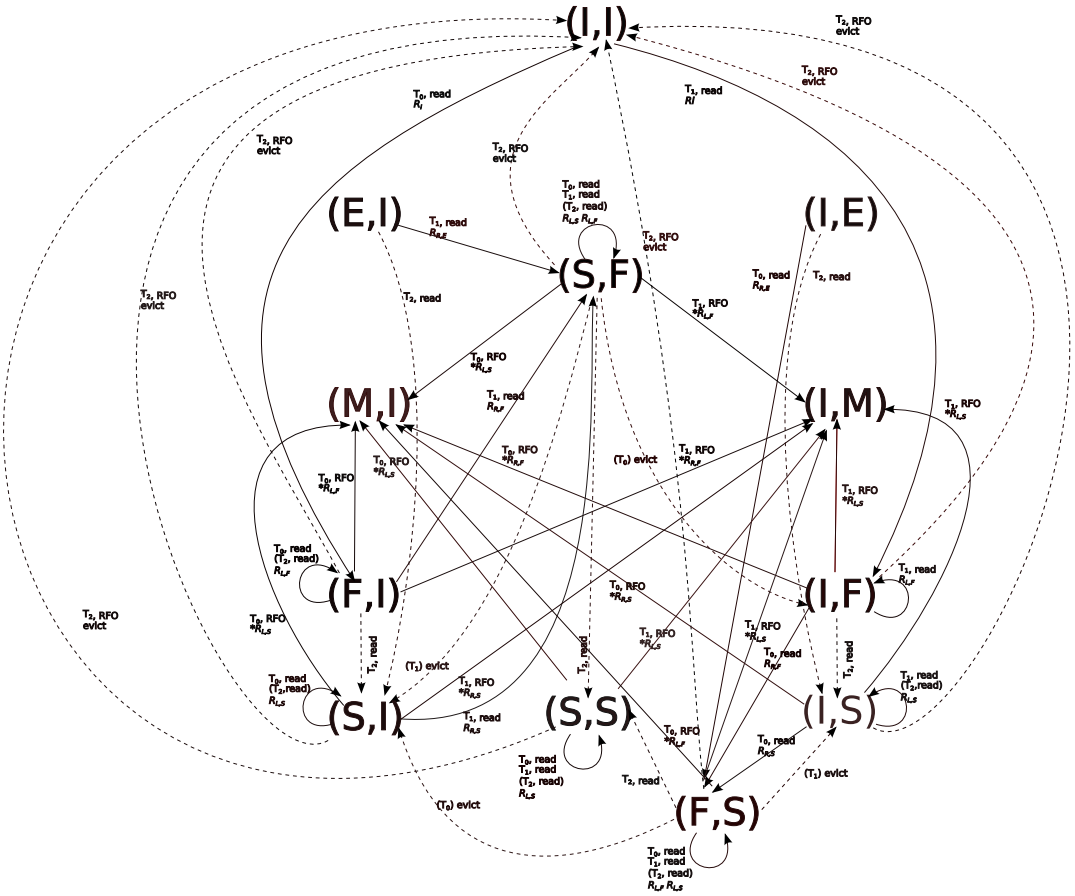


Figure 1.6: Graph of the S and F MESIF protocol transitions of a line within two cores

processor and $R_{N,\mathcal{S}}$ referring to another core located in a different processor. As it happened for Xeon Phi, there is no difference for cached lines when both threads are in the same core except for some variability that appeared for the S state. Regarding invalid lines, there is no difference if threads are in the same or different cores within the same processor. However, the fact that a modified line can not be shared and that it has to be evicted from one cache causes some delays when comparing with sharing an exclusive line, thus, it is not possible to collapse these states into one. However, timing for S or F lines are almost the same.

The largest differences appear when communicating cores from different processors. In this scenario, the latencies can be clusterized in a similar way that we did for the same processor but with an extra overhead due to the transfers across QPI. The only exception is for shared lines. However this is a direct consequence of the benchmark design. In this scenario, T_0 reads a line that T_1 is sharing with a third thread that, in our case, is located in the same processor than T_0. The similar results obtained for S and F leads us to believe that each processor has one cache with the line in F, instead of having a global forward state maintained by the QPI links, because the owner of the local shared copy is the one answering the request, instead of the $forwarder$ located in another processor.

Table 1.8: Results of the BenchIT memory latency test on Sandy Bridge (nanoseconds)

	Same Core	Same processor	Diff. processors
M	$R_L = 2.3$	$R_{P,M} = 41.6$	$R_{N,M} = 115.4 \simeq R_{P,M} + 70$
E		$R_{P,E} = 32.8$	$R_{N,E} = 75.7 \simeq R_{P,E} + 40$
S		$R_{R,S} = 15.7$	$R_{N,S} = 17.3 \simeq R_{R,S}$
F			
I	$R_{P,I} = 69.3$		$R_{N,I} = 109.3 \simeq R_{P,I} + 40$

Chapter 2

Single-line Ping-Pong Modeling

Once we have defined and analyzed the effects of the cache coherency protocol and parametrized the cost of basic transitions, we are going to analyze the behavior of the cache coherency protocols in a simple communication scenario.

This scenario consists of a simple ping-pong in which two threads (T_0 and T_1) interchange buffers of one cache line in a ping-pong manner. That is, T_0 sends one line to T_1 copying it into the receiver buffer of T_1 and then, T_1 sends another line to T_0, measuring the cost of half of the round-trip time (RTT). The benchmark uses two buffers on each thread, as represented on Figure 2.1, a send-buffer and a recv-buffer. The sender thread copies a byte from its send-buffer to the recv-buffer of the other thread. The receiver thread is polling over this byte in its receiver buffer to check whether it has receive the message or not, using the Canary-Protocol family to reduce the number of notification messages needed [9].

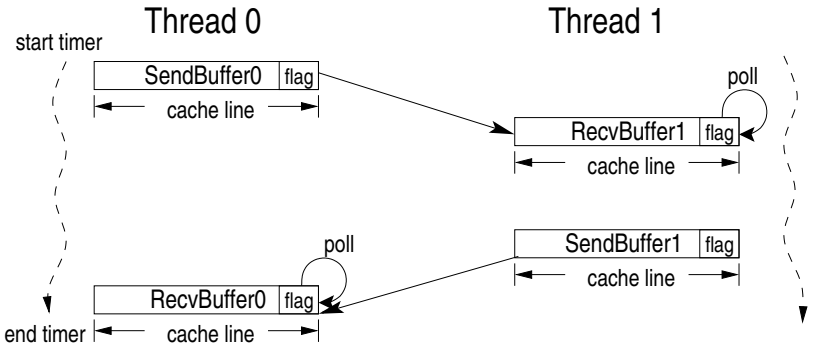


Figure 2.1: Ping-Pong test between two threads using four buffers

The ping-pong is run 5000 times accessing one line from a buffer in a pseudo-random sequence and the final time is the average of the individual measures.

The desired coherency state for each line is established before each ping-pong interchange. A S or F state indicates that the line is shared between the two threads performing the pin-pong. Considering the symmetry of the benchmark, send-buffers from both threads will have the same cache coherence state ($\mathcal{S}_s$), and so do the recv-buffers ($\mathcal{S}_r$). We discarded the case in which the recv-buffer line is invalid, because since the receiver is polling on it, it can not be assured that it is invalidated when the sender is performing the copy.

Moreover, having the send-buffer lines on cached states (M, E, S or F) should not make any difference. This buffer is only read by the sender and, in these four cases, the line is already placed in its L1 cache. However, for the recv-buffers on the MESIF protocol the access times are different for the different cached cases, while on MESI (as seen in the clusterized model, Table 1.5) there should be no difference. As a consequence, it is expected that it will be possible to distinguish among two groups of states for the send-buffers: (M,E,S,F) (or M,E,S for extended MESI) and (I); three for the recv-buffers in MESIF (discarding the I state): (M), (E) and (S,F); and only one (M,E,S) for

extended MESI. Finally, when using two threads within a core, the S and F states have also been discarded since both threads are using the same cache and the line will never achieve these states.

2.1 Estimation of the Costs

The ping-pong operation is performed through cache line transfers, thus it should be possible to find an estimation of the latency based upon the results obtained with BenchIT. The operations follow the steps depicted in Figure 2.1.

1. First of all, T_0 reads its buffer ($R_{L,\mathcal{S}_s}$)
2. and copies it into T_1's receiver buffer ($R_{R,\mathcal{S}_r}$ if T_1 is in a remote core and $R_{L,\mathcal{S}_r}$ if T_0 and T_1 share the same cache).
3. Finally, T_1 reads its recv-buffer that has been modified by T_0 ($R_{R,M}$ or $R_{L,M}$ depending on T_1 relative location).

Then, the same operation is performed with inverse roles. Since we use half of the RTT, we can derive Equation (2.1), where the term O stands for an overhead that might be introduced by the increment of coherency traffic due to having two active communicating threads, and $\mathcal{T}_1$ indicates that it is measuring the latency of a single-line ping-pong. This equation shows the estimation when both threads are running in different cores. To represent the scenario when both are running within the same core, the R_R must be changed to R_L.

$$\mathcal{T}_1 = \frac{t2 - t1}{2} = \frac{RTT}{2} \simeq R_{L,\mathcal{S}_s} + R_{R,\mathcal{S}_r} + R_{R,M} + O \tag{2.1}$$

2.2 Statistical Analysis of the Results

Each result is composed by the average and the standard deviation of 5000 samples, assuming a Gaussian Distribution. The standard deviation of the estimations has been calculated as shown in Equation (2.2), assuming independency (covariance = 0). Subscripts 0, 1 and 2 refer to the first three terms of the right side of Equation (2.1).

$$\begin{aligned} \mu_{estim} &= \mu_0 + \mu_1 + \mu_2 \\ \sigma_{estim} &= \sqrt{\sigma_0^2 + \sigma_1^2 + \sigma_2^2} \end{aligned} \tag{2.2}$$

Ideally, the overhead from equation (2.1) would be 0. However, to assess this, we have conducted a t-test for each result using equality of means as null hypothesis. When the t-test result is to reject the null hypothesis, the overhead is calculated as the difference between the real result and the estimation. Since the variances are unknown and unequal, the $t - statistic$ is calculated using the Welch t-test [20] as described in Equation (2.3). Subscripts 0 and 1 refer to the estimation and the actual values respectively.

$$\begin{aligned} t &= \frac{\mu_0 - \mu_1}{\sigma_{\mu_0 - \mu_1}}, \\ \sigma_{\mu_0 - \mu_1} &= \sqrt{\frac{\sigma_0^2}{n_0} + \frac{\sigma_1^2}{n_1}} \end{aligned} \tag{2.3}$$

The degrees of freedom of the distribution are calculated using Equation (2.4).

$$\nu = \frac{(\frac{\sigma_0^2}{n_0} + \frac{\sigma_1^2}{n_1})^2}{\frac{(\frac{\sigma_0^2}{n_0})^2}{n_0 - 1} + \frac{(\frac{\sigma_1^2}{n_1})^2}{n_1 - 1}}, \tag{2.4}$$

And the standard deviation of the difference between the real results and the estimations has been calculated using the Welch equations already explained (see Equation (2.5)).

$$\begin{aligned} \mu_{diff} &= \mu_0 - \mu_1 \\ \sigma_{estim} &= \sqrt{\frac{\sigma_0^2}{n_0} + \frac{\sigma_1^2}{n_1}} \\ n_0 &= n_1 = 5000 \end{aligned} \tag{2.5}$$

For every result on Sandy Bridge, the p-value allows us to reject the null hypothesis that $\mu_0 = \mu_1$ with more than 99% of confidence except for the I-S case when two threads are running on two different cores within the same processor. For Xeon Phi, we can reject the null hypothesis with a confidence around 93-99% in all cases. Thus, we will provide an estimation of the overhead for every scenario.

2.3 Sandy Bridge Results

Tables 2.1, 2.2 and 2.3 show the results of the ping-pong benchmark using two threads from the same core (Table 2.1), two different cores within the same processor (Table 2.2) and two cores from different processors (Table 2.3). For each configuration, the results show the estimated latency calculated applying Equation (2.1) to the results from Table 1.7. They also include the average and standard deviation of the empirical results, and the average and standard deviation of the difference between estimation and real value (using Welch's equations).

Table 2.1: Results in nanoseconds of the single-line ping-pong benchmark on Sandy Bridge using two threads from the same core.

		Estimation		Ping-Pong result		Difference	
$\mathcal{S}_s$	$\mathcal{S}_r$	avg	stdev	avg	stdev	avg	stdev
M	M	6.9	2.6	24.7	4.8	17.8	0.08
M	E	6.9	2.6	24.6	4.2	17.7	0.07
M	S	-	-	-	-	-	-
M	I	-	-	-	-	-	-
M	F	-	-	-	-	-	-
E	M	6.9	2.6	24.9	4.4	18.0	0.07
E	E	6.9	2.6	24.4	4.5	17.5	0.07
E	S	-	-	-	-	-	-
E	I	-	-	-	-	-	-
E	F	-	-	-	-	-	-
S	M	-	-	-	-	-	-
S	E	-	-	-	-	-	-
S	S	-	-	-	-	-	-
S	I	-	-	-	-	-	-
S	F	-	-	-	-	-	-
I	M	73.9	8.6	101.7	17.8	27.8	0.28
I	E	73.9	8.6	102.4	23.8	28.5	0.36
I	S	-	-	-	-	-	-
I	I	-	-	-	-	-	-
I	F	-	-	-	-	-	-
F	M	-	-	-	-	-	-
F	E	-	-	-	-	-	-
F	S	-	-	-	-	-	-
F	I	-	-	-	-	-	-
F	F	-	-	-	-	-	-

Regarding results on Table 2.1 (discarded results are explained at the beginning of the chapter), when both buffers are cached, the difference with the estimation is around 17.7 ns. When the sender line is invalidated, the overhead over the estimation is always around 28 ns, as shown in Equations (2.6) and (2.7).

$$\begin{aligned} &\mathcal{T}_1 = \frac{t2-t1}{2} = \frac{RTT}{2} \simeq R_{L,\mathcal{S}_s} + R_{L,\mathcal{S}_r} + R_{L,M} + O \\ &\mathcal{S}_s \in \{M, E\} \\ &\mathcal{S}_r \in \{M, E\} \\ &O \simeq 17.7 \end{aligned} \tag{2.6}$$

$$\begin{aligned} &\mathcal{T}_1 = \frac{t2-t1}{2} = \frac{RTT}{2} \simeq R_{L,I} + R_{L,\mathcal{S}_r} + R_{L,M} + O \\ &\mathcal{S}_s \in \{I\} \\ &\mathcal{S}_r \in \{M, E\} \\ &O \simeq 28 \end{aligned} \tag{2.7}$$

Table 2.2: Results in nanoseconds of the single-line ping-pong benchmark on Sandy Bridge using two threads from different cores within the same processor.

		Estimation		Ping-Pong result		Difference	
$\mathcal{S}_s$	$\mathcal{S}_r$	avg	stdev	avg	stdev	avg	stdev
M	M	85.4	5.1	80.1	8.6	-5.3	0.1
M	E	76.7	4.6	72.7	6.2	-3.9	0.1
M	S	59.6	4.2	73.8	7.2	14.1	0.1
M	I	-	-	-	-	-	-
M	F	59.0	3.7	74.1	14.1	15.1	0.2
E	M	85.4	5.1	78.4	7.5	-7.1	0.1
E	E	76.7	4.6	72.4	7.0	-4.2	0.1
E	S	59.6	4.2	73.1	10.5	13.4	0.2
E	I	-	-	-	-	-	-
E	F	59.0	3.7	72.0	13.1	12.9	0.2
S	M	85.9	5.2	80.9	7.5	-5.0	0.1
S	E	77.2	4.7	72.8	6.0	-4.4	0.1
S	S	60.1	4.3	74.8	8.6	14.7	0.1
S	I	-	-	-	-	-	-
S	F	59.5	3.8	74.7	16.5	15.2	0.2
I	M	152.4	8.7	126.2	12.9	-26.3	0.2
I	E	143.7	8.4	126.4	12.9	-17.2	0.2
I	S	126.6	8.1	126.4	13.6	-0.2	0.2
I	I	-	-	-	-	-	-
I	F	126.0	7.9	128.5	17.8	2.5	0.3
F	M	85.4	5.1	78.4	7.9	-7.0	0.1
F	E	76.7	4.6	73.0	9.9	-3.6	0.2
F	S	59.6	4.2	73.4	7.3	13.7	0.1
F	I	-	-	-	-	-	-
F	F	59.0	3.7	72.5	12.3	13.5	0.2

Table 2.2 shows the results of the single-line ping-pong benchmark using two threads running on different cores within the same processor, thus, cache coherency is maintained by the L3 cache. Except for the scenario in which the send-buffer is in memory, latencies when the recv-line is in M or E are actually lower than the estimation. However, the difference is lower than the standard deviation

of the measurements, thus, although overhead is statistically relevant, it is within the noise of real measurements.

Regarding S and F states, there is some overhead due to the need of invalidating the line on the other thread, thus having the same performance as the E scenarios, allowing us to clusterize this ping-pong results. Equations (2.8) and (2.9) show this considerations.

$$\begin{aligned} &\mathcal{T}_1 = \frac{t2-t1}{2} = \frac{RTT}{2} \simeq R_{L,\mathcal{S}_s} + R_{P,M} + R_{P,M} + O \\ &\mathcal{S}_s \in \{M, E, S, F\} \\ &\mathcal{S}_r \in \{M\} \\ &O \simeq -5.5 \end{aligned} \tag{2.8}$$

$$\begin{aligned} &\mathcal{T}_1 = \frac{t2-t1}{2} = \frac{RTT}{2} \simeq R_{L,\mathcal{S}_s} + R_{P,E} + R_{P,M} + O \\ &\mathcal{S}_s \in \{M, E, S, F\} \\ &\mathcal{S}_r \in \{E, S, F\} \\ &O \simeq -4 \end{aligned} \tag{2.9}$$

It is worth mentioning that having recv-lines in S or F causes huge variability among different executions of the benchmark, in fact, it was necessary to repeat the tests several times to select the execution with a "median" latency, since there were differences within an interval of 20 nanoseconds.

Regarding scenarios with the send-buffer in I, latency of the benchmark is always similar and slightly lower than the estimation. It could be explained because the core where the sender thread is running should be able to overlap the fetching of the invalidated line from memory, with the fetching of the recv-line. Thus, we can select the E case again with an overhead around -17 ns.

$$\begin{aligned} &\mathcal{T}_1 = \frac{t2-t1}{2} = \frac{RTT}{2} \simeq R_{L,I} + R_{P,E} + R_{P,M} + O \\ &\mathcal{S}_s \in \{I\} \\ &\mathcal{S}_r \in \{M, E, S, F\} \\ &O \simeq -17 \end{aligned} \tag{2.10}$$

Table 2.3 shows the ping-pong results using two threads running on different processors connected by QPI. This experiments suffered also from the variability of the previous ones, not only for F state but for almost every line state.

In this case, except for the scenario when $\mathcal{S}_r = I$, the differences over the estimated time are almost constant for each recv-buffer state, thus we can derive equations (2.11), (2.12), (2.13) and (2.14). If the recv-buffer is in M or E state, the estimated latency is higher than the one obtained. The most plausible reason seems to be prefetching. Lines are transferred via QPI, and, T_1, while waiting for the modified line to receive it, could start requesting the recv-buffer from T_0 since it will have to write to it later, thus, reducing the total amount of time. Special consideration should be given to S and F. The latency predicted is lower than the empirical one since the estimation does not take into account the snooping for invalidation across QPI. In fact, the latencies of having a recv-buffer in S or F state are higher than for M and E. This suggest that the snooping for invalidation of a shared line has a large cost that stalls the core. There also seems to be a difference between having the line in S or in F state. Thus, although BenchIT results indicated that there is a local F in each processor that can serve *read* requests, this suggest that there is only a global F when it comes to invalidate lines.

$$\begin{aligned} &\mathcal{T}_1 = \frac{t2-t1}{2} = \frac{RTT}{2} \simeq R_{L,\mathcal{S}_s} + R_{N,M} + R_{N,M} + O \\ &\mathcal{S}_s \in \{M, E, S, F\} \\ &\mathcal{S}_r \in \{M\} \\ &O \simeq -81.7 \end{aligned} \tag{2.11}$$

Table 2.3: Results in nanoseconds of the single-line ping-pong benchmark on Sandy Bridge using two threads running in different processors connected by QPI.

		Estimation		Ping-Pong result		Difference	
$\mathcal{S}_s$	$\mathcal{S}_r$	avg	stdev	avg	stdev	avg	stdev
M	M	233.1	7.3	151.5	15.1	-81.6	0.2
M	E	193.5	6.4	149.2	13.0	-44.3	0.2
M	S	135.1	5.8	238.5	13.3	103.4	0.2
M	I	-	-	-	-	-	-
M	F	133.7	5.8	178.5	25.3	44.7	0.4
E	M	233.1	7.3	151.5	15.4	-81.5	0.2
E	E	193.4	6.4	149.2	12.5	-44.2	0.2
E	S	135.0	5.8	237.7	15.1	102.6	0.2
E	I	-	-	-	-	-	-
E	F	133.7	5.8	178.5	23.2	44.7	0.3
S	M	233.6	7.3	151.5	16.3	-82.1	0.3
S	E	194.0	6.4	150.8	18.0	-43.2	0.3
S	S	135.5	5.9	239.2	16.6	103.7	0.2
S	I	-	-	-	-	-	-
S	F	134.2	5.8	178.5	21.9	44.2	0.3
I	M	300.1	10.1	201.5	16.3	-98.5	0.3
I	E	260.4	9.4	200.8	15.1	-59.7	0.3
I	S	202.0	9.1	241.5	14.9	39.5	0.2
I	I	-	-	-	-	-	-
I	F	200.7	9.0	213.0	16.4	12.3	0.3
F	M	233.1	7.3	150.7	26.2	-82.3	0.4
F	E	193.4	6.4	150.7	20.6	-42.7	0.3
F	S	135.0	5.8	237.7	15.0	102.6	0.2
F	I	-	-	-	-	-	-
F	F	133.7	5.8	178.5	23.0	44.7	0.3

$$\begin{aligned} &\mathcal{T}_1 = \frac{t2-t1}{2} = \frac{RTT}{2} \simeq R_{L,\mathcal{S}_s} + R_{N,E} + R_{N,M} + O \\ &\mathcal{S}_s \in \{M, E, S, F\} \\ &\mathcal{S}_r \in \{E\} \\ &O \simeq -43.5 \end{aligned} \tag{2.12}$$

$$\begin{aligned} &\mathcal{T}_1 = \frac{t2-t1}{2} = \frac{RTT}{2} \simeq R_{L,\mathcal{S}_s} + R_{N,S} + R_{N,M} + O \\ &\mathcal{S}_s \in \{M, E, S, F\} \\ &\mathcal{S}_r \in \{S\} \\ &O \simeq 103 \end{aligned} \tag{2.13}$$

$$\begin{aligned} &\mathcal{T}_1 = \frac{t2-t1}{2} = \frac{RTT}{2} \simeq R_{L,\mathcal{S}_s} + R_{N,F} + R_{N,M} + O \\ &\mathcal{S}_s \in \{M, E, S, F\} \\ &\mathcal{S}_r \in \{F\} \\ &O \simeq 44.5 \end{aligned} \tag{2.14}$$

When the send-line is invalidated we have the four scenarios in Equation (2.15).

$$
\begin{aligned}
\mathcal{T}_1 &= \frac{t2-t1}{2} = \frac{RTT}{2} \simeq R_{L,I} + R_{N,\mathcal{S}_r} + R_{N,M} + O_{\mathcal{S}_r} \\
\mathcal{S}_s &\in \{I\} \\
\mathcal{S}_r &\in \{M, E, S, F\} \\
O_M &\simeq -98.5 \;\; O_E \simeq -59.7 \;\; O_S \simeq 39.5 \;\; O_I \simeq 12.31
\end{aligned}
\tag{2.15}
$$

2.4 Intel Xeon Phi Results

Tables 2.4, 2.5, 2.6 and 2.7 include the results of the pingpong benchmark run using two threads from the same core (Table 2.4), two threads running on adjacent cores (Table 2.5), two threads running on middle-distant cores (Table 2.6) and two threads running on cores separated by the maximum distance (Table 2.7). For each configuration, the results show the estimated latency using Equation 2.1 and the results from Table 1.6 using the average and the standard deviation of the estimation calculated as explained before. They also include the average and standard deviation of the results obtained with the ping-pong benchmark, and the difference between the estimation and the real value.

Table 2.4: Results in nanoseconds of the ping-pong test on Intel Xeon Phi for two threads running on the same core. S and R in the header stand for Send and Recv buffers.

		Estimation		PingPong result		Difference	
$\mathcal{S}_s$	$\mathcal{S}_r$	avg	stdev	avg	stdev	avg	stdev
M	M	25.8	0.4	78.8	5.5	53.0	0.1
M	E	25.8	0.4	73.7	6.8	47.9	0.1
M	S	-	-	-	-	-	-
M	I	-	-	-	-	-	-
E	M	25.8	0.4	76.4	6.9	50.6	0.1
E	E	25.8	0.4	73.8	6.6	48.0	0.1
E	S	-	-	-	-	-	-
E	I	-	-	-	-	-	-
S	M	-	-	-	-	-	-
S	E	-	-	-	-	-	-
S	S	-	-	-	-	-	-
S	I	-	-	-	-	-	-
I	M	294.9	34.0	429.5	57.5	134.6	0.9
I	E	294.9	34.0	425.6	59.9	130.7	0.9
I	S	-	-	-	-	-	-
I	I	-	-	-	-	-	-

Regarding results on Table 2.4 (discarded results are explained at the beginning of the chapter), when the send-buffer is cached, if the recv-buffer is in M, the difference with the estimation is around 50 ns, and 46ns if it is in E (equations (2.16), (2.17)).When the send-buffer is in memory, the overhead over the estimation is always around 130 ns (Equation (2.18)). In the equations we use the clusterized notation for Intel MIC derived from Table 1.5b.

$$
\begin{aligned}
\mathcal{T}_1 = \frac{t2-t1}{2} = \frac{RTT}{2} &\simeq R_L + R_L + R_L + O \\
\mathcal{S}_s &\in \{M, E\} \\
\mathcal{S}_r &\in \{M\} \\
O &\simeq 50.
\end{aligned}
\tag{2.16}
$$

$$\begin{aligned} \mathcal{T}_1 = \frac{t2-t1}{2} = \frac{RTT}{2} \simeq R_L + R_L + R_L + O \\ \mathcal{S}_s \in \{M, E\} \\ \mathcal{S}_r \in \{E\} \\ O \simeq 46. \end{aligned} \tag{2.17}$$

$$\begin{aligned} \mathcal{T}_1 = \frac{t2-t1}{2} = \frac{RTT}{2} \simeq R_I + R_L + R_L + O \\ \mathcal{S}_s \in \{I\} \\ \mathcal{S}_r \in \{M, E\} \\ O \simeq 130 \end{aligned} \tag{2.18}$$

Table 2.5: Results in nanoseconds of the ping-pong benchmark on Intel Xeon Phi for two threads running on adjacent cores. S and R in the header stand for Send and Recv buffers.

		Estimation		Ping-Pong result		Difference	
$\mathcal{S}_s$	$\mathcal{S}_r$	avg	stdev	avg	stdev	avg	stdev
M	M	491.0	30.7	506.2	89.5	15.2	1.3
M	E	477.2	30.0	505.0	82.5	27.7	1.2
M	S	481.8	24.0	526.8	72.9	45.0	1.1
M	I	-	-	-	-	-	-
E	M	491.0	30.7	505.3	102.4	14.3	1.5
E	E	477.2	30.0	504.7	73.7	27.4	1.1
E	S	481.8	24.0	529.0	48.2	47.2	0.8
E	I	-	-	-	-	-	-
S	M	491.1	30.7	506.3	97.5	15.2	1.4
S	E	477.3	30.0	505.5	108.7	28.2	1.6
S	S	481.9	24.0	529.1	96.2	47.2	1.4
S	I	-	-	-	-	-	-
I	M	760.1	45.9	853.7	81.3	93.6	1.3
I	E	746.3	45.3	850.4	116.3	104.1	1.8
I	S	750.9	41.6	876.2	112.8	125.4	1.7
I	I	-	-	-	-	-	-

The rest of tables (Tables 2.5, 2.6 and 2.7) represent three possible scenarios of threads communicating from different cores of the ring bus. There is no observance of differences due to the distance between cores and the difference between the estimation and the real value is always less than the standard deviation (except for some cases of invalid lines), thus, as it happened for Sandy Bridge, the empirical results are within the noise of the measurements. As an example, Equations (2.19) and (2.20) show the overhead in terms ot the recv-buffer state for middle-distant cores.

$$\begin{aligned} \mathcal{T}_1 = \frac{t2-t1}{2} = \frac{RTT}{2} \simeq R_L + R_R + R_R + O \\ \mathcal{S}_s \in \{M, E, S\} \\ \mathcal{S}_r \in \{M, E, S\} \\ O_M \simeq 20 \;\; O_E \simeq 20 O_S \simeq 46.5 \end{aligned} \tag{2.19}$$

$$\begin{aligned} \mathcal{T}_1 = \frac{t2-t1}{2} = \frac{RTT}{2} \simeq R_L + R_R + R_R + O \\ \mathcal{S}_s \in \{I\} \\ \mathcal{S}_r \in \{M, E, S\} \\ O_M \simeq 97.4 \;\; O_E \simeq 94.7 \;\; O_S \simeq 115.1 \end{aligned} \tag{2.20}$$

Table 2.6: Results in nanoseconds of the ping-pong benchmark on Intel Xeon Phi for two threads running on middle-distant cores. S and R in the header stand for Send and Recv buffers.

		Estimation		Ping-Pong result		Difference	
$\mathcal{S}_s$	$\mathcal{S}_r$	avg	stdev	avg	stdev	avg	stdev
M	M	478.0	36.2	501.0	74.2	22.9	1.167
M	E	479.1	36.1	500.0	78.8	20.5	1.226
M	S	476.7	43.3	523.0	82.3	46.3	1.316
M	I	-	-	-	-	-	-
E	M	478.0	36.2	500.0	78.5	21.9	1.223
E	E	479.1	36.1	497.1	77.2	18.0	1.205
E	S	476.7	43.3	522.8	54.4	46.2	0.983
E	I	-	-	-	-	-	-
S	M	478.1	36.2	497.6	94.3	19.5	1.429
S	E	479.2	36.1	500.2	86.7	21.0	1.328
S	S	476.7	43.3	524.0	65.1	47.2	1.106
S	I	-	-	-	-	-	-
I	M	747.1	49.7	844.4	106.5	97.4	1.662
I	E	748.1	49.6	842.8	102.0	94.7	1.604
I	S	745.7	55.1	860.8	97.7	115.1	1.586
I	I	-	-	-	-	-	-

Table 2.7: Results in nanoseconds of the ping-pong benchmark on Intel Xeon Phi for two threads running on maximum-distant cores. S and R in the header stand for Send and Recv buffers.

		Estimation		PingPong result		Difference	
$\mathcal{S}_s$	$\mathcal{S}_r$	median	stdev	avg	stdev	avg	stdev
M	M	488.8	14.7	506.1	23.9	17.3	0.4
M	E	486.1	29.6	504.5	31.8	18.4	0.6
M	S	482.1	24.8	522.0	50.0	39.9	0.8
M	I	-	-	-	-	-	-
E	M	488.8	14.7	497.8	28.9	9.0	0.5
E	E	486.1	29.6	503.6	25.0	17.6	0.5
E	S	482.1	24.8	522.7	53.4	40.6	0.8
E	I	-	-	-	-	-	-
S	M	488.9	14.7	505.3	23.9	16.5	0.4
S	E	486.1	29.6	494.2	23.7	8.0	0.5
S	S	482.1	24.8	522.7	45.9	40.5	0.7
S	I	-	-	-	-	-	-
I	M	757.8	37.0	834.0	68.8	76.1	1.1
I	E	755.1	45.1	840.4	77.6	85.3	1.3
I	S	751.1	42.1	865.8	78.2	114.7	1.3
I	I	-	-	-	-	-	-

Chapter 3

Communication Model for Xeon Phi

Once we had analyzed the modeling of single transfers in both processors, we focused on the most scalable architecture, Intel Xeon Phi, to develop a whole communication model that we will used to design algorithms of data exchange. Before that, we made a preliminary study of bandwidth on both Sandy Bridge and Xeon Phi, that is included in Appendix A. However, once that we have seen that modeling communication based on cache line transfers is possible in both architectures, and given the differences in the clusterized models developed and in bandwidth performance, that reflect the differences in the architectures, we decided to focus on one of them to provide a thorough and comprehensive analysis and modeling.

In this chapter, we will show how we studied the effect of having several threads accessing to the same data and the multi-line ping-pong modeling. From now on, we will use the clusterized Xeon Phi cache model to reduce the number of scenarios to analyze.

3.1 Multi-line Ping-Pong Model

In this section, we show how to model multi-line ping-pong transfers, having more than one cache line per buffer. Assuming `x86` total store order [16], the receiver will only poll for the canary value on the last line of the recv-buffer while the sender copies the content of the send-buffer. To be able to study the effect of different cache states, the buffer size has to be limited to 8 kb due to the use of four buffers per pair of threads and the L1 cache size (32 kb).

Intuitively, and assuming pipelining, the sender should fetch the buffers in $\frac{2N}{P}R_{\mathcal{L},\mathcal{S}}$ where N is the number of cache lines of each buffer and P is the number of outstanding memory requests per core. After the copy, the receiver reads the last line, that has been modified by the sender, in $R_{R,M}$. However, this simple model misses several factors that affect performance as the eviction overhead, the hardware prefetcher, the signal buses or the DTD capabilities to serve the outstanding requests. To approach this overhead, we tested a multiplicative factor based on the results, but, although it was asymptotically accurate, the relative error reached the 40-50% for small messages (2-12 lines) when the send-buffer was in I, and around 30% when it was in E.

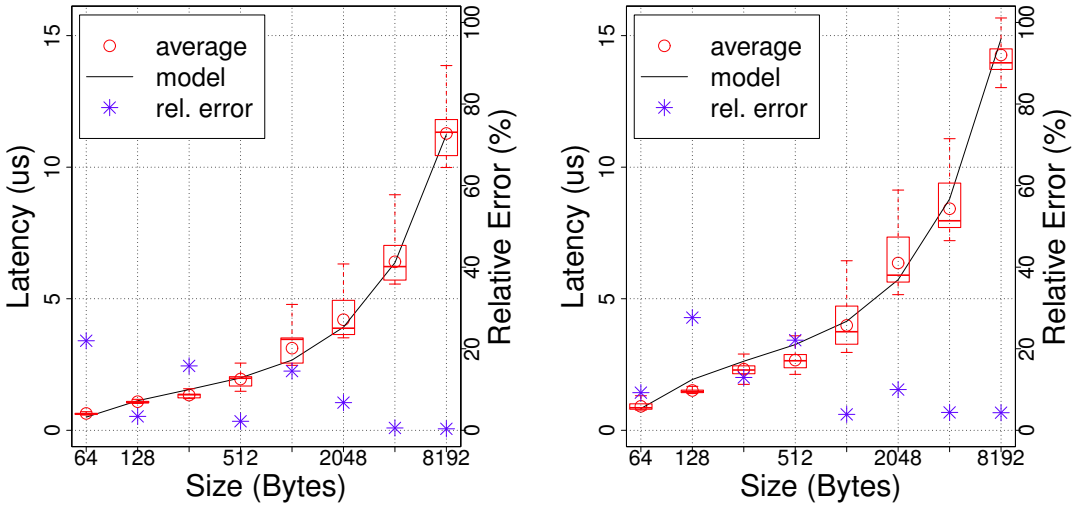
To obtain a more accurate model, we use linear regression with a typical transfer function. Equation (3.1) shows the model function where o is the asymptotic fetch latency for each cache line (including hardware prefetch, etc.), and p, q model the startup overhead which consists of a fixed part q that is amortized partially by the number of fetched lines p.

$$\mathcal{T}_N = o \cdot N + q - \frac{p}{N} \tag{3.1}$$

If we apply this model to a single line broadcast, it will essentially lead to the one-line model discussed before in Section 2.4 $\mathcal{T}_1 = R_L + 2R_R + O = q + o - p$.

The parametrization of the model has been performed with ping-pong tests using buffers from 64 bytes (one cache line) to 8 kb, varying the initial cache state of the buffers[1].

[1]This test can use the $\mathcal{S}_r = I$ because the receiver is polling only the last line, and when the sender fetches the recv-buffer lines they are all invalidated except for the last one.



(a) Sender and receiver buffers in Exclusive state. The parameters of the model (in nanoseconds) are $76.0 \cdot N + 1521.0 - \frac{1096.0}{N}$

(b) Sender and receiver buffers in Invalid state. The parameters of the model (in nanoseconds) are $94.9 \cdot N + 2750.0 - \frac{2017.5}{N}$.

Figure 3.1: Latency and performance model for a multi-line ping-pong

Table 3.1: Parametrization (in nanoseconds) of the multi-line Ping-Pong model

$\mathcal{S}_s$	$\mathcal{S}_r$	q	o	p
E	E	1521.0	76.0	1096.0
E	I	1778.4	73.2	1276.9
I	E	2698.5	94.4	1868.5
I	E	2750.0	94.9	2017.5

The results of our ping-pong measurements and the model fits are shown in Figure 3.1. The measurement for each size were repeated 5000 times and timed separately using `x86 RDTSC`. The left axis shows boxplots [14] of each value where the horizontal line is the median, the upper and lower parts of the box denote the first and third quartile and the whiskers show the minimum and maximum data values (outliers were removed). We use boxplots to visualize the statistical noise across measurements. The right axis and asterisks show the relative error of the model.

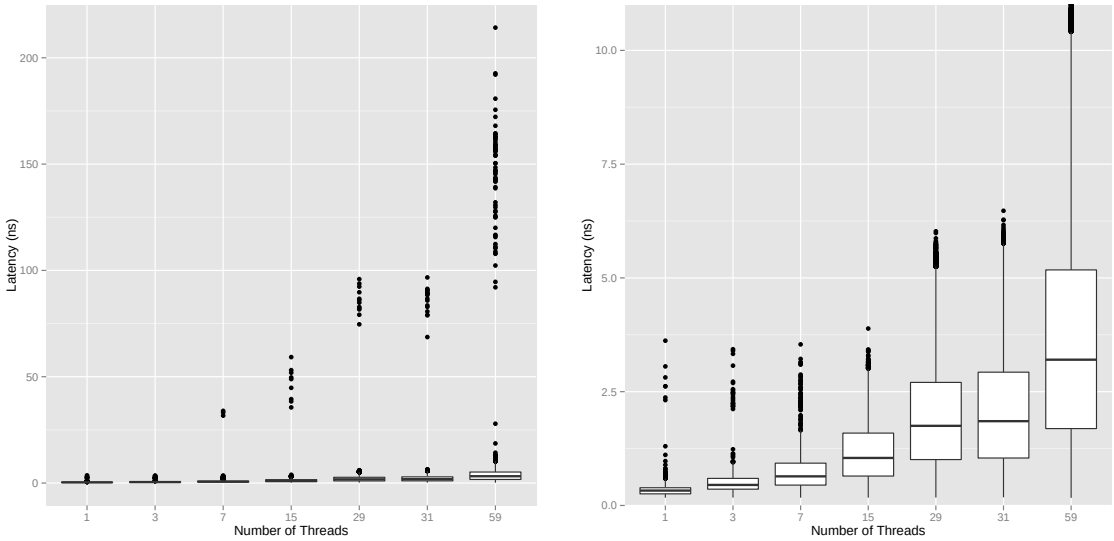
Table 3.1 summarizes the parameters obtained applying the regression model to each of the combinations of cache states.

3.2 Contention Analysis

It was observed from the bandwidth benchmarks hat there is no congestion when accessing independent buffers from memory (see Appendix A). However, this changes when several threads are accessing the same data, then causing contention in the access to the DTDs [4]. To assess the behaviour of the memory system when the same data is accessed, we designed a benchmark where all threads copy one line from a global send-buffer into a private recv-buffer. Since the owner of the global send-buffer is completely idle, we will talk about number of threads taking into account only the number of receivers.

To ensure that all threads are requesting the line simultaneously, we have used a synchronization mechanism based on the TimeStamp Counter (`RDTSC`) given that is consistent among cores [10, §2.1.7] in a normal power state of the Xeon Phi. Thus, an array of time intervals of time is created before launching the threads, then every iteration starts when the `RDTSC` reaches the desired value.

The benchmark uses 5000 iterations and the addresses of the common send-buffer and the private recv-buffers are randomized to avoid the influence of the DTDs. Previous experiments showed that using the same address in each iteration for the private recv-buffer, and/or the common send buffer, can introduce bias in the results caused by the DTDs accessed.

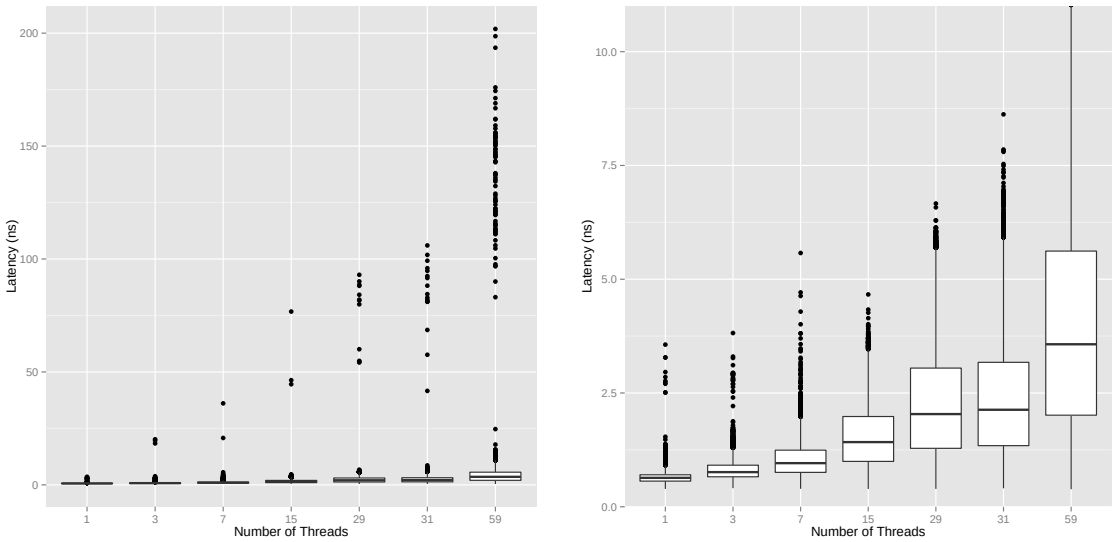


(a) EE-boxplot

(b) EE-boxplot without outliers

Figure 3.2: Boxplot of the contention analysis for the E-E scenario

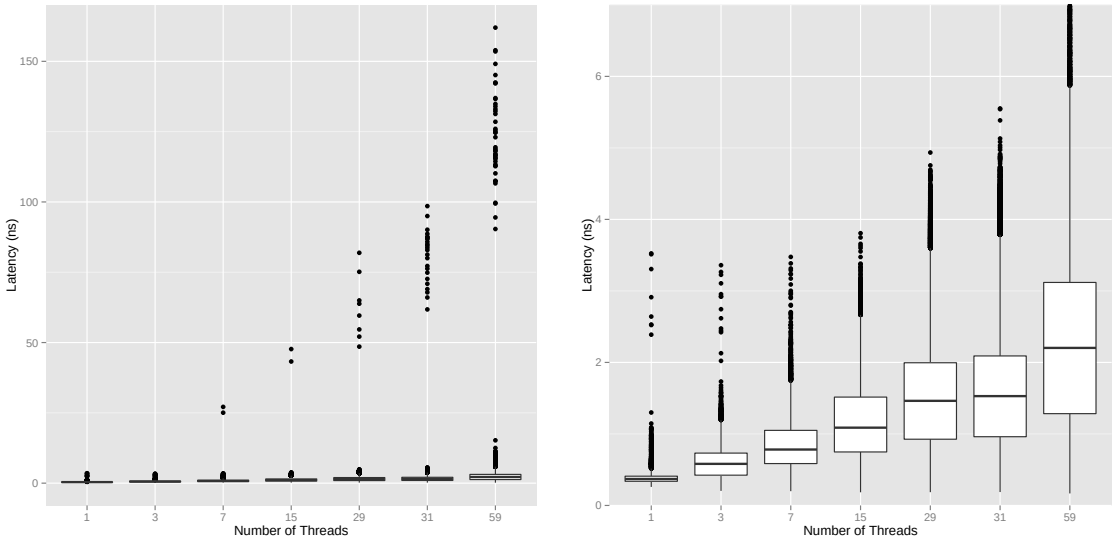
The results have been obtained whith each of the lines in two different states: E and I. Figures 3.2 to 3.5 show the boxplots of the results obtained. The left graph represents all the values obtained from the experiments, showing that when the number of threads increases, the variability in the observations is large, which could be caused by interferences of the targeted DTDs in the different iterations or by the OS (using 60 threads we can not avoid the core that is running the OS). The right graphs show the same boxplots but focusing on the median values and leaving part of the outliers beyond the axis. In every scenario, the increase of threads requesting the same lines causes contention due to the interaction with the same DTD.



(a) EI-boxplot

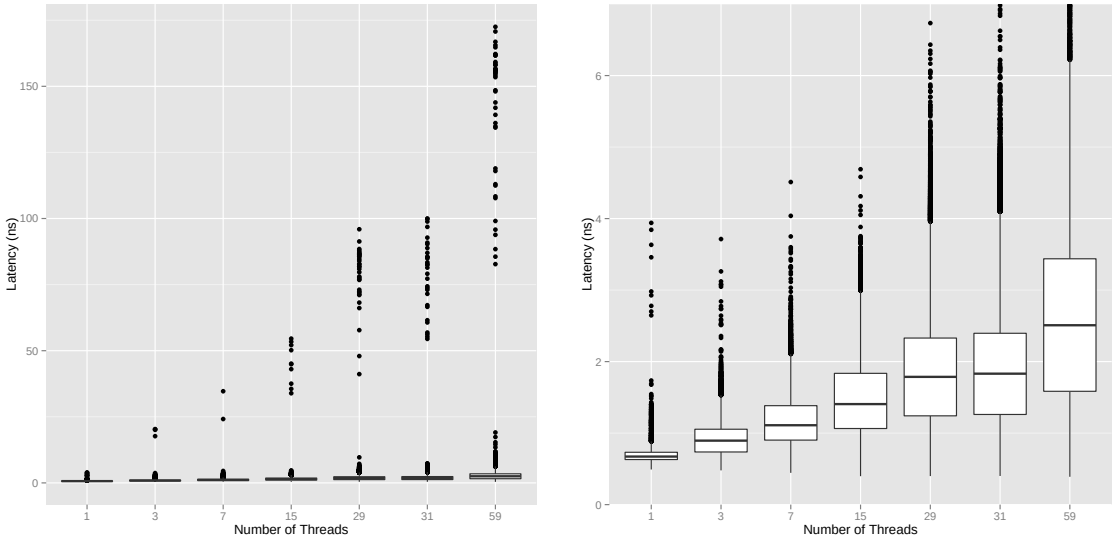
(b) EI-boxplot without outliers

Figure 3.3: Boxplot of the contention analysis for the E-I scenario



(a) II-boxplot (b) EI-boxplot without outliers

Figure 3.4: Boxplot of the contention analysis for the I-E scenario



(a) II-boxplot (b) EI-boxplot without outliers

Figure 3.5: Boxplot of the contention analysis for the I-I scenario

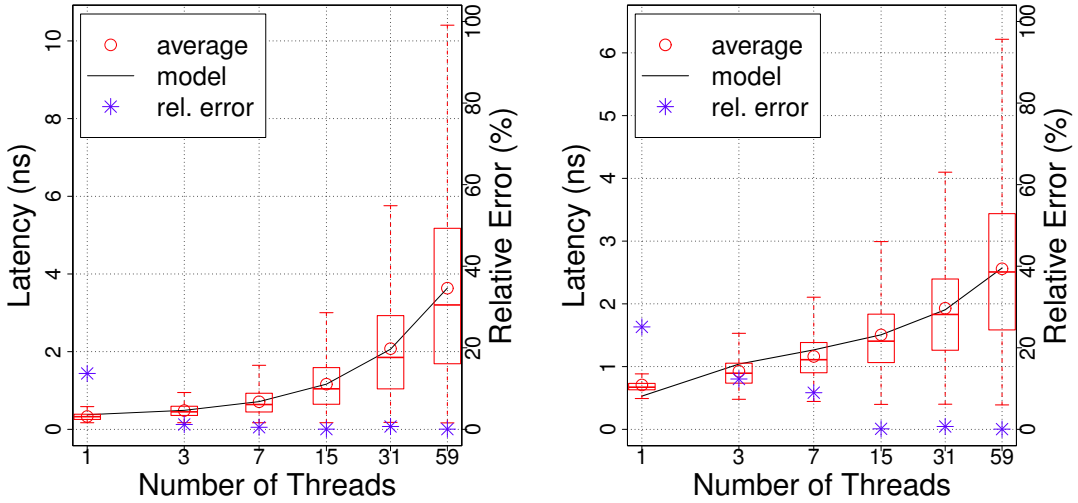
3.2.1 Statistical Analysis of the Results

Contention for cached lines can be estimated with a linear model $\mathcal{T}_C(n_{th}) = c \cdot n_{th} + b$, where n_{th} is the number of threads, and c represents the slope and the overhead imposed when adding a new thread. If $n_{th} = 1$, there is no contention and $\mathcal{T}_C(1) = R_L + R_R = c + b$ (the cost of copying a global send-line into a private recv-line). Equation (3.2) shows the DTD contention model when buffers are in E state in the owner's cache.

$$\mathcal{T}_C(n_{th}) = R_L + R_R + c \cdot (n_{th} - 1) = b + c \cdot n_{th} \tag{3.2}$$

However, if the global line is in memory, the performance is limited by the access to memory and the model is similar to the one developed for the multi-line ping-pong in terms of the number of threads accessing the line instead of the message size.

$$\mathcal{T}_C(n_{th}) = c \cdot n_{th} + b - \frac{a}{n_{th}} \tag{3.3}$$



(a) Sender and receiver buffers in Exclusive state. The parameters of the model (in nanoseconds) are $320.5 + 56.2 \cdot n_{th}$.

(b) Sender and receiver buffers in Invalid state. The parameters of the model (in nanoseconds) are $23.4 \cdot n_{th} + 1202.0 - \frac{695.8}{n_{th}}$.

Figure 3.6: Contention in the access to the same line

Table 3.2: Parametrization (in nanoseconds) of the contention model

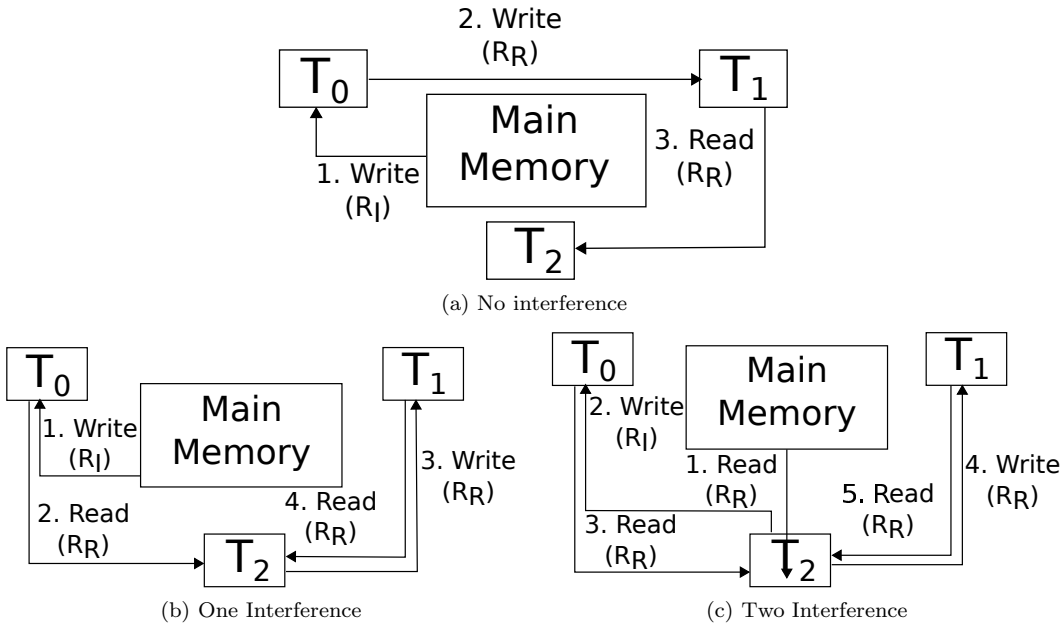
$\mathcal{S}_s$	$\mathcal{S}_r$	b	c	a
E	E	320.5	56.2	-
E	I	604.4	57.6	-
I	E	863.6	23.9	667.4
I	E	1202.0	23.4	695.8

Figure 3.6 shows the results of the benchmark for different number of threads and state of the global and private buffers (E for both in Figure 3.6a and I in Figure 3.6b). The parameters of the model are included in Table 3.2.

Chapter 4

Designing Collective Algorithms on Xeon Phi

The developed model allows us to design optimized communication algorithms. However, the use of several interacting threads, inevitably causes huge variability and different overheads. As an example, let us assume that we have a line that is invalid in every cache, thus located in memory, and that two threads, T_0 and T_1, have to write to it. Moreover, another thread, $T2$, is waiting for them to write the value, thus, T_2 is going to poll the line to check if T_0 and T_1 have already written. In this scenario, the performance will depend on the order in which this three threads reach the operation, as shown in Figure 4.1.



(a) No interference

(b) One Interference

(c) Two Interference

Figure 4.1: Interference in the access to a cache line by two writers (T_0 and T_1) and a reader that is polling the line waiting for writes.

If T_0 and T_1 write to the line, and then T_2 checks it (Figure 4.1a), the cost will be $R_I + 2R_R$. But, if T_2 checks it right after the write of T_0 (Figure 4.1b), the line makes an extra travel to T_2 before going to T_1, and T_2 has to read it again to get the expected value ($R_I + 3R_R$). And there is still a worse scenario: when T_2 is the first that gets the line (Figure 4.1c) and keeps polling, causing the line to travel from T_2's cache to each writer and the other way round, increasing the cost up to $R_I + 4R_R$.

Since algorithms are going to suffer from this variability, we have expressed them in terms of *Min-Max Models* including the best and worst performance case. To construct these models, we have first

analyzed which data transfers are performed in each algorithm and the possible variations, and then, represent them in terms of the model developed in previous sections using equations as puzzle pieces.

In the rest of the section, we assumed that data buffers are initially in exclusive state in the owner's cache to simplify the discussion. Similar results were obtained with buffers in invalid state and the models can be adapted by applying the invalid-state equations where the exclusive-state models are used. For the same reason, shared structures are assumed to be in memory (invalid) at the beginning of each algorithm.

4.1 Fast Message Broadcasting

The broadcast operation consists of sending one message from one thread, called *root*, to every other thread. We will talk in terms of trees since they are the most common communication patterns in broadcast algorithms.

In a shared memory scenario, a send-receive pair of operations can be performed in two different ways. In a *sender-driven* approach, the sender copies the data into the recv-buffer, similar to the ping-pong benchmark; the receiver may notify the sender with the canary protocol that the recv-buffer is ready. In a *receiver-driven* approach, the receiver would copy the message after the sender has notified that it is ready (notification forwards). In addition, the receiver has to acknowledge the reception of the message (notification backwards).

For the broadcast operation, where the sender communicates with several receivers, the receiver-driven approach allows simultaneous copies and thus leads to better load balancing for larger numbers of threads despite the additional acknowledgment. To model the acknowledgement and notification of readiness we will decompose the broadcast operation into three stages: notification forwards (the root or parent notifies that the buffer is ready to be copied), transfer of data, and notification backwards (the children notify that they have performed the copy).

4.1.1 Notification

The notification forwards and backwards uses shared structures in order for them to be accessible to every thread. There, the root can notify that the message is ready to be copied and the rest of threads can confirm that they have received the message, so that the root can free the shared structure. If the algorithm uses a tree, each parent has to communicate with its descendants and every descendant has to notify backwards to is parent, thus, several notification substructures will be needed.

Given that the parent has to provide, along with a notification flag, the data that is going to be copied or the address where it is stored, the notification forwards can be seen as a notification with payload where data and flag can be fetched in a single line. Hence, if data is small enough to fit in the same line, the descendants will poll the notification line and they will copy the data directly from there. If the space in the notification line is not enough, the parent will set the flag and an address (zero-copy protocol) from which descendants will copy the data.

The notification backwards from the descendants to the parent uses cache lines that are independent from the notification forwards structures to avoid interference with the copy of the data. We analyzed two variants of this notification: the first one with one cache line in which every thread adds a value after finishing. The parent reads this value and checks if the operation is done. This requires every thread, but the parent, to write to the same line, and, since only one thread can write a line at a time, these writes are going to be serialized. In the second variant, each thread avoids serialization by writing its own notification line, but the parent has to read them all to check if the operation is done. We will focus on the use of one line because both the model and the empirical results confirmed that it provides better performance.

The model for the notification backwards assumes that each thread writes an immediate value and thus there is no cost associated with reading an additional cache line.

Since all threads, but the root, write to the same line, every thread (but the root) has to read and modify the notification line ($R_I + (n_{th} - 2)R_R$). Then, in the best case (min), the root only reads the line at the end (R_R).

$$\mathcal{T}_{nb,min}(n_{th}) = R_I + (n_{th} - 1) \cdot R_R \tag{4.1}$$

In the worst case (max), the root will check the notification line after each time a thread wrote to it. This means that the root makes a first tentative reading the line from memory (R_I), then, the first thread to notify will fetch the line from the root's cache (R_R) and, after that, the root will make one R_R to check the value. This scheme will be repeated for each writing thread, as shown in Equation (4.2).

$$\mathcal{T}_{nb,max}(n_{th}) = R_I + 2(n_{th} - 1) \cdot R_R \tag{4.2}$$

4.1.2 Small Broadcast

Now that the notification has been modeled, we have to the design how the data is transferred to every descendant. Karp et al. developed an optimal algorithm [12] in the LogP model, but this is not applicable in our state-based min-max model with separate notification. However, we can use a similar technique to design our optimal tree taking into account that all the descendants of a given node can get the data at the same time.

First of all, we will describe the structure of a generic tree assuming that each level i can use a different number of descendants (k_i) and that the height of the tree is d. In this structure, the number of processes in each level (n_i) of the tree is given by equation 4.3.

$$n_0 = 1,\ n_i \leq \prod_{j=1}^{i} k_j \tag{4.3}$$

Hence, the total number of threads can be expressed as:

$$n_{th} \leq 1 + \sum_{i=1}^{d} \prod_{j=1}^{i} k_j \tag{4.4}$$

All of the k_i descendants of one thread from level i are accessing to the same line, thus, by increasing the number of descendants, we also increase contention, and every one of the descendants would be able to get the data in $\mathcal{T}_C(k_i)$. It is also worth mentioning that different threads accessing different data should not cause any congestion, thus, it is possible to apply the contention model to each group of descendants ignoring other groups of threads. Taking this into account, he latency of copying a message throughout this tree is:

$$\begin{aligned} \mathcal{T}_{tree} &= \sum_{i=1}^{d} \mathcal{T}_C(k_i) = \sum_{i=1}^{d} (c \cdot k_i + b) \\ &= \sum_{i=1}^{d} (R_R + R_L + c \cdot (k_i - 1)) \\ &= d \cdot b + \sum_{i=1}^{d} (c \cdot k_i) \end{aligned} \tag{4.5}$$

The optimized tree has to find a tradeoff between the number of threads that get the value at the same time, thus causing congestion ($c \cdot k_i$), and the number of levels of the tree, which increases the term $d \cdot b$. It is expected for the values of k_i to decrease while descending throughout the tree since the lower the value of k_i, the lower the latency of acquiring one message.

Figure 4.2 presents an example of a 10-threads broadcast tree using (arbitrarily chosen) $d = 2$, $k_1 = 3$, $k_2 = 2$. The backwards arrows indicate from which node the receivers copy the message. Threads from level 1 get the message in $t_1 = c \cdot k_1 + b = 3c + b$ and, then, leaf nodes will copy it in $c \cdot k_2 + b = 2c + b$, thus, the total time will be $t_2 = c \cdot (k_1 + k_2) + 2b = 5c + 2b$.

Now that we have the tree structure defined, we have to add the notification. The total time for notification backwards is the time spent from the moment in which the last descendant receives the message until the root is aware that every thread has it. The correspondent equations from Section 4.1.1 must be applied to each level of the tree, providing the cost of notification backwards in the critical path.

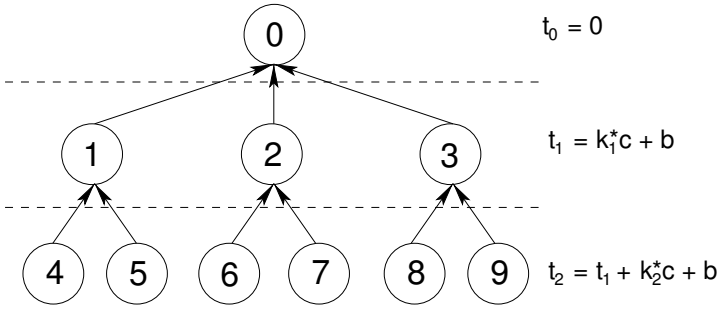


Figure 4.2: Tree for an 10-threads broadcast assuming d = 2, $k_1 = 3$, $k_2 = 2$.

Regarding notification forwards (i.e., the parent notifying to its descendants that the buffer is ready to be copied), first, there is a global flag where the root sets the shared structure as occupied by the current operation (R_I). Each descendant has to check the flag and copy the data (that are on the same line), which can be estimated by the contention model (Equation (4.5)), and then, each parent has to read its own structure (R_I), copy the data into this structure (R_L) and set it as ready (R_L). In the worst case, the descendants can read the flag before it is set and interfere while the parent is copying the data and setting the flag. Moreover, when interference involves several threads, they will cause contention. Although the first reading affected by contention is an invalid line, the contention model for a cached line is used for simplicity purposes. Equation (4.6) show the best (min) and worst (max) case model for this notification forwards.

$$\begin{aligned} \mathcal{T}_{fw,min} &= R_I + \sum_{i=1}^{d} (R_I + 2R_L) = (d+1)R_I + 2dR_L \\ \mathcal{T}_{fw,max} &= R_I + \sum_{i=1}^{d} 2\left(R_R + \sum_{i=1}^{d} (c \cdot k_i + b)\right) \end{aligned} \tag{4.6}$$

The optimal tree to perform a one-item broadcast is thus the solution to the minimization problem expressed in equation (4.7), combining notifications and reception of data.

$$\begin{aligned} \mathcal{T}_{sbcast} = \min_{d,k_i} &\left(\mathcal{T}_{fw} + \sum_{i=1}^{d} (c \cdot k_i + b) + \sum_{i=1}^{d} \mathcal{T}_{nb}(k_i + 1)\right) \\ & N \leq 1 + \sum_{i=1}^{d} \prod_{j=1}^{i} k_j, \quad \forall i < j, k_i \leq k_j \end{aligned} \tag{4.7}$$

This equation can be solved with numerical methods to obtain d and all k_i for the optimal broadcast tree.

4.1.3 Large Broadcast

If we use the tree developed for the small broadcast when each thread has to copy N lines, assuming that the N lines are sent in N_{pack} packets of size N_{cl}, the leafs will not start copying until the first package has arrived.

In order to avoid having idle threads in the first stages, and given that it is possible to divide the N-line message in $N_{pack} = n_{th} - 1$ slices, we can construct an algorithm in stages in which every thread, but the root, starts copying one different slice of the message. Having every line of the message in the root's cache could cause some contention (the root has to communicate with the DTDs to change the state of each line from E or M to S) and for the next stages, each thread copy one slice of the message from a different thread, having only one thread copying from the same location at the same time. Given that the root is thread 0, we can construct a cyclic algorithm in which each thread i copies the slice $i-1$ from the root in the stage 0, and, in stage j, thread i copies the slice $((i-1)+j) \mod (n_{th}-1)$ from

the thread that copied this slice from the root ($slice + 1$). The performance model of this pipelined algorithm is stated in Equation (4.8).

$$\begin{aligned}
\mathcal{T}_{pipedbcast} &= \mathcal{T}_{init} + \mathcal{T}_{1^{st}} + \mathcal{T}_{rest} + \mathcal{T}_{fin} \\
\mathcal{T}_{init,min} &= R_I + 2R_L \\
\mathcal{T}_{1^{st},min} &= 2R_L + \mathcal{T}_C(n_{th} - 1) + \mathcal{T}_{N_{cl}} \\
\mathcal{T}_{rest,min} &= (n_{th} - 2)(R_R + R_L + \mathcal{T}_{N_{cl}}) \\
\mathcal{T}_{fin,min} &= 2R_R + R_L
\end{aligned} \tag{4.8}$$

This algorithm will use the same notification structure than the small broadcast with one modification. Since only one thread accesses to this information in each stage, it is possible to have flag, address and notification in the same line, allowing the receiver to fetch it only once during the stage. Moreover, the owner of the line only checks the notification at the end of the whole algorithm, minimizing interference. The model has been divided in four parts:

1. Initialization ($\mathcal{T}_{init}$): every thread checks its notification line and sets the local buffer address.
2. First stage ($\mathcal{T}_{1^{st}}$): the root sets its flag to ready (R_L), the rest of threads check it ($\mathcal{T}_C(n_{th} - 1)$ is an upper bound to the real value because the contention model implies the copy of a line and in this scenario threads only read the value), copy of the first slice of the message ($\mathcal{T}_{N_{cl}}$ using the multi-line model) and sets its own flag to ready (R_L).
3. Rest of copy stages ($\mathcal{T}_{rest}$): the rest of packets ($n_{th} - 2$) are copied, including the check for readiness (R_R) and the notification to the owner (R_L).
4. Finalization ($\mathcal{T}_{fin}$): each thread checks for completion (R_R) and sets the own structure as free (R_L). The extra R_R represents the notification to the root. To avoid interference and serialization in this notification, each thread will notify the first copy in a different stage.

Having only one thread accessing one location at every stage minimizes interference, however, there are still some points in which it can appear. In $\mathcal{T}_{1^{st}}$, as happened in the notification forwards from Equation (4.6), the polling threads can interfere with the root. Moreover, in $\mathcal{T}_{rest}$, any thread (e.g., T_2) can finish its stage earlier tan others and try to read a flag before it is set, e.g., by T_1. When setting it, T_1 forces the line to be evicted from T_2's cache, that will have to fetch it again later. And finally, it is possible to assume that the last thread writes the notification after the first check for completion, adding some extra costs.

$$\begin{aligned}
\mathcal{T}_{init,max} &= R_I + 2(R_R + \mathcal{T}_C(n_{th} - 1)) + R_R \\
\mathcal{T}_{1^{st},max} &= 2R_R + \mathcal{T}_C(n_{th} - 1) + \mathcal{T}_{N_{cl}} \\
\mathcal{T}_{rest,max} &= (n_{th} - 2)(2R_R + \mathcal{T}_{N_{cl}}) \\
\mathcal{T}_{fin,max} &= 2R_R + R_L
\end{aligned} \tag{4.9}$$

Although this algorithm minimizes contention and interference, it can also preclude the benefits of prefetching (Equation (3.1)) that is only exploited for each packet.

Thus, we analyze a second algorithm, a flat tree, that takes fully advantage of prefetching since all receivers access the whole message after the root notified them. This algorithm ends when the receivers acknowledge the root that they have copied the message. Since the number of threads colliding is large, the notification system uses two lines, in the same way as the small broadcast. The analysis to be done here is how contention affects the performance of requesting multiple contiguous lines, thus, we have to combine the contention and the multi-line models. For this purpose, we use the slope factor of the multi-line ping-pong model (o) as the time that it takes for one thread to get the message. This operation will be affected by the congestion caused by the rest of threads but the root ($n_{th} - 2$). As intercept or constant factor, we arbitrarily chose the b from the contention model (assuming that the buffers are in exclusive state). In this scenario, the Flat Tree algorithm represents a good tradeoff between the benefits of prefetching and the drawbacks of contention. The rest of themodel is equivalent to one stage of the small broadcast tree. Equation (4.10) reflects the best and worst models for this

algorithm.

$$\begin{aligned} \mathcal{T}_{ftbcast} &= \mathcal{T}_{notif} + \mathcal{T}_{copy} \\ \mathcal{T}_{copy} &= b + c \cdot (n_{th} - 2) + o \cdot N \\ \mathcal{T}_{notif,min} &= R_I + 3R_L + R_R + T_C(n_{th} - 1) + (n_{th} - 1)R_R \\ \mathcal{T}_{notif,max} &= R_I + R_L + 3R_R + 2T_C(n_{th} - 1) + 2(n_{th} - 1)R_R \end{aligned} \tag{4.10}$$

We expect the second algorithm (flat tree) to perform better for large message sizes.

4.2 Barrier Synchronization

A barrier synchronization involves every thread acknowledging that every other thread has reached the synchronization point. We have modeled it as a dissemination barrier since it has been proven to be the best algorithm for single-port LogP systems, but optimizing the parameters within our min-max models. The dissemination algorithm uses $r = log_m(n_{th})$ rounds in which thread T sends a notification to thread $(T + i(m+1)^r) \mod n_{th}, 0 < i \leq r$ and waits for the notifications from $(T - i(m+1))^r) \mod n_{th}, 0 < i \leq r$. In our shared memory scenario, assuming that every thread owns a notification line, each "send" operation consists of setting a flag and waiting until the receivers acknowledge that they have read this flag; and, "receive" is to notify to the senders the read of the corresponding flags. The pseudocode for this operation is shown in Figure 4.3.

```
/* Type definition and initialization */
typedef struct synchro{
        int flag;
        int notification;
        char padding[56];  //each struct occupies one cache line
}synchro_t;

syncrho_t Synchro[nthreads]; //shared array with one flag per thread.

initialize(Syncrho); //set flag to -1 and notification to 0


/* Barrier method */

rounds = ceil(log(m,nthreads));
for(r = 0; r < rounds; r++){
        synchro[thread_id].notification = 0;
        synchro[thread_id].flag = r;
        for(i = 1; i <= m; i++){
                peer = (thread_id-i*(m+1)^r) mod P;
                while(synchro[peer].flag < r){}
                synchro[peer].notification++;
        }
        while(synchro[thread_id].notification < m){}
}
```

Figure 4.3: Pseudo-code of the dissemination barrier for shared memory

In the best case (min), the owner was the last reader of its line (to check its value in the previous round), having it in cache when setting it to ready (R_L), and, the cost of checking it after every receiver has finished is R_R. Moreover, it has to read m threads' flags and, assuming no interference and that flags are already set, the thread will read and write to them just fetching each line once. Although every thread has to read m lines, they are not contiguous and exposed to be prefetched, thus we will not apply the multi-line model. The contention model does not apply either because, although m threads are accessing to each line, they are performing writes that have to be serialized. The total cost is shown in Equation (4.11). The m value must be chosen to minimize this cost.

$$\begin{aligned} \mathcal{T}_{barr,min} &= r(R_L + R_R \cdot m + R_R) \\ r &= \lceil log_m(n_{th}) \rceil \end{aligned} \tag{4.11}$$

However, in every round, the own line can be in other core's cache, e.g. if other thread is already checking the flag,(R_R) and the notification value can be checked once and every time that it is modified

by a notifier thread ($(2m+1)R_R$). Finally, if the first read of other thread's flags results in failure (the flag has not been set yet), at least another read of the line has to be performed. Taking into account that other $m-1$ threads can get the line and modify it in between, this interference could result in $(3m) \cdot R_R + m(m-1)R_R$. Since it is unlikely to happen, the model includes only one interference per line m .

$$\begin{aligned} \mathcal{T}_{barr,max} &= r(R_R + 4m \cdot R_R + (2m+1)R_R) \\ r &= \lceil log_m(n_{th}) \rceil \end{aligned} \tag{4.12}$$

The best m can again be found using numerical methods.

4.3 Small Reduction

A reduction is the application of an operation to data collected from all threads. In this section we will analyze the implementation of the reduction of one item.

In a reduction, the root is receiving from multiple threads, thus, performing a communication pattern which is exactly the opposite to the broadcast on. A first approach could be having all those threads writing to a common location. Then, each thread will have to:

1. check a flag to see if the buffer is ready (R_R),
2. read the buffer (R_L),
3. apply the reduction operation to the buffer using its private data (R_L),
4. write the result to the data buffer and
5. notify that it has finished (R_R).

If several threads are accessing to the same buffer, steps 2 to 5 have to be performed in an atomic manner, thus, serializing. To avoid serialization, the root has several buffers in which each descendant writes its data. Then, the root reads them all and performs the operation.

This scheme can be structured similarly to the broadcast tree. Each thread from level i has k_i buffers where its k_i children copy their own data. Then, the parent performs the operation with the data from these buffers. In each stage of the tree, the parent has to set a flag (R_I) that their children (k_i) read (causing some contention) before writing to the corresponding buffer ($R_R + R_L$) and notifying that the data is ready (R_R). Once the parent gets the acknowledgment (R_R), it performs the operation (which is modeled using the multi-line model). The tree minimizing Equation 4.13 forms our solution.

$$\mathcal{T}_{red,min} = \sum_{i=1}^{d} [R_I + \mathcal{T}_C(k_i) + (1+k_i)R_R + R_L + \mathcal{T}_{k_i}] + R_R \tag{4.13}$$

The interference in the notification forwards (some threads read the parent's flag before it is set) and in the notification backwards (the parent checks the notification before it is complete) is reflected in the worst case (max) in Equation (4.14).

$$\begin{aligned} \mathcal{T}_{red,max} = &\sum_{i=1}^{d} [R_I + 2\mathcal{T}_C(k_i) + 2(1+k_i)R_R + R_L + \mathcal{T}_{k_i}] \\ &+ R_R \end{aligned} \tag{4.14}$$

Here again, we compute the optimal d and k_is using numeric techniques.

4.4 Evaluation

The evaluation of the designed algorithms has been performed on an Intel Xeon Phi 5110P with 60 cores at 1052 MHz, the host machine is an Intel Xeon E5-2670 Sandy Bridge with 8 cores at 2.60 Ghz. The Intel MIC software stack is the MPSS Gold update 2.1.4346-16, with the Intel Composer

XE 2013.0.079, the Intel Compiler v.13.0 and Intel MPI v.4.1.0.024. The benchmarks used are the EPCC OpenMP Benchmarks 3.0 and the Intel MPI Benchmarks (IMB) 3.2. The goal of this section is to check whether the model predictions are accurate enough and compare the results with existing solutions

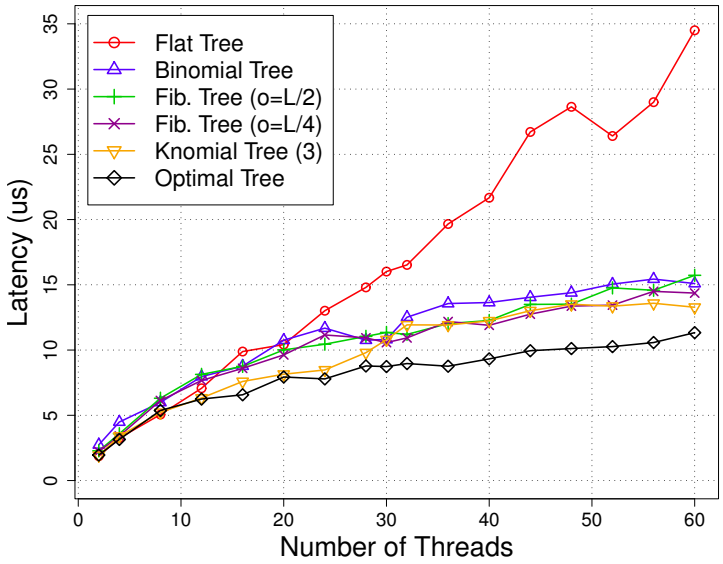


Figure 4.4: Small Broadcast performance comparing our optimal algorithm with widely used broadcast trees.

The benchmarks that measure the performance of our algorithms were developed to ensure a given cache state in each of the 1000 iterations. Before each one of these iteration, threads are synchronized with a custom `RDTSC`-based synchronization and the data lines are placed in the desired cache state.

To guarantee that all threads start at the same time we used the synchronization system explained in Section 3.2 based on the consistency of the `RDTSC`. Thus, we generate time intervals for threads to achieve before starting, then assuring that they enter each operation at the same time. A second synchronization before the collective operation is performed. The time is measured for every operation call and the whole distribution of times is used for statistical analysis of the obtained results.

Before testing, we have obtained the best parameters for all the parametrizable algorithms (small broadcast, reduction and synchronization) by minimizing the best case models. The optimization based on the worst case equations was also taken into account with similar results, hence, only the parameters obtained for the best case model are shown in the graphs. As an example, when having 30 threads, the parameters for a small broadcast were $d = 2$, $k_1 = 5$, $k_2 = 5$; for reduction $d = 3$, $k_1 = 3$, $k_2 = 3$, $k_3 = 2$; and for barrier $m = 6$ ($r = 2$). For 60 threads, the parameters were $d = 3$, $k_1 = 4$, $k_2 = 4$ and $k_3 = 3$ for small broadcast and reduction, and $m = 4$ ($r = 3$) for barrier. Parameters differ for each number of threads and that is the reason of some variations in the models as seen around 28 processes in Figure 4.7. Appendix B contains all the parameters used for every parametrizable algorithm.

All benchmarks launch one thread per core and, when using 60 threads, the variability increases because it is not possible to avoid the core that runs the OS.

Figure 4.4 shows a comparison between different algorithms for small broadcast: flat tree, binomial tree, k-nomial tree (k=3), Fibonacci tree and our optimal tree, all of them with buffers in E state. For Fibonacci trees [12], given that they are designed for LogP and that in this system it is not exactly applicable, we have chosen $o = L/2$ and $o = L/4$ to construct the tree. As expected, the optimal algorithm developed using the model obtains the lowest latency even though, some of the other algorithms, e.g., Fibonacci Trees, are optimal in other models.

Figures 4.5 to 4.8 represent the performance obtained with the algorithms modeled in Chapter 3. Table 4.1 summarizes the min-max models of the benchmarked algorithms. Results are presented with the corresponding boxplots and the min-max model. The large broadcast uses 8 kb messages because it is the higher buffer size that was modeled for the multi-line ping-pong, and the operation used in

Table 4.1: Min-Max models of the benchmarked algorithms

Small Broadcast

$$\mathcal{T}_{sbcast} = \min_{d,k_i}\left(\mathcal{T}_{fw} + \sum_{i=1}^{d}(c\cdot k_i + b) + \sum_{i=1}^{d}\mathcal{T}_{nb}(k_i+1)\right)$$

$$\mathcal{T}_{nb,min}(n_{th}) = R_I + (n_{th}-1)\cdot R_R \qquad \mathcal{T}_{nb,max}(n_{th}) = R_I + 2(n_{th}-1)\cdot R_R$$

$$\mathcal{T}_{fw,min} = (d+1)R_I + 2dR_L \qquad \mathcal{T}_{fw,max} = R_I + \sum_{i=1}^{d} 2\left(R_R + \sum_{i=1}^{d}(c\cdot k_i + b)\right)$$

Large Broadcast (Pipelined)

$$\mathcal{T}_{pipedbcast} = \mathcal{T}_{init} + \mathcal{T}_{1^{st}} + \mathcal{T}_{rest} + \mathcal{T}_{fin}$$

$$\mathcal{T}_{init,min} = R_I + 2R_L \qquad \mathcal{T}_{init,max} = R_I + 2(R_R + \mathcal{T}_C(n_{th}-1)) + R_R$$

$$\mathcal{T}_{1^{st},min} = 2R_L + \mathcal{T}_C(n_{th}-1) + \mathcal{T}_{N_{cl}} \qquad \mathcal{T}_{1^{st},max} = 2R_R + \mathcal{T}_C(n_{th}-1) + \mathcal{T}_{N_{cl}}$$

$$\mathcal{T}_{rest,min} = (n_{th}-2)(R_R + R_L + \mathcal{T}_{N_{cl}}) \qquad \mathcal{T}_{rest,max} = (n_{th}-2)(2R_R + \mathcal{T}_{N_{cl}})$$

$$\mathcal{T}_{fin,min} = 2R_R + R_L \qquad \mathcal{T}_{fin,max} = 2R_R + R_L$$

Large Broadcast (Flat Tree)

$$\mathcal{T}_{ftbcast} = \mathcal{T}_{notif} + \mathcal{T}_{copy}$$

$$\mathcal{T}_{copy} = b + c\cdot(n_{th}-2) + o\cdot N$$

$$\mathcal{T}_{notif,min} = R_I + 3R_L + R_R + \\ + T_C(n_{th}-1) + (n_{th}-1)R_R \qquad \mathcal{T}_{notif,max} = R_I + R_L + 3R_R + \\ + 2T_C(n_{th}-1) + 2(n_{th}-1)R_R$$

Barrier Synchronization

$$\mathcal{T}_{barr,min} = r(R_L + R_R\cdot m + R_R) \qquad \mathcal{T}_{barr,max} = r(R_R + 4m\cdot R_R + (2m+1)R_R)$$

$$r = \lceil log_m(n_{th})\rceil$$

Small Reduction

$$\mathcal{T}_{red,min} = \sum_{i=1}^{d}[R_I + \mathcal{T}_C(k_i) + \\ + (1+k_i)R_R + R_L + \mathcal{T}_{k_i}] + R_R \qquad \mathcal{T}_{red,max} = \sum_{i=1}^{d}[R_I + 2\mathcal{T}_C(k_i) + \\ + 2(1+k_i)R_R + R_L + \mathcal{T}_{k_i}] + R_R$$

the reduction is a summation. As it can be seen, the min-max model is able to capture the inherent variability of the use of threads and allowed us to obtain the best parameters for the small broadcast, the small reduce and the barrier.

To compare the results with current shared memory communication solutions, the graphs also include the latency obtained with MPI and OpenMP (when applicable). It is worth mentioning that the benchmarks used for OpenMP and MPI measure the average result without synchronizing threads before each iteration and without forcing any cache state, avoiding the eviction of the shared data and taking advantage of temporal locality across iterations. Our benchmark forces the data to be in exclusive state in the buffer owner's cache, thus invalidating it in any other cache. However, even in that case, our algorithms outperform MPI and OpenMP except for two scenarios. In Figure 4.7, with 60 threads, the OpenMP barrier obtained a latency that is lower than our algorithm, however, it seems that it is highly optimized for a large number of threads while ours is optimized separately for each number of threads. Moreover, they take advantage of the non-cache-invalidation policy between iterations used in the benchmarks. In the results of the large broadcast (Figure 4.6), the "pipelined" algorithm is outperformed by MPI when the number of threads is larger than 32, although it does not happen if the flat tree algorithm is used. As mentioned in Section 4.1.3, the flat tree obtains a good tradeoff between contention and prefetching while the pipelined algorithm is not able to take advantage of prefetching.

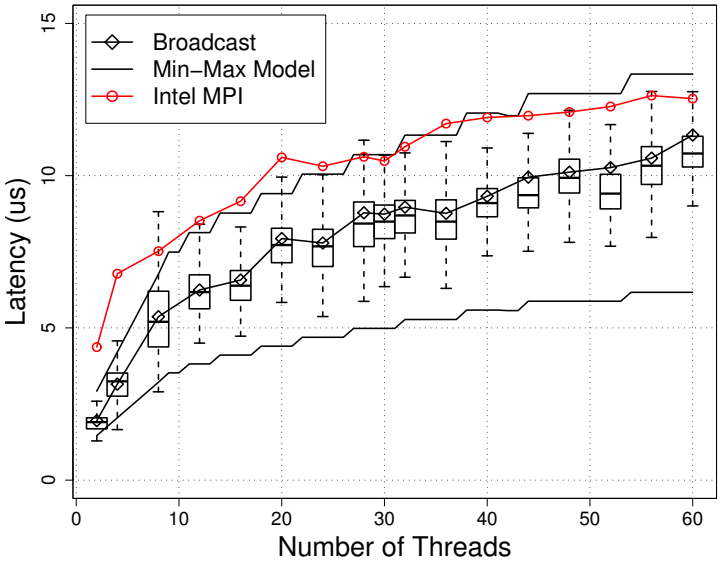


Figure 4.5: Small Broadcast performance compared to the model and the Intel MPI implementation.

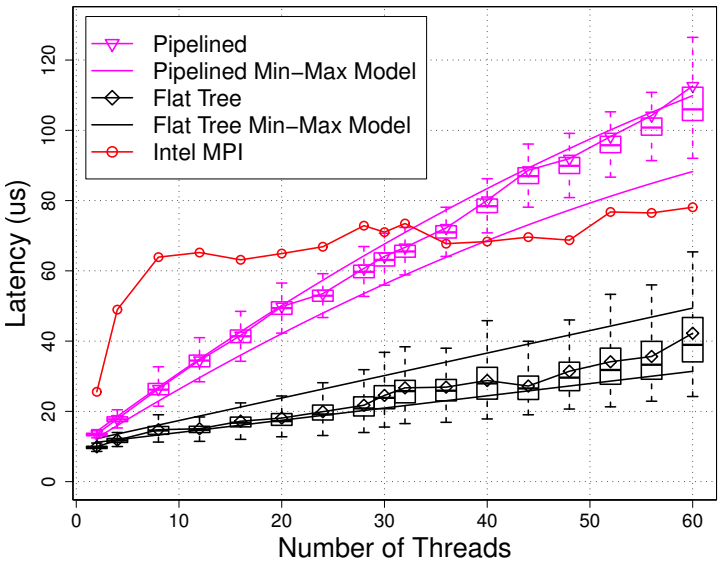


Figure 4.6: Large Broadcast (8 kb) algorithms compared to the model and the Intel MPI implementation.

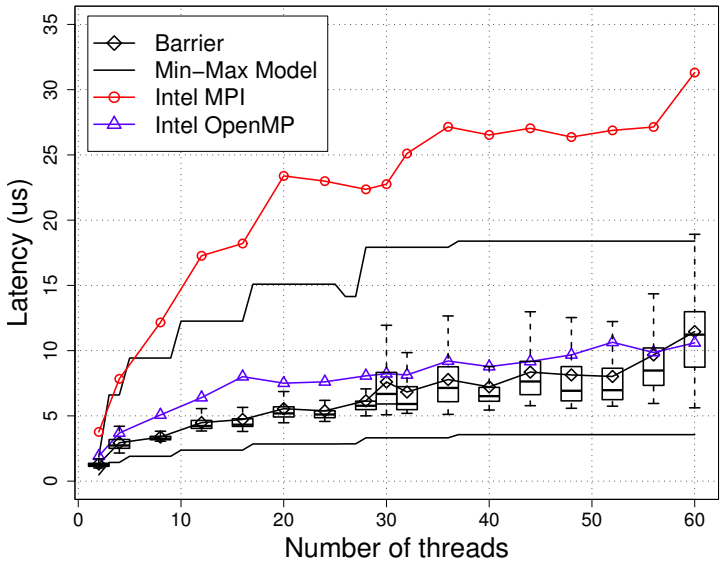


Figure 4.7: Barrier Synchronization results compared to the model and the Intel OpenMP and MPI implementations.

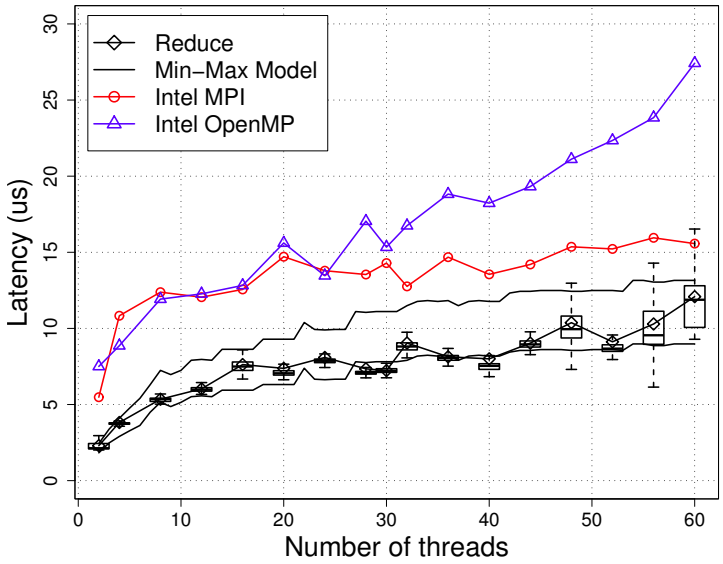


Figure 4.8: Small Reduction performance compared to the model and the Intel OpenMP and MPI implementations

Chapter 5

Conclusions

5.1 Related Work

The optimization of parallel computation is based on the study of the architectural features that can influence performance. However, algorithm design requires models that simplify and abstract complex systems, e.g., LogP [6], LogGP [2], PlogP [13] or Hockney [8] model the communications in distributed memory systems. On the other hand, models like PRAM [11], that assume that processors can access global memory without cost, study the logical structure of parallel computation removing communication from the analysis.

These models have been successfully used to design optimal communication algorithms. In [12], Karp et al. show that Fibonacci trees are optimal for small broadcasts. The authors of [18] use a simple linear communication model to develop bandwidth-optimal broadcast and reduction algorithms.

With the increase in the number of cores per processor, the modeling of shared memory communications is also crucial to develop efficient algorithms to transfer information through shared memory among the cores of the system. Petrovic et al. [17] discuss communications in the precursor of the Intel Xeon Phi, the Intel SCC. However, this system did not provide cache coherency, which simplifies the interactions among threads greatly.

The cache coherency protocols have been also widely studied, specially in terms of internal memory hierarchy models analyzing the effects of evictions and memory locality. Agarwal et al. [1] present a comprehensive model for associative caches and other works like [19] or [3] study the behavior of the memory hierarchy on multi-core systems, but focus on the behavior of caches and the optimization of parallel codes avoiding cache misses, and does not discuss the effects of communications among cores.

Early experiences on Intel Xeon Phi coprocessor [5] showed that this architecture provides scalable performance, which combined with the possibility of obtaining highly parallel applications with standard programming paradigms, makes it really interesting to explore the communications among cores in a shared memory environment.

5.2 Discussion and Conclusions

We found that, especially for small data, the notification system and interference caused by threads in the polling stages, can impact performance more than the actual data transfer. In order to model these effects, we had to resort to min-max models that complicate the algorithm development considerably. Nevertheless, our model allows algorithm designers to abstract away from the architecture and the detailed cache coherency protocols and design algorithms on purely analytic ground. We showed that our models can be combined into a powerful framework for tuning and developing parallel algorithms.

Regarding the applicability to other architectures, the simplified cache model should be adapted to the specific characteristics of the processor and cache coherency protocol that it is being analyzed. As an example, for Sandy Bridge, it would be necessary to take into account that the use of cores from different chips will result in performance differences. However, the steps to follow are basically the same:

1. Analysis of cache states and latency of line transfers among cores, carrying out a clusterization of the results.

2. Analysis of multi-line transfers taking into account the clusterized model and using linear regression.

3. Analysis of congestion of accesses to the same line, again, possibly fitting a linear regression model.

To develop these three preliminary steps, we have developed a group of benchmarks to complement the BenchIT memory benchmark that provides one-line transfers latency and that we plan to make available soon. Once the pieces are stated, the algorithms just have to be adapted to the specific features of the processor that is being used.

In general, we found that optimizing for cache-coherency protocols is harder than optimizing for systems that offer direct remote memory access. The developed models and techniques are more complex than, for example algorithms in the LogP model. Based on results gathered in [17], we would assume that direct remote cache access (DRCA) would lead to parallel systems with higher performance and better predictability and transparency. Thus, we conjecture that DRCA would greatly simplify the design of parallel algorithms.

However, if such architectures are not an option, our models describe a viable method for designing parallel algorithms on cache-coherent architectures. Indeed, our simplified model can be used rather mechanically to optimize and parametrize well-known algorithms. In addition, we showed how to develop new and optimal algorithms requiring slightly more effort. While all our models do not provide precise predictions rather than a range of possible performance, we demonstrated how they can be used to guide algorithm design and development.

The algorithms we developed with the help of our analytical models show performance improvements over Intel's hand-tuned MPI and OpenMP libraries in nearly all configurations with a maximum improvement of 4.3 times. Our method can also be used for other architectures and algorithms.

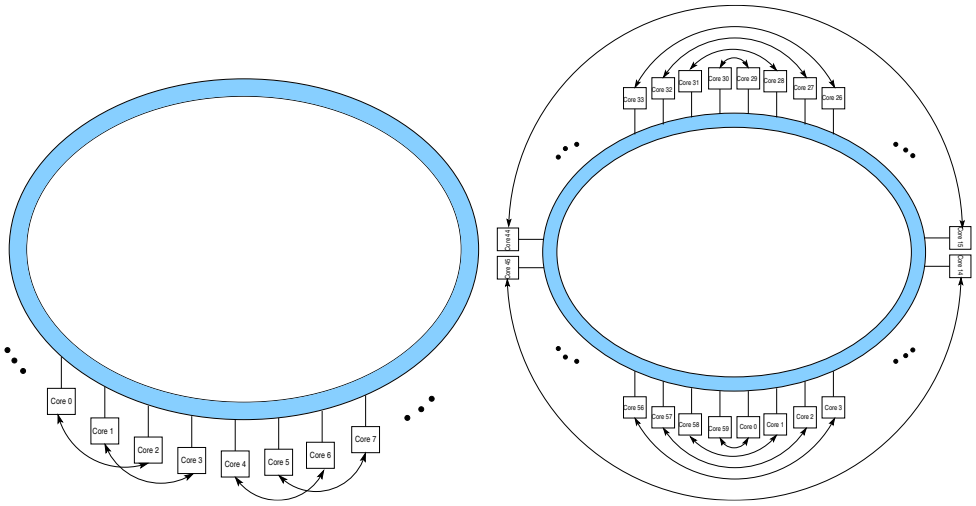
Appendix A

Preliminary Experiments in Bandwidth Analysis

The analysis of the communication bandwidth has been carried out with multi-line ping-pong benchmarks performed between several pairs of threads at the same time. These benchmarks are very similar to the one-line but, while the sender reads all the lines in sender and receiver buffers to perform the copy, and assuming `x86` memory ordering, the receiver only reads the last line to check the canary value. The benchmarks perform several iterations and all threads are synchronized before each one. Moreover, buffers are in I for every test because we use buffers that are larger than the cache size. The time is measured for all ping-pong pairs and the average is used to calculate the bandwidth. It is also worth mentioning that the bandwidth shown represents the bandwidth of the transfer of one message while the operation actually has to transfer two buffers to perform the copy. Copy of data is performed with vectorial instructions. The Sandy Bridge architecture supports the AVX2 instruction set, with registers up to 128 bits (32 bytes). The Xeon Phi provides a new set of vectorial instructions with 512 bits (64 bytes) operands.

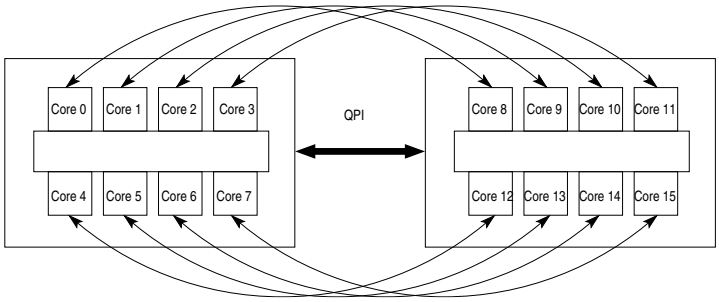
To gain insight in the effect that having several communicating pairs has on performance, we have designed several tests in which threads are located differently along the rings and the QPI. The configurations used are shown in Figuree A.1. The first one (Figure A.1a), uses groups of four threads in which the pairs communicating are interleaved. As an example, the figure shows two groups with interleaved pairs on Xeon Phi.

The second one (Figure A.1b) is designed to force every pair of communicating processes to go across the same link. It assumes that communications will always go through the shortest path. In the figure, 30 pairs of cores are communicating on Xeon Phi using this configuration. When using more than half of the total number of cores, the extra pairs will communicate through the other half of the ring. However, on Sandy Bridge, only 8 cores can use the same links inside one processor, thus, a third configuration has been used to measure the bandwidth of the QPI link (Figure A.1c). Each thread from the first processor has a pair in the second one, hence, all pairs have to use the QPI link to communicate.



(a) Groups of four

(b) Stressed Link



(c) Stressed QPI

Figure A.1: Bandwidth Benchmarks Configurations

A.1 First Attempt of Bandwidth Modeling

With this results, we tried to obtain a first approach to the model of the bandwidth depending on the number of running threads on Sandy Bridge and the message size. Analyzing how the ping-pong test is carried out, the sender must fetch every buffer line from memory, which could be estimated as: $\frac{2N}{P}R_I$ where N is the number of cache lines of each buffer and P is the number of outstanding requests that can be performed per core. Most of the times, there is no difference between R_L and R_R because the access time to a specific line does not depend on which core has reserved the memory, except when threads are running in cores from different processors connected by QPI. Thus, when analyzing the application of the model on Sandy Bridge, this difference will be discussed. Then, the receiver has to read the last line, that has been modified by the sender: $R_{R,M}$. However, this model did not capture the real performance because it does not have into account several factors that can influence performance as cache evictions, prefetching, capacity of the DTD or the memory to solve the outstanding requests, etc. To estimate this effects, we introduced a multiplier factor $\mathcal{F}$ to be empirically adjusted for each scenario.

$$\begin{aligned} \mathcal{T}_N &= (\frac{2N}{P}R_I + R_{R,M})\mathcal{F} \\ BW &= \frac{N \cdot CS}{\mathcal{T}_N \cdot 10^{-9}} \quad (MB/S) \\ N &= \text{number of cache lines} \\ CS &= \text{cache line size (bytes)} \end{aligned} \tag{A.1}$$

A.2 Results on Sandy Bridge

Results of this benchmark on Sandy Bridge are shown in Figures A.2, A.3 and A.4. Except for the QPI test (Figure A.4), pairs of threads only communicate within a processor, hence, when the use of 16 cores makes almost no difference, regarding the bandwidth obtained with 8 cores. Having 2 or 4 cores seems not to degrade performance, even when using the QPI link for every communicating pair. However, performance starts to degrade when increasing the number of threads over 8, especially for messages larger than 2 MBytes when using QPI, and it degrades severely when using 16 cores communicating across QPI.

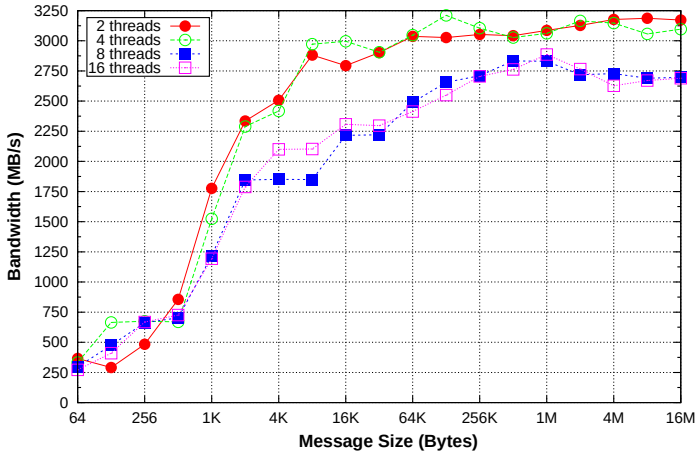


Figure A.2: Bandwidth on Sandy Bridge using multi-line ping-pongs among pairs of threads grouped

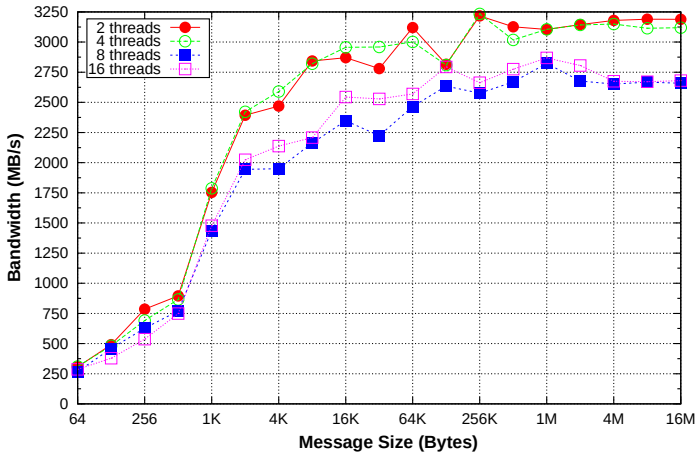


Figure A.3: Bandwidth on Sandy Bridge using multi-line ping-pongs among pairs stressing one link inside each processor.

The parameter P has been adjusted empirically to 38.

Results within a processor show that there are two scenarios: the use of less than 8 threads per processor and the use of 8 threads per processor. Using 8 or 16 threads per node does not make a difference when communications are performed inside each processor. However, the curves are very similar and we can use the same model adjusting the overhead factor. Figure A.5 shows the preliminary model for this cases. When using 2 and 4 threads, the $\mathcal{F}$ parameter is set at 5.5, and for 8 and 16

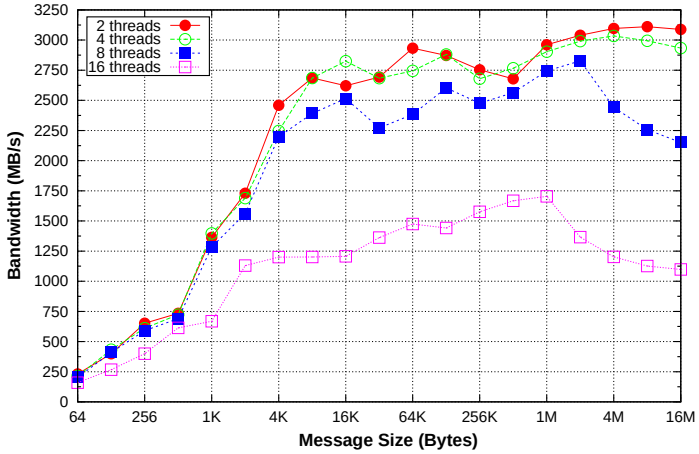


Figure A.4: Bandwidth on Sandy Bridge using multi-line ping-pongs among pairs stressing the QPI link between both processors

threads, it is set at 6.25. This overhead has been estimated empirically relying on the obtained results.

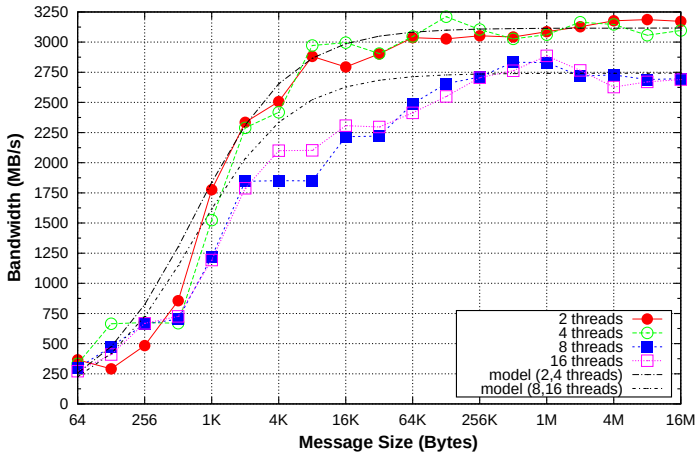


Figure A.5: Bandwidth model for Sandy Bridge when communications are performed within a processor

To evaluate the accuracy of the model, Figure A.6 show the differences in percentage between the basic model and the real results, for both 2 and 4 threads in the benchmark that uses groups of four, and in the benchmark that stresses one internal link. Although the model seems to estimate accurately the performance of large messages, there are some issues regarding the estimation of the bandwidth of the transfer of smaller messages, where this simple model is not able to capture the real performance.

Figure A.7 represent the percentage of differences for the proposed an the basic model but when using 8 threads within a processor, or 16 threads (8 per processor). Here, there are also some higher differences for messages between 8 and 32 kb. This is because the model does not capture exactly the initial congestion.

Regarding communications across QPI, there are several aspects that have to be taken into account in order to adjust the model. First of all, the number of outstanding requests (P) is estimated to be 19 instead of 38. Another change is that the estimation of the time for the R_I uses the latency of the transfer across the QPI link. Although the send-buffer is fetched from the local memory, since we are assuming that the send and the recv buffer are fetched simultaneously, it makes sense to use the latency

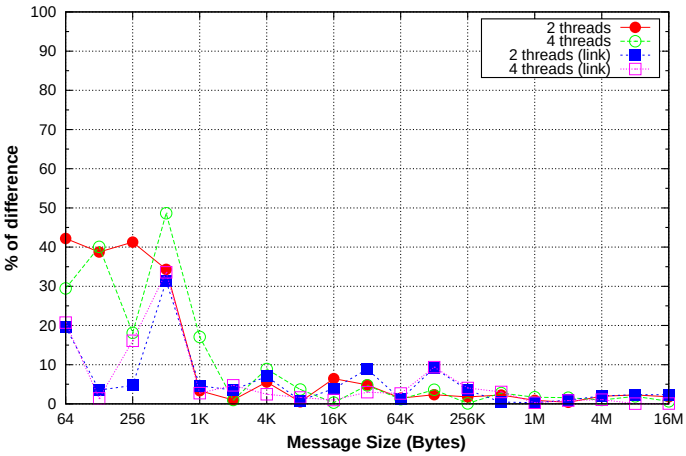


Figure A.6: Percentage of differences between real results and the model for Sandy Bridge when using less than 8 threads within one processor

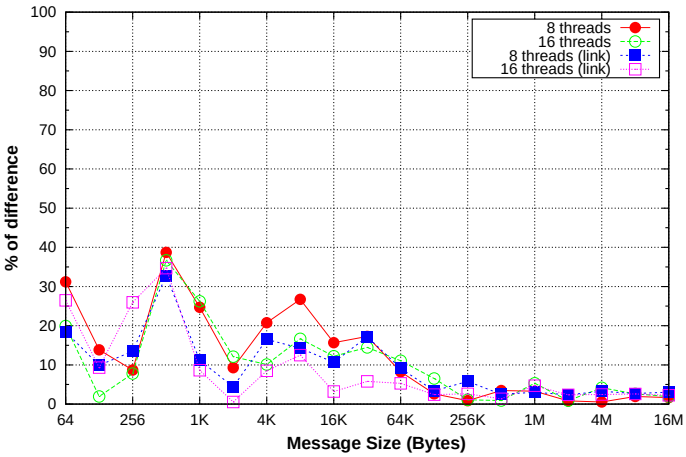
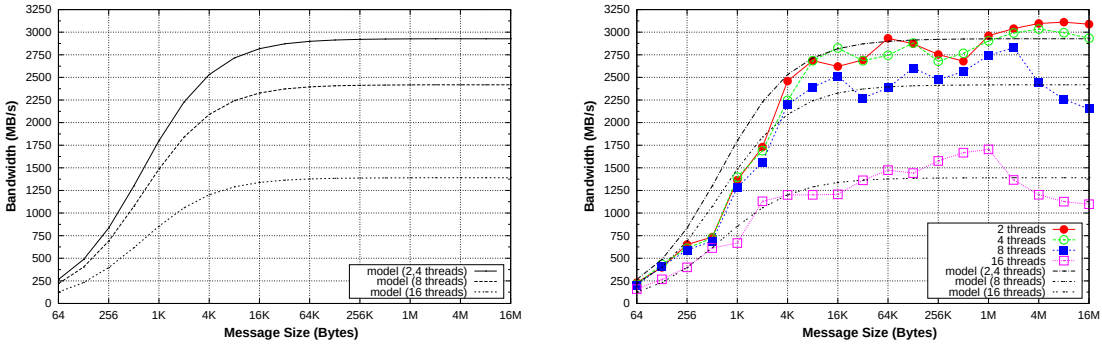


Figure A.7: Percentage of difference between real results and the model for Sandy Bridge when using 8 threads within one processor (or 16 threads, 8 per processor)

of the slowest request. Finally, the bandwidth benchmarking exposes that there are three scenarios: 2 or 4 threads, 8 threads and 16 threads. In the first case, the overhead factor has been estimated as 1.9, in the second scenario, as 2.3 and, in the third one, as 4.

Figure A.8 shows the model with the parameters discussed above.For 8 and 16 threads, the real data shows a fall of the bandwidth at 2 MBytes for 8 threads and 1 MByte for 16 threads. This would require a discontinuous function to capture this congestion effect. However, to simplify the modelling, the function has been treated as continuous.

Figure A.9 shows the percentage differences of performance between real data and the basic model. It can be observed that there are some disruptions for large messages when using 8 or 16 threads due to the issues analyzed above.



(a) Bandwidth of the proposed and basic models

(b) Bandwidth of the models compared to the real data

Figure A.8: Bandwidth models for Sandy Bridge when communications are performed across the QPI link

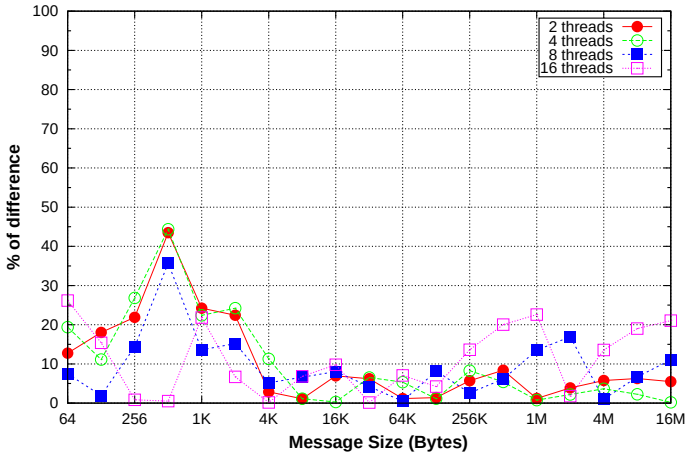


Figure A.9: Percentage of difference between real results and the models for Sandy Bridge when communications are performed across the QPI link

A.3 Results on Intel Xeon Phi

These results, shown on Figures A.10 and A.11, have been obtained using up to 60 cores on a testbed KNC B0 with 61 cores and 1080 MHz previous to the commercial version. Bandwidth is almost five times lower than for Sandy Bridge but it is less affected by the number of cores communicating at the same time, except for messages smaller than 512 kb.

When the message is larger, the bandwidth tends to achieve a stabilized and homogeneous value that is under the peak performance of the ring. Nevertheless, it allows to increase scalability since there is no performance degradation. Differences that appear could be related to the DTD accesses when the buffer lines fetched, since the probability of being using the same DTDs increases when duplicating the number of threads.

Here, there are communications only within the same processor, thus the parameter P has been adjusted to 38 for all the different scenarios.

Figure A.12 compares the model with the results obtained for the stressed link test (for Xeon Phi there is no real difference in bandwidth between both benchmarks). The estimation curve fits the results with an overhead factor of 5.8 established empirically, except for messages under 256 kb. Again, the problem with small messages is that there are effects that this model does not capture accurately and that affects especially when the message is small.

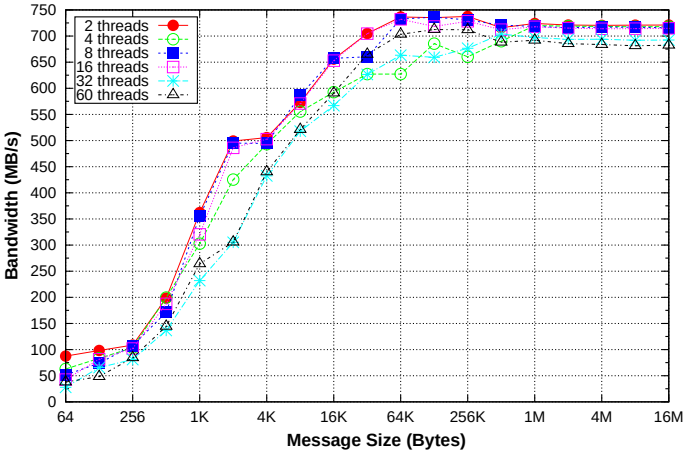


Figure A.10: Bandwidth on Xeon Phi using multi-line ping-pongs among pairs of threads grouped

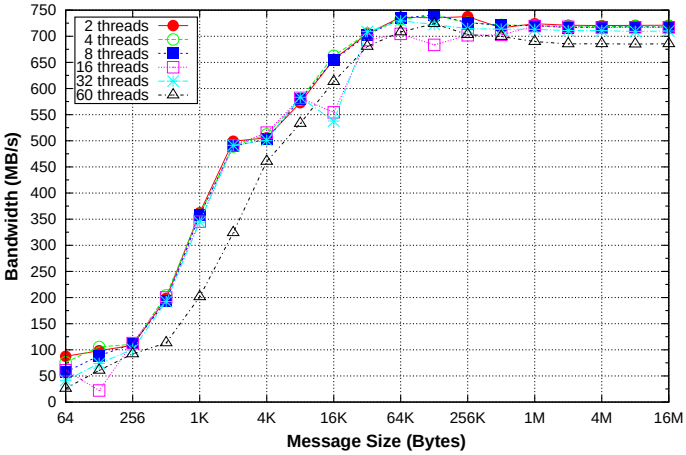


Figure A.11: Bandwidth on Xeon Phi using multi-line ping-pongs among pairs stressing one link

Figure A.13 includes the percentage of the differences between the actual results and the model, confirming that it is very accurate for messages larger than 256 kb.

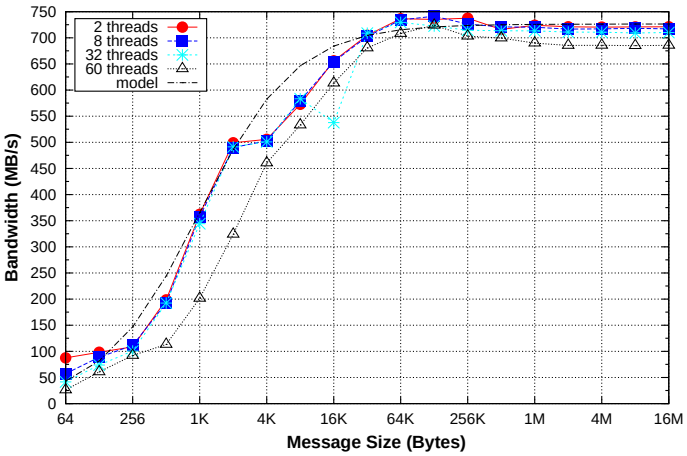


Figure A.12: Bandwidth model for Xeon Phi.

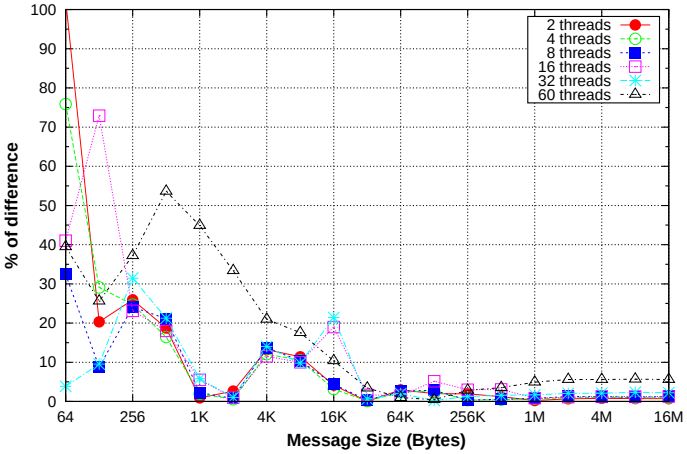


Figure A.13: Percentage of difference between real results and the Xeon Phi bandwidth model.

Appendix B

Optimal Parameters for Communication Algorithms

This Appendix includes the parameters obtained by minimizing equations of Chapter 3 using the best case (min) equations.

B.1 Small Broadcast

Here the parameters to obtain are the height of the tree (d) and the number of descendants of each node from level i (k_i).

$$\begin{aligned} \mathcal{T}_{sbcast} &= \mathcal{T}_{fw,min} + \sum_{i=1}^{d} (c \cdot k_i + b) + \sum_{i=1}^{d} \mathcal{T}_{nb}(k_i + 1) \\ \mathcal{T}_{fw,min} &= R_I + \sum_{i=1}^{d} (R_I + 2R_L) = (d+1)R_I + 2dR_L \\ \mathcal{T}_{nb,min}(n_{th}) &= R_I + (n_{th} - 1) \cdot R_R \\ N &\leq 1 + \sum_{i=1}^{d} \prod_{j=1}^{i} k_j, \quad \forall i < j, k_i \leq k_j \end{aligned} \tag{B.1}$$

Table B.1: Parameters used for the Small Broadcast

number of threads (n_{th})	d	k_1	k_2	k_3
2	1	1	-	-
3	1	2	-	-
4	1	3	-	-
5	1	4	-	-
6	1	5	-	-
7	1	6	-	-
8	1	7	-	-
9	2	3	2	-
10	2	3	2	-
11	2	3	3	-
12	2	3	3	-
13	2	3	3	-
14	2	4	3	-
15	2	4	3	-

Continued on next page

Table B.1 – *Continued from previous page*

number of threads (n_{th})	d	k_1	k_2	k_3
16	2	4	3	-
17	2	4	3	-
18	2	4	4	-
19	2	4	4	-
20	2	4	4	-
21	2	4	4	-
22	2	5	4	-
23	2	5	4	-
24	2	5	4	-
25	2	5	4	-
26	2	5	4	-
27	2	5	5	-
28	2	5	5	-
29	2	5	5	-
30	2	5	5	-
31	2	5	5	-
32	2	6	5	-
33	2	6	5	-
34	2	6	5	-
35	2	6	5	-
36	2	6	5	-
37	2	6	5	-
38	3	3	3	3
39	3	3	3	3
40	3	3	3	3
41	3	4	3	2
42	2	6	6	-
43	2	6	6	-
44	3	4	3	3
45	3	4	3	3
46	3	4	3	3
47	3	4	3	3
48	3	4	3	3
49	3	4	3	3
50	3	4	3	3
51	3	4	3	3
52	3	4	3	3
53	3	4	3	3
54	3	4	4	3
55	3	4	4	3
56	3	4	4	3
57	3	4	4	3
58	3	4	4	3
59	3	4	4	3
60	3	4	4	3

B.2 Barrier Synchronization

The parameter needed to minimize Equation (B.2) is m, because the number of rounds (r) is calculated with this value.

$$\begin{aligned} \mathcal{T}_{barr,min} &= r(R_L + R_R \cdot m + R_R) \\ r &= \lceil log_m(n_{th}) \rceil \end{aligned} \tag{B.2}$$

Table B.2: Parameters used for the Synchronization Barrier

number of threads (n_{th})	m	$rounds = log_m(n_{th})$
2	1	1
3	2	2
4	2	2
5	3	2
6	3	2
7	3	2
8	3	2
9	3	2
10	4	2
11	4	2
12	4	2
13	4	2
14	4	2
15	4	2
16	4	2
17	5	2
18	5	2
19	5	2
20	5	2
21	5	2
22	5	2
23	5	2
24	5	2
25	5	2
26	3	3
27	3	3
28	6	2
29	6	2
30	6	2
31	6	2
32	6	2
33	6	2
34	6	2
35	6	2
36	6	2
37	4	3
38	4	3
39	4	3
40	4	3
41	4	3
42	4	3
43	4	3
44	4	3
45	4	3
46	4	3

Continued on next page

Table B.2 – *Continued from previous page*

number of threads (n_{th})	m	$rounds = log_m(n_{th})$
47	4	3
48	4	3
49	4	3
50	4	3
51	4	3
52	4	3
53	4	3
54	4	3
55	4	3
56	4	3
57	4	3
58	4	3
59	4	3
60	4	3

B.3 Small Reduction

The small reduction (Equation (B.3)) needs the same parameters as the small broadcast: height of the tree (d) and number of descendants of each node in level i (k_i).

$$\mathcal{T}_{red,min} = \sum_{i=1}^{d} [R_I + \mathcal{T}_C(k_i) + (1 + k_i)R_R + R_L + \mathcal{T}_{k_i}] + R_R \tag{B.3}$$

Table B.3: Parameters used for the Small Reduction

number of threads (n_{th})	d	k_1	k_2	k_3
2	1	1	-	-
3	1	2	-	-
4	1	3	-	-
5	1	4	-	-
6	1	5	-	-
7	2	3	1	-
8	2	3	2	-
9	2	4	1	-
10	2	3	2	-
11	2	4	2	-
12	2	3	3	-
13	2	4	2	-
14	2	4	3	-
15	2	4	3	-
16	2	4	3	-
17	2	4	3	-
18	2	4	4	-
19	2	4	4	-
20	2	4	4	-
21	2	4	4	-
22	3	3	2	2
23	2	5	4	-
24	2	6	3	-

Continued on next page

Table B.3 – *Continued from previous page*

number of threads (n_{th})	d	k_1	k_2	k_3
25	2	5	4	-
26	2	5	4	-
27	3	3	3	2
28	3	4	2	2
29	3	3	3	2
30	3	3	3	2
31	3	3	3	2
32	3	4	4	1
33	3	4	3	2
34	3	3	3	3
35	3	4	3	2
36	3	3	3	3
37	3	4	4	1
38	3	4	3	2
39	3	3	3	3
40	3	4	3	2
41	3	4	3	2
42	3	6	2	2
43	3	4	4	2
44	3	4	4	2
45	3	4	3	3
46	3	4	3	3
47	3	4	3	3
48	3	4	4	2
49	3	4	4	2
50	3	4	3	3
51	3	4	3	3
52	3	4	3	3
53	3	4	4	2
54	3	4	4	3
55	3	4	4	3
56	3	6	3	2
57	3	6	3	2
58	3	4	4	3
59	3	4	4	3
60	3	4	4	3

Bibliography

[1] A. Agarwal, J. Hennessy, and M. Horowitz. An Analytical Cache Model. *ACM Trans. Comput. Syst.*, 7(2):184–215, May 1989.

[2] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. LogGP: Incorporating Long Messages into the LogP Model - One Step Closer towards a Realistic Model for Parallel Computation. In *Proc. 7th Annual ACM Symp. on Parallel Algorithms and Architectures (SPAA'95)*, pages 95–105, New York, NY, USA, 1995.

[3] Diego Andrade, Basilio B. Fraguela, and R. Doallo. Accurate Prediction of the Behavior of Multithreaded Applications in Shared Caches. *Parallel Computing*, 39(1):36 – 57, 2013.

[4] George Chrysos. Intel® Xeon Phi™ Coprocessor (Codename Knights Corner). Keynote talk at the 24th Hot Chips: A Symp. on High Performance Chips, 2012.

[5] T. Cramer, D. Schmidl, M. Klemm, and D. an Mey. OpenMP Programming on Intel Xeon Phi Coprocessors: An Early Performance Comparison. In *Proc. Many-core Applications Research Community (MARC) Symp. at RWTH Aachen University*, pages 38–44, November 2012.

[6] David Culler et al. LogP: towards a Realistic Model of Parallel Computation. *SIGPLAN Not.*, 28(7):1–12, July 1993.

[7] Daniel Hackenberg, Daniel Molka, and Wolfgang E. Nagel. Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems. In *Proc. 42nd Annual IEEE/ACM Intl. Symp. on Microarchitecture (MICRO'42)*, pages 413–422, New York, NY, USA, 2009.

[8] Roger W. Hockney. The Communication Challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Computing*, 20(3):389 – 398, 1994.

[9] T. Hoefler and T. Schneider. Optimization Principles for Collective Neighborhood Communications. In *Proc. 25th ACM/IEEE Intl. Supercomputing Conf. for High Performance Computing, Networkin, Storage and Analysis (SC'12)*, Salt Lake City, UT, USA, 2012.

[10] Intel® Xeon Phi™ Coprocessor: Software Developers Guide. `https://www-ssl.intel.com/content/www/us/en/processors/xeon/xeon-phi-coprocessor-system-software-developers-guide.html`, 2012.

[11] Richard M. Karp and Vijaya Ramachandran. A Survey of Parallel Algorithms for Shared-Memory Machines. Technical report, University of California at Berkeley, Berkeley, CA, USA, 1988.

[12] Richard M. Karp, Abhijit Sahay, Eunice E. Santos, Klaus Erik Schauser, and Abhijit Sahay Eunice E. Santos. Optimal Broadcast and Summation in the LogP Model. In *Proc. 5th Annual ACM Symp. on Parallel Algorithms and Architectures (SPAA'93)*, pages 142–153, Velen, Germany, 1993.

[13] Thilo Kielmann, Henri E. Bal, and Kees Verstoep. Fast Measurement of LogP Parameters for Message Passing Platforms. In *Proc. 15th IPDPS 2000 Workshops on Parallel and Distributed Processing*, pages 1176–1183, 2000.

[14] Robert McGill, John W. Tukey, and Wayne A. Larsen. Variations of Box Plots. *The American Statistician*, 32(1):12–16, 1978.

[15] D. Molka, D. Hackenberg, R. Schoene, and M. S. Mueller. Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System. In *Proc. 18th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'09)*, pages 261–270, Raleigh, NC, USA, 2009.

[16] Scott Owens, Susmit Sarkar, and Peter Sewell. A Better x86 Memory Model: x86-TSO. In *Proc. 22nd Intl. Conf. on Theorem Proving in Higher Order Logics (TPHOLs'99)*, pages 391–407, Berlin, Heidelberg, 2009.

[17] Darko Petrović, Omid Shahmirzadi, Thomas Ropars, and André Schiper. High-performance RMA-based Broadcast on the Intel SCC. In *Proc. 24th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA'12)*, pages 121–130, New York, NY, USA, 2012.

[18] Peter Sanders, Jochen Speck, and Jesper Larsson Träff. Two-Tree Algorithms for Full Bandwidth Broadcast, Reduction and Scan. *Parallel Comput.*, 35(12):581–594, December 2009.

[19] Leslie G. Valiant. A Bridging Model for Multi-core Computing. *Journal of Computer and System Sciences*, 77(1):154 – 166, 2011.

[20] B. L. Welch. The Generalization of 'Student's' Problem when Several Different Population Variances are Involved. *Biometrika*, 34(1-2):28–35, 1947.