Master Degree Program in Computer Science and Networking, 2017-18

High Performance Computing

First Resit - January 17th, 2018

Write your name, surname, student identification number (numero di matricola), e-mail. The answers can be written in English or in Italian. Please, present the work in a legible and readable form. All the answers must be properly and clearly explained, focusing on the requested issues.

1. A system $\Sigma = \{P_1, P_2\}$ is composed of two processes whose executable version is the following:

   $P_1:: \{ x = F(y); \text{notify}(E V_1); \text{wait}(E V_2); z = H(x, y); \}$

   $P_2:: \{ \text{wait}(E V_1); y = G(x); \text{notify}(E V_2); \}$

   where $x$, $y$ and $z$ are 32-bit integer variables with $y$ statically initialized to 5, and $E V_1$ and $E V_2$ are two Boolean flags. The three variables and the two flags are allocated in separated cache blocks whose home node is the PE of the first process. Assume that the two event flags are initialized to $false$ and the synchronization primitives have the following basic implementation:

   $\text{notify}(E V):: \{ EV = true; \}$

   $\text{wait}(E V):: \{ \text{while } EV == false \text{ do skip; } \}$

   The computation is executed on a parallel machine having the following characteristics:
   - single-CMP architecture with wormhole-based flow control and double-buffering firmware interfaces. The internal interconnect is a full crossbar with 4 PEs and 4 MINFs. External memory composed of four mutually interleaved macro-modules with $T_M = 40r$, each connected to a MINF;
   - D-RISC processors with service time per instruction equal to $T_{inst} = 3r$, on-demand inclusive 2-level cache per PE. Directory-based cache coherence with basic invalidation semantics.

   Each of the three functions $F, G, H$ consists in 20 D-RISC assembler instructions, and the mapping of the two processes in the CMP is chosen randomly.

   a) Describe whether and where memory ordering issues arise in $\Sigma$ and how they are solved;

   b) Provide a sufficiently accurate estimation for the completion time of $\Sigma$.

2. A process $P$ encapsulates a statically initialized matrix $B$ of size $M^2$ integers and receives a stream of input matrices of the same size from a LC input channel. The computation performed is the following:

   $P:: \{ \text{while true do }$

   $\text{receive(ch\_in, A); }$

   $\text{for } i = 0 \text{ to } M-1 \text{ do }$

   $\text{for } j = 0 \text{ to } M-1 \text{ do }$

   $\text{C}[i, j] = F(A[i, j], B[j, i]);$

   $\text{send(ch\_out, C); }$

   $\}$

   Answer to the following points by providing detailed explanations and justifications for your choices:

   a) Show a possible definition of Virtual Processors for the given problem. Explain whether the chosen VPs leads to a map or a stencil parallelization;

   b) Discuss at least three possible choices to map Virtual Processors onto actual workers by studying which kind of distribution of the input matrices and of the internal matrix are needed and which is the most convenient memory layout of the data structures in each case.

3. The problem of cache coherence in multiprocessor systems is traditionally solved for low-parallelism machines through the implementation of snoopy-based protocols.

   a) Discuss the general idea of snoopy-based solutions, which are the necessary architectural supports and which are the most critical scalability issues;

   b) A Processing Element $PE_i$ has in its cache (suppose only one level of cache) a block $b$. Show the state transition diagram of the MESI protocol for a remote access (for a LOAD or a STORE) generated by $PE_{j=m}$ and snooped through the snoopy bus.
**Solution**

1. In machines where the execution of STORE instructions can be re-ordered, the program must be carefully implemented in order to guarantee that when the event notification is made visible to the destination process, all the STORE instructions on the shared variables modified before this notification (from the viewpoint of the notifying process) are made visible to all the processors. To this end, some STORE instructions must have a *synchronous* semantics, i.e. the STORE execution is locally complete when the state related to that block in the directory (GSK) in the home node is updated and an explicit acknowledgement firmware message is received. In this way, after the completion of a synchronous STORE, the calling processor is sure that the effect of such modification on a shared variable is made visible to all the PEs in the system (eventually, if multiple copies of the modified block exist in other caches, all of them have been invalidated and the invalidations are complete).

In machines implementing the Sequential Consistency memory model, all the STORE instructions have this implicit behavior. This is practically true also in machines with the TSO memory ordering model. For other more relaxed models, explicit synchronous STOREs or equivalent memory FENCE instructions must be properly added by the system programmer. In the program, all the STORE instructions on the blocks of the variables $x$ and $y$ executed by the two processes must have the synchronous semantics.

Initially, the cache blocks are in memory and not present in any cache. To evaluate the completion time of the program, we have to consider the following temporal diagram:

![Temporal Diagram]

The execution time of the function $F$ is the following:

$$T_F \approx L_{\text{read} - \text{c1}}(\sigma_1) + 20 \cdot T_{\text{instr}} + L_{\text{c2} - \text{c1}}(\sigma_1) = 84\tau + 60\tau + 10\tau = 154\tau$$

The cache line of the variable $y$ must be transferred from the main memory into the C2/C1 of the first process, which is also the home node of that cache block. The latency $L_{\text{c2} - \text{c1}}(\sigma_1)$ is the one of the STORE *write_back* with synchronous semantics on the variable $x$.

Then, the first process executes a notification that consists in a STORE instruction on the block of $EV_1$ in order to set the flag to *true*. We observe that, overlapped with the execution of the function $F$, the second process starts the execution of the *wait* primitive by loading from the main memory the cache line of the $EV_1$ flag and doing a busy waiting on it. The initial latency of the *wait* primitive executed by $P_2$ (the latency to transfer the $EV_1$ flag from the main memory under the control of the home node) is of few tens of clock cycles, which is likely overlapped with the latency spent by $P_1$ to execute the function $F$. Therefore, the notification by $P_1$ consists in a STORE that generates an invalidation to the C2/C1 of $P_2$. Although this STORE does not necessarily have a synchronous semantics, the latency of the invalidation must be paid by $P_2$ in order to “see” the modification of the event block:

$$T_{\text{notify-p1}} \approx L_{\text{inv}}(\sigma_1) = 14\tau$$

After the notification, and to complete the *wait* primitive, the second process loads again the block of the first event from the C2 of $P_1$ and checks its value (that is now *true*). Focusing only on the cache-to-cache latencies, we have:

$$T_{\text{wait-p2}} \approx L_{\text{c2} \text{c}}(\sigma_1) = 20\tau$$

Then, the second process starts the execution of the $G$ function whose latency can be approximated as follows:
\[ T_G \approx L_{c2c}(\sigma_1) + 20 \cdot T_{instr} + L_{store}(\sigma_1) = 20\tau + 60\tau + 22\tau = 102\tau \]

The latency \( L_{store}(\sigma_1) \) is the one of the \textit{STORE} write \_back with \textit{synchronous} semantics to update the variable \( y \). Analogously, also in this case the initial latency of the \textit{wait} primitive performed by the first process (which consists in the transferring of the \textit{EV2} flag from the main memory) is likely to be overlapped with the latency \( T_G \). The latency of the notify performed by \( P_2 \) consists in the \textit{STORE} with \textit{write \_back} on the block of the second event. This instruction is not necessarily synchronous, however also in this case the latency must be paid in order to make the modification of the event visible to the first process:

\[ T_{notify\_p2} \approx L_{store}(\sigma_1) = 22\tau \]

Therefore, the non-overlapped latency spent in the \textit{wait} by \( P_1 \) is the one to load the block of the second event after the notification performed by \( P_2 \):

\[ T_{wait\_p1} \approx L_{c2c}(\sigma_1) = 20\tau \]

Finally, the first process executes the function \( H \) whose latency is:

\[ T_H \approx L_{c2c}(\sigma_1) + 20 \cdot T_{instr} + L_{write\_asynch}(\sigma_1) = 20\tau + 60\tau + 64\tau = 144\tau \]

We can observe that the cache line of \( x \) is already in C2/C1 of the first process while the block of \( y \) must be transferred from the C2 of the second process. The latency of \( L_{write\_asynch}(\sigma_1) \) is the one for the \textit{write \_back} \textit{STORE} on the variable \( z \). Although this is \textit{asynchronous} \textit{STORE}, the latency from the C1 of the first process up to the main memory should be included to have a precise evaluation of the completion time.

In conclusion, the completion time can be estimated as follows:

\[ T_C \approx T_F + T_{notify\_p1} + T_{wait\_p2} + T_G + T_{notify\_p2} + T_{wait\_p1} + T_H \]

Which is:

\[ T_C \approx 154\tau + 14\tau + 20\tau + 102\tau + 22\tau + 20\tau + 144\tau \leq 500\tau \]

2. By observing the iteration of the innermost for loop, it is easy to note that all the iterations are independent. So, the idea is to have \( M^2 \) virtual processors each one executing one statement to write the final value of the corresponding element of the output matrix \( C \). More formally, the virtual processors are associated with the following elements of the input matrix and of the internal matrix:

\[ VP_{ij} = \{ C[i,j], A[i,j], B[j,i] \} \]

and the completion time can be theoretically reduced from \( O(M^2) \) to \( O(1) \). With the given definition, the owner compute rule is respected (only one VP writes on an element of \( C \)), and each VP does not need to read elements outside the ones associated with it, i.e. the recognized form of data parallelism is a \textit{map}.

Virtual processors can be mapped onto workers according to at least three different strategies:

- \textit{row-wise mapping}: each worker executes \( M^2/n \) virtual processors, the ones that allows the process to write \( M/n \) contiguous rows of the output matrix. According with this policy, the internal matrix is statically partitioned by columns to the workers (each worker has \( M/n \) contiguous columns of the matrix), while each input matrix is dynamically scattered by rows to the workers (each worker receives \( M/n \) contiguous rows of the input matrix). Accordingly, the input target variables of the workers must be allocated in memory by rows (\textit{row-major layout}) while the internal matrix partitions are allocated in main memory by columns (\textit{column-major layout}). The partitions of the output matrix \( C \) in the workers are stored in memory by rows (\textit{row-major layout});

- \textit{column-wise mapping}: each worker executes \( M^2/n \) virtual processors, the ones that allows the process to write \( M/n \) contiguous columns of the output matrix. According with this policy, the internal matrix is statically partitioned by rows to the workers (each worker has \( M/n \) rows of the matrix), while each input matrix is dynamically scattered by columns to the workers (each worker receives \( M/n \) contiguous columns of the input matrix). According to the original sequential algorithm provided, it is still necessary to allocate the input target variables of the workers by
rows (row-major layout) and the internal matrix partitions by columns (column-major layout). The partitions of the output matrix $C$ are stored in memory by rows (row-major layout). A different memory layout is more convenient when the number of columns of $A$ and $C$ and of rows for $B$ assigned per worker is relatively small (with high parallelism degrees). In that case to exploit in a better way spatial locality the partitions of $A$ and $C$ can be allocated by columns and the ones of $B$ by rows. However, this requires to change the original code by computing in the innermost loop the statement $C[j, i] = F(A[j, i], B[i, j])$ instead of the original one;

- block-wise mapping: each worker executes $M^2/n$ virtual processors, the ones that allows the process to write a square block of $M^2/n$ elements of the output matrix. According with this policy, the internal matrix is statically partitioned by blocks to the workers, while each input matrix is dynamically scattered by blocks to the workers. The blocks of the input target variables and of the output matrix are allocated in memory by rows (row-major layout) while the blocks of the input matrix by columns (column-major layout). To use this solution the sequential for loop executed by each worker must be modified in order to express a block-based computation (write the loops as an exercise).

In all cases, the target variables of the output channels have the same memory layout of the partitions of the result matrix $C$ in the workers.

3. Snoopy-based cache coherence architectures are explained in the block of slides titled “Cache Coherence” which is available in the official home page of the course. The part related to Snoopy-based Architectures is explained in the slides 27-38 and the state transition diagram of the question is shown at slide 37.