Master Degree Program in Computer Science and Networking, 2014-15

High Performance Computing

1st appello – January 13, 2015

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.

1) Consider the following module operating on streams:

```
Q::   int A[M], B[M]; int x; channel in input_stream (1); channel out output_stream; < init A >;
      while (true) do
            { receive (input_stream, B);
              ∀ i = 0 .. M – 1:   A[i] = F(A[i], B[i]);
              x = 0; ∀ i = 0 .. M – 1:   x = x + A[i];
              send (output_stream, x);
            }
```

The input stream is generated by a module P, and the output stream is absorbed by a module R.

a) Assuming that Q is a bottleneck, define all the feasible parallelizations and their implementation schemes, including (but not limited to) the exploitation of memory hierarchies.

b) Let n_0 be the optimal parallelism degree of Q evaluated on the basis of sequential Q. Assuming that the number of processing nodes is sufficient for any parallel version, explain all the possible reasons for which the effective parallelism degree may be different from n_0, and, for each reason, the effects on bandwidth and efficiency.

c) Explain whether the following statement is true or false or true under certain conditions: "the transient interarrival time of Q is the same for all versions with sequential Q and with parallel Q".

2) Define and explain all the streamed firmware messages which are generated during the execution of a *send* primitive on an exclusive-mapping multiprocessor architecture. The interprocess communication run-time support is Rdy-Ack with shared memory synchronization and home flushing.

3) Consider the following executable version of two processes:

P :: *while* (true) *do* { S = F(S, x); *notify(updated_S)*; < local calculations >; *wait(updated_x)* }

Q :: *while* (true) *do* { *wait(updated_S)*; x = G(S, x); *notify(updated_x)*; < local calculations > }

where *S* is an array of *M* = 512K integers and *x* is an integer. *S* and *x* are *shared*. *wait* and *notify* are implemented according to an I/O-based approach. The referred events denote the updating of *S* and *x*.

The computation {P, Q} is executed on a multiprocessor architecture with the following characteristics:

i) single-CMP architecture with a wormhole-based double-buffering toroidal 4-ary 2-cube internal interconnect, and 4 MINFs. Each PE has a local I/O communication unit;

ii) the CPU is D-RISC with service time per instruction equal to 2τ, on-demand inclusive 2-level cache, primary data cache with capacity 32K words and 8-word blocks, secondary cache with capacity 1M words. The main memory has capacity 4Tera words;

iii) directory-based, invalidation-based cache coherence support with basic invalidation semantics.

P and Q are mapped onto two PEs chosen randomly.

a) Explain the behavior of P and Q in terms of processor synchronization, cache management of all used data structures, and cache coherence operations.

b) Compile P and evaluate its base latencies for *wait*, *notify* (whole latency), and for *S = F(S, x)*. *F* is available as a procedure executing *5M* instructions.

Solution outline

to be properly expanded when References are made

1)

a) Q is a module executing a *map-reduce* computation. Q has an internal state (array A) modified for each input stream value, thus it cannot be parallelized as a farm.

Data parallelism can be applied with several implementations of *map-reduce*, with independent workers (*ref: 7.8, 7.10.4*) or with loop-unfolding pipeline (*ref: 5.3, 7.9*). In the latter case, *scatter* and *reduce* are naturally distributed in the pipeline stages with low complexity, at the expense of a greater latency.

In the most general case of independent workers, the *asynchronous* reduce can be implemented according to a centralized or to a worker-distributed tree-structured approach (the *reduce* implementation with *n-1* additional modules is rarely acceptable): discuss this issue as in *7.10.4.a*.

The effect of memory hierarchies exploitation is discussed in *b-i)*.

b) Possible reasons are introduced in *4.5.2* and expanded in several Sections of the textbook.

i) Different exploitation of memory hierarchies. Array A is characterized by locality and reuse, and B by locality only. According to the *M* value, A reuse might not be exploitable in caches, or it can be exploited in secondary cache only, or in both cache levels. Since A is partitioned, if A reuse cannot be exploited in cache for sequential Q, it is possible that the parallel Q is able to exploit reuse in C1 and/or C2: the effect is a parallelism degree lower than n_0 with the same bandwidth and better efficiency.

ii) Scatter of B may be a bottleneck (*7.10.1*). The effective *n* is lower than n_0, but the ideal bandwidth cannot be achieved; efficiency is equal to one. A good reuse of A is beneficial since *n* is lower.

iii) In principle, the distributed *reduce* implementation might increase T_{calc} remarkably, so *n* must be increased in order to achieve the maximum bandwidth (*7.10.4.a*). In our case, this situation is not likely, since *F* cannot be very fine grained otherwise the module cannot be parallelized, thus:

$$n_0 \sim M \frac{T_F}{T_A}$$

If the tree-structured *reduce* is realized onto the same *map* workers, *in principle* the service time is penalized by the entire latency of the parallel *reduce* phase (following the initial sequential *reduce* executed on worker partitions), since the tree-root worker becomes the bottleneck. However, for this computation

$$T_s^{(n)} = T_A + (T_+ + T_{send}(1)) \lg n \sim T_A$$

because of the very fine granularity of addition operations and of 1-word communications, so *reduce* doesn't affect performance appreciably.

iv) Because of shared memory contention, notably on secondary caches for shared blocks, it is possible that R_Q/R_{Q0} is remarkably > 1 in the parallel version (*21.6, 21.7, 21.8, 24*). This depends on the values of *p, Tp,* R_{Q0}*, Ts* in the parallel version. The effect is that both T_{calc} and T_A increase, so $n < n_0$, and the bandwidth and efficiency are not optimal. Notice that

- with some interconnection networks, R_{Q0} itself depends on the parallelism degree and mapping of the program, e.g. with fat-trees and meshes,
- the value of *p* could be relatively large *(O(lg n)* because of the distributed *reduce i*mplementation; in these cases *F* must be remarkably coarse grained in order to have T_p large enough for achieving R_Q/R_{Q0} close to *1*.

v) Communication parameters T_{setup} and T_{trasm} are not constant, i.e. in general they are different for distinct modules and for different program versions, because of computational properties of the modules (notably, additional functionalities and data structures are present in the executable parallel version with respect to the sequential one) and for the *i)* and *iv)* reasons themselves.

c) In general the statement is false. $T_{A\text{-}Q}$ is the ideal service time of P, which may be affected indirectly by the parallel implementation of Q because of the reason *b-iv)*: if $R_Q/R_{Q0} > 1$ $T_{A\text{-}Q}$ is higher than the value for sequential Q. The statement is true only if $R_Q/R_{Q0} = 1$.

2) All streamed firmware message types in the *send* run-time support are cache block reading or write-flushing generated by the cache coherent interpreter of LOAD and STORE instructions contained in the various phases: Ack wait and reset, Rdy notify, message copy into target variable (VTG). Ack, Rdy and VTG_value are fields of VTG_S shared structures.

Block reading implies a request message and a reply message containing the block value. Block flushing implies a write request message, containing the block value, and, being this operation synchronous, a done-reply message.

Block reading and flushing are executed via C2C in C2 and, possibly, in C1 too.

Let us consider the situation in which the sender is the *non-home* node of VTG_S. The generated message types, with destination the home node, are: reading the VTG_S block containing Ack via C2C, and write-flushing of a VTG_S block via C2C. Such blocks are locally invalidated by the sender itself when the synchronous flush is completed.

If the sender is the *home* node of VTG_S, it operates in the local cache only: reads the VTG_S block containing Rdy, and modify all the VTG_S blocks. The non-home receiver reads the VTG_S blocks from the home sender cache via C2C, flushes the modified Rdy-Ack block, and invalidates locally the modified blocks.

3)

a) Structure *S* (*x*) is used in a read-write mode by P (Q) and in a read-only mode by Q (P). The best homing strategy for the shared variables is: P is home node of *S* and Q is home node of *x*.

S is reused by P, and reuse can be exploited in secondary cache C2; $2M/\sigma_I$ faults are generated in C1, however only M/σ_I have effect on performance, since write operations don't cause block transfer. *x* is reused by Q, and reuse can be exploited in C1 (1 block), thus no faults are generated.

Only for the first iteration (vs an infinite loop), *S* (*x*) is transferred into C2 of P (Q) from the main memory.

P (Q) reads *x* (*S*) via C2C from Q (P), executes F (G) on local *S* (*x*) modifying *S* (*x*) and invalidating the *S* (*x*) copy in Q (P). Invalidations are *synchronous*.

The notify synchronization is *asynchronous*: *at most one* notification can be received by the partner before the *wait* execution. Thus, a buffer of one position is sufficient to register the event.

b) P compilation:

```
START:      // S = F(S, x) //
        LOAD   Rx, 0, Rx-param      // via C2C //
            // parameter S is passed by reference via the same register RS for both procedure and
            main; S blocks modifications, done by the procedure, cause invalidations in Q //
        CALL   RF, Ret
            // notify (updated_S); only the event identifier needs to the communicated; the streamed
            firmware message sent by UC consists of the header only //
        STORE  RUC, 0, Q-PE_id      // explain clearly the Memory Mapped I/O utilization //
        STORE  RUC, 1, Revent
        < local calculations >;
```

// *wait(updated_x);* the presence of a buffered notification is signaled by ready-event //

wait (event)::

if (*not* ready-event)

then WAITINT mask, ev, -, - *else* ready-event = false

//

```
        LOAD   Revent-buffer, 0, Rready-event
        IF = 0   Rready-event, THEN
        STORE   Revent-buffer, 0, 0
        GOTO  START
THEN:   WAITINT   Rmask, ev, -, -
        GOTO   START
```

The *interrupt handler* provides to put ready-event at *true*:

// interrupt interface routine //

```
        LOAD   Rinterrupt_table, Revent, Rhandler
        CALL   Rhandler, Rret_int
```

// handler //

```
        STORE   Revent-buffer, 0, 1
        GOTO   Rret_int
```

Notify latency

This latency is evaluated for the code instructions, the streamed firmware message communication through UC-P, network, UC-Q, and interrupt handling (if executed).

Latency of two STORE instructions: 4τ.

Latency of firmware message communication: the path is UC-P, W-P, internal net, W-Q, UC-Q.

The internal net has a 2D-mesh topology with 16 switching nodes, of which 4 are connected to MINFs and 12 to PEs (three 4-node rings). For uniformly random mapping of P and Q, the average distance can be evaluated as follows:

- P, Q in the same ring, with probability 1/3: distance 1 with probability 2/3 and distance 2 with probability 1/3; on the average 4/3,
- P, Q in the neighbor rings, with probability 1/3: 1 + 4/3 = 7/3,
- P, Q in the most distant rings, with probability 1/3: 2 + 4/3 = 10/3.

Thus: d_{net} = 2.33, and the whole path distance is:

$$d = 4 + d_{net} = 6.33$$

The firmware message latency, with double buffering, message length s = 1 and $T_{hop} = \tau$, is:

$$L_{fw-msg} = (s + d - 2)\, T_{hop} = 5.33\, \tau$$

The interrupt handling has latency 8τ (4 instructions).

In conclusion, in the worst case of executed interrupt handler:

$$L_{notify} = 17.33\, \tau$$

Wait latency

The *wait* code executes 4 instructions, thus:

$$L_{wait} = 8\,\tau$$

S = F(S, x) latency

The pure code consists of 2 instructions (LOAD, CALL) and the *F* procedure, with latency

$$\beta \sim 10\,M\,\tau$$

x-block C2C reading operation:

- the streamed message length for request is $s_{req} = 3$, for reply (pure synchronization) $s_{reply} = 9$;
- the same path length as before, $d = 6.33$, thus:

$$L_{C2C}(\sigma_1) = 16.33\,\tau$$

The C2-C1 block transfers for *S* have a whole latency:

$$L_{C2-C1}(M, \sigma_1) = \frac{M}{\sigma_1}\,(\sigma_1 + 2)\,\tau = 1.25\,M\,\tau$$

Invalidation of M/σ_1 *S* blocks in Q:

- the streamed message length for request is $s_{req} = 3$, for reply (pure synchronization) $s_{reply} = 1$;
- the same path length as before, $d = 6.33$, thus:

$$L_{inv}(M) = \frac{M}{\sigma_1} 8.33\,\tau \sim M\,\tau$$

In conclusion:

$$L_S = \beta + L_{C2C}(\sigma_1) + L_{C2-C1}(M, \sigma_1) + L_{inv}(M) \sim 12.25\,M\,\tau$$