Parallel Paradigms Run-Time Support:
Message-Passing vs Shared-Variables

Contents
1. Introduction .......................................................................................................................... 2
2. Shared-variable implementation ............................................................................................ 2
3. Farm computations .................................................................................................................. 3
   3.1. NUMA architecture ........................................................................................................ 3
   3.2. SMP architecture ........................................................................................................... 6
   3.3. Discussion and further improvement of message-passing solutions .............................. 6
4. Data-parallel computations ..................................................................................................... 8
1. Introduction

In Part 1 and Part 2 we have studied implementation models and cost models of structured parallelism forms for shared memory and distributed memory architectures, assuming that the process level adopts a message-passing model according to the pure local environment approach. That is, the parallel paradigm run-time support has been designed by means of proper collections of message-passing processes.

For shared memory architectures it is interesting to study the run-time support designed according to a shared-variable model too, just because it is rather natural for this kind architecture (at least, at a first sight).

The objectives of these notes are:

- understanding the shared-variables implementation of parallel paradigms and its cost model,
- comparing the message-passing vs the shared-variable run-time support of parallel paradigms on shared memory architectures.

The importance if this issue is related to the performance-portability trade-off. The message-passing approach is inherently portable across shared memory and distributed memory machines: the same process-level code is valid for a distributed memory machine and for a shared memory one, and even for mixed heterogeneous architectures too (e.g. clusters of multiprocessors) independently of the process allocation in the same shared memory node or in distinct distributed memory nodes. Instead, a shared-variable code is limited to shared-memory machines only (and often it is specific of a given class of machines).

Key questions are:

- is the portability advantage of the message-passing approach paid with a lower performance?
- are there meaningful cases in which the message-passing solution outperforms the shared-variable one even for shared-memory machines?
- when the shared-variable approach achieves a better performance, is the performance of the message-passing solution acceptable anyway?

As usually, in this analysis we adopt a zero-copy implementation of message-passing LC primitives with communication-calculation overlapping. This approach is confirmed by existing libraries like MPICH-PM, VMMC-2, Active Messages, U-Net and BIP-SMP.

For shared-variable mechanisms low-level mechanisms based on Posix Pthreads can be used by expert designers (let us remember that we are speaking about the run-time support design). Higher level libraries and frameworks include Open MP, TBB, Cilk, and Quark. Some of them are oriented to simple, generic data-parallel programs (“loop parallelism”). Except TBB, streams are not primitive and must be emulated through other mechanisms or libraries.

2. Shared-variable implementation

In a shared-variable model, processes cooperate by sharing data structures, i.e. objects belonging to all their logical address spaces. Shared objects are used, at the same time, for computation data and for synchronization mechanisms (locks, monitors, or their variants). As known, shared memory synchronization may be a source of serious performance degradations because of the software lockout problem.

Let us consider a farm computation. Streams are implemented as shared queues, on which independent Workers execute get and put operations atomically. No explicit Emitter and Collector
module exists. The necessary synchronization on queues (e.g., by locking) affects the software lockout degree, as well as and the $p$ parameter value (related to the evaluation of under-load memory latency) in NUMA architectures.

Notice that the described scheme does not exploit the peculiarity of the farm paradigm, instead it is merely a generic implementation of process mutual exclusion on shared objects. The software lockout problem can be eliminated/alleviated by schemes which are more specific of the farm paradigm semantics.

A notable solution consists in implementing the synchronization by passing pointers (formally, capabilities) from producers to consumers: an explicit Emitter process includes the input stream queue and transmits the pointer (capability) of a queue element to a Worker through a short message, so that the Worker can address the input data structure without copying it and without additional synchronizations. The dual solution is applied to the Worker-Collector cooperation. That is, in this shared-variable technique, messages are still used, but for passing references only, thus it is not a local environment model.

In the following, we assume a pointer-passing solution for shared-variable run-time supports.

3. Farm computations

Let us consider a purely functional node of a stream-based computation graph. Let $T_A$ be the interarrival time, $T_{\text{calc}}$ the ideal service time, and let the stream element be a $M$-word data structure (e.g. an array).

If $T_{\text{calc}} > T_A$ the node is a bottleneck, and potentially it can be parallelized with parallelism degree

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

(we avoid the symbol $\lceil \ldots \rceil$ just for writing simplicity).

The Emitter process includes a queue of input stream elements. This is exactly what happens in a zero-copy message-passing communication (the queue is the target variable queue), and, as seen in Sect. 2, this is the technique used also in a shared-variable solution based on pointer-passing.

Let $n_1$, $n_2$ the parallelism degree of the message-passing and of the shared-variable solution respectively, and $T_{\text{calc}1}$, $T_{\text{calc}2}$ their ideal service times.

The analysis will be done separately for NUMA and SMP architectures.

3.1. NUMA architecture

In this class of multiprocessor architectures, the two implementations differ in the evaluation of the memory access latency, thus in the value of $T_{\text{calc}}$. In fact, while the message-passing solution is characterized by a good low-$p$ mapping:

$$p_1 \sim 3$$

in the shared-variable solution the $p$ value is substantially greater:

$$p_2 \sim n_2 + 1$$

since all the Worker PEs access the Emitter PE local memory.

Thus:

$$T_{\text{calc}1} = T_{\text{calc}1}(t_{a0})$$
with $t_{a0} = t_{a0-local}$, and

$$T_{calc_2} = T_{calc_2}(t_a)$$

with $t_a = t_{a-remote}$. Consequently

$$T_{calc_2} > T_{calc_1}$$

The greater $T_{p2}$ value (parameter of the under-load latency evaluation) partially compensates the difference between $T_{calc_2}$ and $T_{calc_1}$. However, the ratio $T_{p2}/T_{p1}$ is not so large (not an order of magnitude), while the ratio $p_2/p_1$ is substantial and more meaningful for the under-load latency evaluation. In other words, the impact of $p$ on the memory latency is prevailing, so the under-load memory access latency $t_a$ of the shared-variable solution is sensibly greater than the base latency $t_{a0b}$, which confirms the relation between $T_{calc_1}$ and $T_{calc_2}$.

Let us distinguish between the situations in which, in the message-passing solution, the Emitter is not or it is not a bottleneck, i.e. case a) $T_A \geq L_{com}(M)$, case b) $T_A < L_{com}(M)$

\[ a) \ T_A \geq L_{com} (M) \]

In the message-passing solution the Emitter, with ideal service time

$$T_{E_1} = L_{com}(M)$$

is not a bottleneck, so the optimal parallelism degree is fully exploited:

$$n_1 = \frac{T_{calc_1}}{T_A} \quad \rightarrow \quad B(n_1) = \frac{1}{T_A}$$

For the shared-variable solution the optimal bandwidth is achieved for:

$$n_2 = \frac{T_{calc_2}}{T_A} \quad \rightarrow \quad B(n_2) = \frac{1}{T_A}$$

with

$$n_2 > n_1$$

That is, the shared-variable solution achieves the maximum bandwidth with a higher parallelism degree, as illustrated in the following figure:

In conclusion, if $T_A \geq L_{com} (M)$, the message-passing solution offers the best bandwidth for any $n$. 
b) \( T_A < L_{com}(M) \)

The Emitter is a bottleneck in a message-passing approach, while it is not a bottleneck in a shared-variable approach based on pointer-passing, for which

\[ T_{E_2} = L_{com}(1) \]

In this situation the best parallelism degree for the message-passing solution is:

\[ n_1 = \frac{T_{calc_1}}{L_{com}(M)} \]

while for the shared-variable solution it is still:

\[ n_2 = \frac{T_{calc_2}}{T_A} \]

The difference in the denominator further increases the difference between \( n_1 \) and \( n_2 \).

It is easy to derive the relation between \( n_1 \) and \( n_2 \):

\[ n_2 = \alpha n_1 + \beta \left( \frac{t_a}{t_{a0}} - 1 \right) \]

with

\[ \alpha = \frac{L_{com}(M)}{T_A} \quad \beta = \frac{T_{fault_1}}{T_A} \]

The bandwidth comparison is illustrated in the next figure, where

\[ n_0 = \frac{T_{calc_2}}{L_{com}(M)} \]

In conclusion, for NUMA architectures, if \( T_A < L_{com}(M) \) the message-passing solution bandwidth is better than the shared-variable one when the available number of PEs is

\[ N \leq n_0 \]

In this case, the actual parallelism degree is equal to

\[ \min(N, n_1) \]
For
\[ N > n_0 \]
the shared-variable solution offers the best bandwidth, and the actual parallelism degree is
\[ \min(N, n_2) \]

### 3.2. SMP architecture

In this multiprocessor architecture with good approximation \( p \) is mapping independent, thus
\[ T_{\text{calc}_1} = T_{\text{calc}_2} = T_{\text{calc}}(t_a) \]

If \( T_A \geq L_{\text{com}}(M) \), we achieve the same (very similar) bandwidth values for both solutions:
\[ n_1 = n_2 = n_{\text{opt}} = \frac{T_{\text{calc}}}{T_A} \quad \rightarrow \quad B(n_{\text{opt}}) = \frac{1}{T_A} \]

If \( T_A < L_{\text{com}}(M) \), the bandwidth comparison is shown in the next figure, where
\[ n_1 = \frac{T_{\text{calc}}}{L_{\text{com}}(M)} \quad \quad n_2 = \frac{T_{\text{calc}}}{T_A} \]

The two solutions are equivalent when the number of available PEs is
\[ N \leq n_1 \]
otherwise the shared-variable solution offers the best bandwidth.

### 3.3. Discussion and further improvement of message-passing solutions

The net result is that the shared-variable solution outperforms the message-passing one
1. only when \( T_A < L_{\text{com}}(M) \),
2. and for a number of available PEs over a given threshold (\( n_1 \) in SMP, \( n_0 \) in NUMA).

We know that condition 1 is not frequent for stream-based graph computations: it holds in rather extreme situations (interarrival rate greater than the communication bandwidth of single channels),
which are difficult or even impossible to achieve (for example, think about the network card in cluster architectures). For this reason,

- the portable message-passing solution offers the best bandwidth in the large majority of stream-based graph computations.

However a different kind of application is of some interest: the computation to be parallelized is not a node of a stream-based graph computation, instead it is a sequential program to be “accelerated” in some critical points.

In this case, a stream-equivalent computation can be defined and parallelized as a farm (the Master-Worker terminology is often used), in which the “main” program itself acts as the Master that delegates tasks to external Workers (Master as Emitter) and receives results from them (Master as Collector). In other word, in a Master-Worker approach a sequential program is accelerated similarly to what happens in the traditional cooperation mode between CPU and I/O units and/or co-processors.

Because we have no constraint in the interarrival time, better bandwidth values are achieved for a stream generation rate greater than $\frac{1}{L_{\text{com}}(M)}$.

In this case the shared-variable solution, with a relatively high parallelism degree, is able to overcome the stream-equivalent message-passing solution.

However, a further optimization can be applied in the message-passing solution: because the problem is the Emitter bandwidth, a substantial improvement is achieved by associating multiple communication processors to the Emitter PE itself, i.e. overlapping more than one communication (M-word messages) from the Emitter to the Workers.

As a counterpart, the $p$ value is increased for the message-passing solution in NUMA machines too, since multiple communication processors access the Emitter local memory, so $T_{\text{calc}}$ becomes closer to the shared-variable solution.

Notice that this optimization is applied to the Emitter PE only, not to the Worker PEs (each of which have just a single communication processor).

This solution is feasible if the firmware architecture provides a sufficient number of communication processors. For example, some recent multicore chips (especially, network processors) contain several I/O intelligent co-processors connected to all PEs and to be specialized by program.

A cheaper and more general and flexible implementation exists: provided that PEs are multithreaded, the communication processors are implemented (emulated) by means of dedicated threads executed in the same CPUs, or even in distinct CPUs.

This solution has been proposed in:


and experimented on the Intel Phi with 60 4-way-multithreaded cores, equipped with a LC-like runtime support. It is proved that the solution outperforms standard MPI solutions (which are equally portable onto distributed memory machines), and achieves similar or better performances compared to implementations based on standard shared-variable frameworks (which are not portable onto distributed memory machines), especially for fine-grained computations.

Finally, the analysis presented in these notes is simplified because the possible impact of automatic cache-coherence techniques are not taken into account. This relevant issue is part of the on-going PhD research by Silvia Lametti.
4. Data-parallel computations

The previous results can be extended to data-parallel computations operating on streams.

The most critical module in message-passing implementations of stream-based data-parallel programs is the scatter functionality:

- if it is not a bottleneck, then the message-passing solution with one communication processor/thread has no bandwidth problem,
- otherwise the shared-variable solution is the best for high parallelism degrees. Alternatively, the message-passing solution with multiple communication processors/threads can achieve comparable bandwidths.

The shared-variable scatter implementation consists in passing the pointer (capability) of the data structure to be partitioned, along with the partition index, to the Workers. Each Worker accesses its own partition independently, without additional synchronization in a map computation.

For multicast, the shared-variable solution consists merely in multicasting the pointer, so all the Workers can access the data structure independently. The message-passing solution is less critical, compared to the scatter, owing to the tree-structured implementation which often is able to overlap communication to calculations.

Similarly, stencils and reduce do not affect the message-passing performance in the majority of cases in which the stencil/reduce communications are overlapped to the internal calculations.

In the shared-variable approach, a stencil computation means that a consumer Worker reads the needed data by accessing the producer Worker memory, provided that such data are updated. This requires a synchronization, which may be implemented by locking mechanisms or by global barriers: in both cases, software lockout and high-p values affect negatively the performance, which becomes much lower than expected. If synchronization is implemented by pointer-passing, than the solution becomes more similar to the classical message-passing one. It is worth remarking that often stencils require fine-grain communications in the true message-passing model too.

Similar considerations apply to reduce (sub-)computations.