Overlapping of Computation and Communication in the Quadrics Network

Eitan Frachtenberg and Fabrizio Petrini
Technical report

July 31, 2001

Preamble

We intend to design and implement a scheduler testbed on a Quadrics cluster for researching issues in job scheduling and fault tolerance. The Quadrics interconnect is a powerful network that offers low-latency, high-bandwidth user level messages with adequate processing power in the NICs. To make use of the power of this network, we will test schedulers such as Buffered Coscheduling (BCS), where the communication of jobs is buffered and performed at a later time, in parallel with the computation of this job or another. This scheduler requires the ability of the network to overlap running computation jobs with a communication-scheduler process that executes the buffered communication. In this report, we analyze the ability of the Quadrics network to overlap communication and computation test applications on a small Linux cluster. We test various network and scheduling parameters to find the optimal setup for such overlapping.

Test applications

We use two micro-benchmarks and measure their run time in seconds.

1. A CPU-intensive program (called *"burn_cycles"*), looping 5×10^8 times over the C expression 'x+=sin(x)'. It performs no communication and runs on two nodes, two PEs each.
2. A communication-only application: we use Quadrics' dping(1) with different parameters for message size, number of messages, and the message-waiting mechanism: polling or event- (interrupt) driven.

Base run times

The burn_cycles application runs for approximately 63.1 seconds when run alone and uninterrupted (except for the usual UNIX daemons). When dping

command line (no. iterations and message size)	run time
`dping -n 1 1m 1m`	2.4
`dping -n 1 64m 64m`	8
`dping -n 6400 1m 1m`	41.7
`dping -n 100 64m 64m`	46.5

Table 1: Base run times for dping alone with event-driven wait

command line	run time
`dping -n 1 1m 1m`	2.4/63.2
`dping -n 1 64m 64m`	8.9/64.1
`dping -n 6400 1m 1m`	42/63.3
`dping -n 100 64m 64m`	47/64.1

Table 2: Base run times for both applications running in parallel (event-driven wait). Run times are shown for dping first and burn_cycles second.

is running alone with the event wait, we get the run times shown in Table 1. Note that times are approximate and averaged over three runs. Also note that in this experiment and the next ones, the 2nd case (`dping -n 1 64m 64m`) demonstrates a high variance, and since we only average three measurements, this number is not very stable.

The `-n` parameter of dping controls how many iterations of the ping loop the program performs. Obviously, a higher number of iterations decreases the relative overhead and therefore the increase in run time is not proportional to the number of loops. One interesting result is that sending 6400 one-megabyte messages takes less time than sending 100 64-megabyte messages. This is probably due to paging issues affecting the larger message buffer, and could also be related to the relative overhead effect.

Overlapping

We run both types of applications in parallel using the following method: A shell script launches burn_cycles twice in each node using rsh(1), and then launches the dping program using prun(1). The starting time of all processes is thus almost the same, with a span of less than 0.5 sec. Note however that the finishing time is never the same, and in most cases the dping will terminate before the burn_cycles program. Therefore, we can only refer to changes in run time of the computation application in absolute difference and not in percentages. Table 2 shows the run times for both applications when run in parallel (with the default base priority.)

It is interesting to note that run times are hardly affected by the fact the programs are running together: dping has a slowdown of less than 1% and burn_cycles' slowdown is not too significant as well.

command line	run time (priority -20)	run time (priority 20)
`dping -n 1 1m 1m`	60/63.3	2.4/63.3
`dping -n 1 64m 64m`	69/63.7	8/64.2
`dping -n 6400 1m 1m`	102/63.5	41.5/82.7
`dping -n 100 64m 64m`	107/63.7	46.6/64.6

Table 3: Run times for both applications when burn_cycles' base priority is modified. Run times are shown for dping first and burn_cycles second.

Effect of priorities

The local UNIX scheduler can have a critical role in the successful overlapping of processes that compete for different resources. To test the effect of the scheduler on this overlapping benchmark, we try to intervene with its behavior by modifying the base priorities of the processes. Two experiments were conducted: in the first we started the burn_cycles application with the highest possible user priority (-20), while in the second we assigned it the lowest base priority (20). In both cases the dping was run with the default base priority. Note that the scheduler makes its own priority decisions, like lowering those of CPU-hungry applications and raising those of processes that are waking from a blocking call. These decisions are made and taken into account within a very short time, so the effect of the initial priority adjustment is somewhat attenuated.

We can readily see that raising burn_cycles' priority hinders the scheduler's effort at overlapping. In fact, we can deduce from the run times that it effectively blocks dping from running until burn_cycles is completed - meaning the system serializes the applications. On the other hand, burn_cycles' performance is not improved by its higher priority, since its CPU requirements were mostly fulfilled even with the same base priority.

When reversing the situation, we also reverse the effect: instead of dping running slower, burn_cycles is delayed. The amount of extra time burn_cycles is delayed is related to the amount of loops dping performs, so it most noticable in the third case (6400 loops). Since dping does not actually require a lot of CPU (it has no significant speedup when assigned a higher priority), it appears that the UNIX scheduler actually allots more CPU time to dping then it actually uses, and this CPU time is wasted needlessly. This may be a 'worst-case' scenario for the Linux scheduler, where its heuristics fail to preempt dping even when it is blocking.

Effect of polling mechanism

The polling mechanism determines if a process issuing a blocking communication primitive polls the communication channel until the operation is completed (busy-wait), or blocks until an event (interrupt) wakes it up. Obviously, when the dping application runs in stand-alone mode, it has more to benefit from

command line	run time
`dping -n 1 1m 1m`	2.6/63.3
`dping -n 1 64m 64m`	8.7/65.5
`dping -n 6400 1m 1m`	83.6/90.5
`dping -n 100 64m 64m`	51.8/80.3

Table 4: Run times for both applications with polling wait. Run times are shown for dping first and burn_cycles second.

busy-waiting, because it does not potentially yield the CPU to another process, and loses some time due to the context-switch overhead. Indeed, we did observe a slight decrease in dping run times when tested with polling wait. We can similarly argue that when running the applications in parallel, utilization is increased and the computation process is disturbed less when using event-driven waiting. To test this, we run the applications in parallel using polling-waiting (by setting the LIBELAN_WAITTYPE environment variable to ELAN_POLL_EVENT, instead of ELAN_WAIT_EVENT). The run times can be seen in Table.

These results confirm our intuition and show that polling for the completion of communication can not only have an adverse effect on other applications (in this case, burn_cycles), but also on the communicating application itself. One reason for this could be that the local UNIX scheduler, trying to maintain fairness and compensate for the CPU time dping consumes, allocates some CPU time to burn_cycles even when dping is no longer blocked. This results in wasted time for dping, which could have been used to issue the communication in parallel to burn_cycles.

Conclusions

We draw the following conclusions from the experiments presented here:

- The system, comprised of the Linux scheduler and Quadrics' communication primitives, can overlap computation and communication processes rather effectively **as long as the communicating processes do not busy-wait**.
- Running a large amount of small communication operations, as opposed to a small amount of large communication operations, not only taxes the communication process, but also other computation processes in the system.
- In general, we cannot gain much improvement in the overlapping of processes by modifying their priorities. More likely, we would hinder the system's utilization.