Errata – Corrige

Page 28, row 31:
replace \( y = F(y_1, y_2) \) with \( y = F(y_0, y_1) \)

Page 30: at the end, insert the following paragraph

A more general view of Stream Processing: Data Stream Processing

Stream Processing is a fundamental characteristic of many important application fields (real-time control, financial trading, networking, web processing, big-data and real-time analytics, etc.), in which data are not always available as classic structures “in memory” only. Instead, some data might be received continuously and in a temporarily variable fashion through communication channels, so they have a limited permanence time and need to the processed “on the fly”. Moreover, in these cases computations operate on many stream elements at the time, with consequent problems of response time, memory occupancy (through data windowing and data synthesis techniques) and latency control. We speak of Data Stream Processing to denote this generalization of Stream Processing. This book study basic stream-based computations in which only one element at the time is processed by a module: by now methodologies for such computations, which have a very large application in any parallelization problem, are known and stable enough. Data Stream Processing is a currently active research topic, and some simple applications are emerging.

Page 43: replace the LC code in the middle of page as follows

\[
\text{parallel CLIENT[N], SERVER; channel request[N], ch_result[N];}
\]

\[
\text{CLIENT[j] :: … channel in ch_result[j], …; channel out request[j], …;}
\]

\[
\text{… send (request[j], (ch_result[j], x)); … receive (ch_result[j], y); …}
\]

\[
\text{SERVER:: … channel in request [N], …; channel out var ch_out; …}
\]

\[
\text{… for (j = 0; j < N; j++)}
\]

\[
\text{\{ receive (request [j], (ch_out, x)); y = F(x); send (ch_out, y) \};}
\]

Page 51, row 11:
synchronous communication \((k = 0)\). Now, …

Page 53, row 2:
are able to \textit{satisfy} this constraint…

Page 64, rows 8 and 13:
with mean value \( T_A \) and variance…

Page 73, table:
replace the ideal service time of \( M_5 \) with \( 100t \)

Page 122, bottom row: correct the formula as follows

\[
L_{\text{par-prefix}} (n, g) = (g - 1) T_{\odot} + \log_2 n \left( T_{\text{send}} (1) + g T_{\odot} \right)
\]
Page 126, row 33:
and latency equal to $2 \log(n) T_{send}$

Page 127: *correct the formula*

$$T_{w-iter} \geq \log(n) T_{send}$$

Page 128: *at the end of the paragraph* “Example of synchronous stencils … this overhead is negligible”, *add the following phrase*

However, the worker code can be written in such a way that the stencil communications are overlapped to calculation, i.e. the Jacobi stencil becomes an asynchronous one. For example, calculating the internal points before the border points.

Page 130: *change the paragraph*

The service time is affected in the following way: all the workers must wait the entire reduce latency, while the multicast (of a small message length $L$) has negligible effect or it is masked, provided that the service time itself is greater than $\log(n)T_{send}(L)$, as seen in Section 7.10.2. Thus, if no other bottleneck exists:

$$T_{s}^{(n)} = T_A + (T_G + T_{send}) \log n$$

with the same implications on $n$ and data parallelism discussed in point a).

Page 130, row 13: *in the scatter formula*

$$T_{scatter}(n, M) = n T_{send}(g)$$

Page 144: *insert the following paragraph before Section 9.1.1*

**General impact of the response time concept**

Response time is a fundamental performance metric in client-server computations with request-reply behavior. However, the utilization of this metric is much more general and is valid for acyclic graph computations too. The basic relation $R_Q = W_Q + L_s$ applies to any server module as well, provided that its *utilization factor is less than one*, i.e. the server is not a bottleneck (this condition is not automatically verified for server modules in acyclic graph structures). For the purpose of this book, the analysis of acyclic graph can be done without using the response time metric explicitly, thus we limit the application of the response time concept to client-server computations with request-reply behavior only. However the reader possesses the technical methods to evaluate the response time in other kinds of computation.

Page 169, row -2:

block $b2$ is not de-allocated if at least one C1-block belonging to $b2$ is currently

Page 196, row –11: *the pseudo-code is*
int A[N], B[N], C[N];

Page 265: Section 17.3.4, last row of third paragraph

DDR SDRAM, DDR2 SDRAM, DDR3 SDRAM, GDDR, and so on, and the most recent 3D-stacked memory technology (see also Section 18.3.7).

Page 270: add the following phrase at the end of Section 17.3.7

Recently, the Processing in Memory (PIM) idea has received much more attention basically for two main reasons: i) both large-scale data-intensive simulations and big-data applications require the minimization of data transfers across memory hierarchies and interconnection networks; ii) memory technology is evolving towards higher bandwidth, lower latency and energy efficient solutions, in particular the 3D-stacked technology, consisting of several DRAM memory layers stacked on top of a single logic layer, all within the same chip package. The logic layer can be exploited for implementing specialized processing functions (e.g., data reduction, vector operations) or general-purpose cores. Intensive research efforts are on-going on the PIM topic.

Page 287: change the last paragraph in

According to the general theory (Sections 2.2, 3.3, 4.4, 10.1, 10.2), the ideal service time per flit is

\[ T_{\text{com}} = \tau + T_{tr} \quad \text{or} \quad T_{\text{com}} = 2(\tau + T_{tr}) \]

depending on the buffering interface discipline.

Page 288: formulas in row 2 and 14 must be respectively changed in

\[ \frac{2}{T_{\text{com}}} \text{flits/sec} \]
\[ \frac{N}{T_{\text{com}}} \text{flits/sec} \]

Page 294, row 5:

Where \( i \) is the row identifier \( (0 \leq j \leq 2^{n-1}) \)

Page 296: insert the following phrase at the end of Section 18.7.1

Independently of these simple cases, it is worth noticing that the actual distance depends on the parallel program to be executed and on its mapping onto the PEs. This is true for \( k \)-ary \( n \)-cubes and for trees (not for monolithic crossbars and \( k \)-ary \( n \)-flies). Notably, the number of processes could be lower than the number of PEs. Moreover in \( k \)-ary \( n \)-cubes, the mapping could be even more critical: as a trivial example, for a pipeline program a mapping with distance equal to one is possible.

Page 309, row -2:

\[ \text{else} \{ \text{waiting PE_id} = \text{get (semlock.queue)}; \text{reset indiv}; \]  

Page 323, row -14:

concerned, then many actions can be performed through direct interactions

Page 326, top: modify the LOCK assembler code as follows

LOCK: LOAD Rsemlock, 0, Rx, set_indiv
and the successive lines are modified as follows

If the block is not currently in cache, it is transferred (from M or) via C2C. If semaphore \( x \) is “red” it is continuously tested in cache. The code contains an optimization: \( \text{indiv} \) is not used in local cache during the waiting loop. Of course, …

Page 327: replace the list of paragraphs (bullets) starting at row -10 (● a while block has to be communicated, ● depending on … PEs contain the block) with the following phrases

Let’s evaluate the whole load of firmware messages exchanged for Invalidation and for Update, in the same conditions, using C2C (neglecting the global control, which has a similar overhead in both cases).

See also Sections 21.3, 21.4 for the base latency evaluation of the needed firmware messages.

Situation: PE\(_i\), PE\(_{j_1}\), …, PE\(_{j_k}\) contain the same updated block in cache, and PE\(_i\) modifies such block.

**Invalidation:**

- PE\(_i\) sends (multicasts) an invalidation message (length \( s = 3 \) 32-bit words: header + physical address) to PE\(_{j_1}\), …, PE\(_{j_k}\);
- PE\(_{j_1}\), …, PE\(_{j_k}\) independently (in parallel) reply with a ‘done’ message (\( s = 1 \): just the header) to PE\(_i\);
- PE\(_{j_1}\), …, PE\(_{j_k}\) independently send a block request to PE\(_i\) (\( s = 3 \)) and receive the block from PE\(_i\) (\( s = 1 + \sigma_1 \)).

**Update:**

- PE\(_i\) sends (multicast) an update message (\( s = 3 + \sigma_1 \)) to PE\(_{j_1}\), …, PE\(_{j_k}\);
- PE\(_{j_1}\), …, PE\(_{j_k}\) independently reply with a ‘done’ message (\( s = 1 \): just the header) to PE\(_i\).

The whole number of exchanged words is slightly greater for Invalidation. However, in Invalidation the communication bandwidth is higher and the communication latency is lower, since multicast is applied to shorter messages, and all the block transfers occur in parallel. Instead, in Update multicast is applied to blocks, thus the block transfers occur serially (unless special architectural support for multicast is provided).

Page 330: insert the following phrase before the last paragraph

Notice that the figure is just a simplified scheme. It is shown that the associative memory and the related hardware resources belong to the W unit. However, this is not necessary: notably, according to the two-level cache organization that will be studied in Section 20.3.1, all these structures belong to the inclusive secondary cache unit (C2), which acts as the local cache controller.

Page 331: modify the last paragraph as follows

A synchronization problem arises when IM sends a remote request to the owner node and, while it is waiting for the reply, it receives another request for the same block. All these actions have to be executed atomically. The solution adopted can be the same used to implement indivisible sequences for lock-unlock atomicity, i.e. the indivisibility bit (\( \text{indiv} \)). IM can use an \( \text{indiv} \) bit for each physical memory block, by using typical synchronization mechanisms.

Page 332: add the following phrase to the last paragraph

```plaintext
STORE Rsemlock, 0, 0, reset_indiv
IF \( \neq 0 \) Rx, EXIT
WAIT: LOAD Rsemlock, 0, Rx
IF = 0 Rx, WAIT
STORE Rsemlock, 0, 0
EXIT: ...
```
See the observation in Section 20.2.2 about the actual allocation of hardware resources for cache control functions.

Page 338, row 16:
Not only block flush reduces home node latency and contention: it also has the

Page 338: complete point 3b as follows
3.b. the referred block \( b \) is de-allocated from the requestor node cache; this action is performed locally by the requestor node. Such technique is also called self-invalidation in the literature.

Page 350: add the following phrase before Section 21.2.1
As remarked several times (e.g., in Section 18.7.1), for some interconnection networks the actual distance depends on the specific parallel program and on its mapping. In these cases, the base memory access latency depends on the parallel program implementation and, in particular, on the parallelism degree and on the adopted process mapping. Possibly, this dependence could be limited, for example for “distance insensitive” networks, but it always exists, except for external memories connected through \( k \)-ary \( n \)-flies.

Page 377, row -16:
Let us denote by \( L_1, L_2, L_3, L_4 \) the base latencies of the above requests types (or of

Page 378: at the end of Section 21.8.4, add the following phrases and figures
The figure shows the case of \( n_{opt}(R_Q) < n_{opt}(R_{Q0}) \).
It is also possible that \( n_{opt}(R_Q) = n_{opt}(R_{Q0}) \), however \( B_{max}(R_Q) \) is still lower than \( B_{max}(R_{Q0}) \):

Finally, if \( n_{opt}(R_Q) > n_{opt}(R_{Q0}) \), we might achieve \( B_{max}(R_Q) = B_{max}(R_{Q0}) \) with \( n = n_{opt}(R_Q) \), while \( B_{max}(R_Q) < B_{max}(R_{Q0}) \) if we maintain \( n = n_{opt}(R_{Q0}) \):

\[
\begin{align*}
B & < 1/T_A(R_{Q0}) \\
B(R_Q) & = B(R_{Q0}) \\
n_{max} & = n_{opt}(R_Q) \\
B & > 1/T_A(R_Q) \\
B(R_Q) & > B(R_{Q0})
\end{align*}
\]
Page 383: correct the false, false case of the receive pseudo-code as follows

```pseudo
false, false: {
    modify CH_Buffer.Size;
    unlock (X);
    target_variable_pointer = CH_Buffer[CH_Buffer.Extraction_Index];
    modify CH_Buffer.Extraction_Index
}
```

Page 387: correct the first statement of RTS(send) and RTS(receive) pseudo-codes as follows

```pseudo
RTS(send)::
    wait until (ACK = 1);
RTS(receive)::
    wait until (RDY = 1);
```

The same correction at page 388 in the pseudo-codes of RTS(receive) with and without message copy.

Page 393, row 12: change the phrase as follows

Let $\beta_{code} = 4T_{instr}$ (where $T_{instr}$ is the mean service time per instruction) be the calculation time of a single iteration in absence of faults or block write operations. For example, $\beta_{code} = 4\tau - 5\tau$ in a scalar CPU.

Page 393: after row 15 (expression of $T_{trans}$) insert the following

If the sender process is executed in the home node, the Store notification is saved, thus:

$$T_{trans-home} \sim \frac{L_{invalidation}}{\sigma_1} + \frac{\sigma_1 + 2}{\sigma_1} \tau + \beta_{code}$$

Page 395: At the end of the last paragraph on Synchronization issues (ending with ... alleviated by using the periodic retry technique), add the following paragraph

The best solution is a simple improvement in the design of C2 unit: if home, a C2C block read request is served only if and when the block has been updated by a write-back operation of the associated C1. As far as the cost model is concerned, in the following we’ll assume that the optimized design of C2 is adopted.

Page 397: 2. Non home run-time support - Modify the last raw of the first paragraph as follows

repeated block transfers or, better, the optimized design of C2 is adopted.

Page 411: figure of the parallel program graph

the names of the channel ch_in and ch_out must be replaced by input-stream and output_stream respectively

Page 413: in <send B> code modify the MOD instruction as follows

```pseudo
MOD Rindex_out, 8, Rindex_out
```

Page 414, 415:

all the instances of symbol PROS must be replaced by PROD

Page 420: comment 4

Stores, though with lower latency, …

Page 425: before Section 24.3 add the following paragraph

**Observation on the “most stressed modules” issue**

The situation of this example about the determination of the most stressed modules must not be considered as a rule. In principle, for a parallel program various possibilities should be analyzed and compared in order to
recognize such modules in the whole graph. However, often a rough analysis of service times and interaction rates is sufficient, like in this example. The reader is invited to find other examples which behave differently from the described one.

Page 425, row -4:

waits ACK = 1 in the first …

Page 425: First paragraph of PROD::, ending with if ACK = 0. Add or, better, the optimized design of C2 is adopted.

The same at page 426 for the first paragraph of CONS::

Page 529, row -5:

change 1205 in 1025