Master Degree Program in Computer Science and Networking

High Performance Computing

First appello – January 18, 2013

The answers can be written in English or in Italian.

All the answers must be properly and clearly explained, and presented in a legible and readable form.

Question 1

A module Q operates on integer streams (input *x*, output *y*), encapsulates an integer array *A[M]* with $M = 10^4$, and is defined as follows:

int A[M]; int x, y;
$\forall$ *x :*
{$\forall$ *i = 0 .. M – 1 :*
A[i] = F(x, A[i]);
$y = \sum_{i=0}^{M-1} A[i]$
}

Q is executed on a D-RISC architecture with scalar pipelined CPU, data cache capacity of 32K words and block size of 8 words. The processing time of function *F* on this architecture is equal to $10^4\tau$ in absence of cache faults.

The interarrival time is equal to $10^3 M\tau$.

a) Define a parallel implementation of Q to be executed on a parallel architecture with $N = 32$ PEs based on the CPU described above, $T_{setup} = 1000\ \tau$, $T_{trasm} = 25\ \tau$, exclusive process mapping, zero-copy communication support, and communication processor.

b) Evaluate the service time and the relative efficiency of the whole parallel program and of each component module.

c) Explain if the executable file of Q can be reused in the parallel version: discuss this issue by explicitly analyzing the various parts composing the executable file (and discuss also the reusability of the configuration file).

Question 2

Verify if the assumed values of T_{setup} and T_{trasm} are well approximated for the execution of the parallel program of Question 1 on the following architecture:

i) all-cache SMP multiprocessor with shared main memory;

ii) 2-ary 5-fly wormhole interconnection network, with 1-word links and flits, single buffering interfaces, and link transmission latency equal to 4τ;

iii) interleaved memory macro-modules, each one with 8 modules and clock cycle equal to 30τ;

iv) CPU defined in Question 1;

v) periodic retry locking;

vi) automatic directory-based cache coherence with invalidation.

Note: students can request to shortly look up the Course Notes copy on the teacher's desk for formulas, numeric values and figures related to performance metrics.

Solution

to be integrated with proper explanations

Question 1

a) In order to define the parallel implementation, we need to know T_{calc}, i.e. the ideal service time of Q, which corresponds to the completion time of the sequential module for a generic stream element.

The first loop is clearly dominated by the processing time of F (T_F), thus the first part has a completion time

$$\sim M\, T_F = 10^4\, M\, \tau$$

The completion time of the second part, i.e. *reduce (A, +)*, is negligible: for a CPU with time slot $t = 2\ \tau$, (*evaluate the completion time for the given CPU*)

$$T_{reduce} = 10\, M\, \tau$$

The communication latency, $T_{send}(1) \sim T_{setup}$, is negligible as well.

The penalty due to cache faults is negligible, since A is *reused* for all the stream elements and $M <$ data cache capacity, thus A can be maintained in cache after the first stream element. The only cache faults occur in the execution of *send* and *receive* primitives, thus their effect has been already evaluated in the given values of T_{setup} and T_{trasm}.

In conclusion:

$$T_{calc} = 10^4\, M\, \tau$$

The optimal parallelism degree is:

$$n = \frac{T_{calc}}{T_A} = 10$$

The parallelization cannot be done by the farm paradigm, since the module has an internal state (A itself) which is modified for each stream element.

The solution is a data-parallel implementation *map-reduce*, with A statically partitioned and x multicasted. Because of the negligible service time and latency, the *reduce* function is implemented by a single module OUT operating sequentially on the results of the n local *reduce* executed by the workers.

The multicast can be implemented sequentially by a single module IN, which is not a bottleneck:

$$T_{multicast} = n\, T_{send}(1) \sim 10^4\ \tau \ll T_A$$

b) Because there are no bottlenecks in the multicast-map-reduce graph, the performance metrics of the whole parallel program are

$$T^{(10)} = T_{id}^{(10)} = T_A \qquad\qquad \varepsilon^{(10)} = 1$$

For the single modules:

	Service time	Efficiency
Input module IN (multicast):	$T_{multicast}$	$\rho_{IN} = \frac{T_{multicast}}{T_A} = \frac{10}{M} \sim 0$
Worker:	T_A	$\rho_W = 1$
Output module OUT (final reduce)	$\sim T_{send}(1)$	$\rho_{OUT} = \frac{1}{M} \sim 0$

c) The executable file of process Q contains the following information defining Q's virtual memory:

1. code of sequential program;
2. variables *A*, *x*, *y*, and target variable of OUT;
3. input and output channel descriptors;
4. PCB and other data structures of run-time support, including relocation table;
5. code of *send*, *receive* and other functionalities of run-time support, including low-level scheduling and interrupt/exception handlers.

Let's try to reuse such information for the generic worker. Modules IN and OUT must be produced for the specific parallel program.

1, 2 are reusable with no problem.

3 is reusable provided that channels are referred by channelname variables, which must be properly assigned for each worker.

4 is reusable provided that PCBs of other processes are allocated dynamically in the virtual memory (capability based addressing).

5: *send*, *receive* and *handlers* are reusable (notice that even for the sequential module the run-time support exploits locking, since Q belongs to a parallel program operating on streams), while the low-level scheduling has to be replaced by the exclusive mapping version if the original version uses a multiprogrammed approach.

The configuration file, associated to the executable file, must be re-generated, since the partner processes are different.

Question 2

Under assumptions *iv)*, *v)*, *vi)*, the cost model for interprocess communication is evaluated as (Part 2, Sections 7.3, 7.4)

$$T_{setup} \sim 5\Omega \qquad\qquad T_{transm} \sim \frac{\Omega}{\sigma}$$

where Ω is the under-load memory access latency for cache block transfer. *Insert explanations on how these values are determined in relation with the problem specification*.

Let us first evaluate the base memory access latency.

A memory read request communication uses a firmware message, of length m = 2 words (header, block physical base address), from a PE to a shared memory macromodule. The distance d is constant and equal to 9: $d_{net} = n = 5$ hops for the network, plus 4 hops including: source data cache, source MINF (external memory interface), source W (PE interface unit), and destination I_M (memory interface unit).

Each hop has latency $t_{hop} = \tau + T_{tr} = 5\tau$.

The base latency for a read request communication is given by:

$$\Omega_{0-req} = (2m + d - 3)\, t_{hop} = 50\,\tau$$

The base latency of a cache block transfer is given by:

$$\Omega_0 = \Omega_{0-req} + \tau_M + \Omega_{0-reply} = 210\,\tau$$

since the reply firmware message has length $m = \sigma + 2$ = words.

At this point, we evaluate the under-load memory latency as a function

$$\Omega = \Omega(\Omega_0, p, T_p)$$

In this case, the mean time between two consecutive shared memory accesses is approximated by:

$$T_p = T_{calc} = 10^4 \, M \, \tau$$

thus we are in presence of a very coarse grain computation.

For the SMP architecture, according to the interleaved memory properties, the average number of processing nodes in conflict for the same memory macromodule is given by:

$$p = \frac{N}{B_M}$$

where B_M is the effective bandwidth of the interleaved memory. With 32 PE and 32 memory macromodules, we have $B_M \sim 20$ (*Part 2, Section 2.4.2, page 33*), thus:

$$p \sim 2$$

With these values of p and T_p, we derive (*Part 2, Section 4.3, page 73*):

$$\Omega \sim \Omega_0 = 210 \, \tau$$

i.e., the base latency is a very good approximation of the under-load latency.

At this point, we able to estimate the communication metrics:

$$T_{setup} \sim 1050 \, \tau \qquad\qquad T_{transm} \sim 26 \, \tau$$

and we verify that the assumptions are sufficiently well approximated (5% error).

Additional comments

1) Though being able to achieve the result $\Omega \sim \Omega_0$ is important, in general this might not be possible for some parallelization problems and/or architectures. The value of Ω must be always evaluated explicitly, by estimating p and T_p and by using the function $\Omega = \Omega(\Omega_0, p, T_p)$ curves in Part 2, Section 4.3, page 73.

If necessary, the bandwidth of an interleaved memory is determined using the curves of Part 2, Section 2.4.2, page 33.

2) T_{calc} must be evaluated explicitly for the given CPU. It is not acceptable to say "supposing that the reduce processing time is negligible" or "an integer addition costs one clock cycle" (which is wrong), nor to assume that the processing time of a loop always coincides with the body processing time. In some cases an approximation could be sufficient, but always based on the effective assembler code evaluation. Of course, the analytical cost model for the pipeline CPU makes the evaluation must faster compared to the graphical simulation.

3) The presence of primary and secondary cache must be taken into account explicitly in T_{calc} evaluation, analyzing the locality and reuse properties of the given computation and referring to the specified cache architecture (on demand, prefetching, writing techniques, and so on). If the fault probability is not negligible, the block transfer latency must be evaluated for the given *parallel* architecture: in particular, the block transfer latency from the shared memory to the highest cache level (e.g. secondary or primary, depending on the problem specifications) is given by Ω.

4) The expressions (as a function of Ω) for T_{setup} and T_{transf} used in Question 2 are not general: they are valid for the assumptions of Part 2, Sections 7.3, 7.4. In general, proper expressions must be derived for different assumptions about architecture, locking technique, run-time support algorithm and cache coherence technique.

5) Remember that, in general, the solution to the cost model of a parallel program on a given architecture implies an iterative procedure: the parallelism degree n is a function of T_{calc} and T_{send}, which in turn are functions of Ω, which in turn is a function of n itself. In several cases our methodology simplifies the problem, notably if the situation $\Omega \sim \Omega_0$ is largely independent of n. But this is not true in general.

6) Collective communications or functions (scatter, gather, multicast, reduce) must be implemented, for the actual parallel version, in the most effective way. For example, it is useless to implement a reduce or a multicast by a tree structure if just the sequential versions achieve the best performance. This issue has to be evaluated accurately and clearly explained. Notice that the actual version of a data-parallel program not necessarily uses the same structures of the virtual processors version (e.g., in Question 1 the virtual process version uses a tree structured reduce, while the actual version uses the sequential implementation).

7) Finally, the background concepts and techniques of Part 0 are necessarily intermixed and exploited in the questions related to Part 1 and Part 2. Although the exam syllabus is on Parts 1 and 2, it is not acceptable that issues related to Part 0 are ignored or skipped or oversimplified.