Role of Message-Passing in Performance Oriented Parallel Programming *

Vipin Kumar, George Karypis, and Ananth Grama

Abstract

The message-passing programming paradigm is often called the assembly language of parallel computers. The paradigm requires the user to partition the data among processors and specify interaction among processors explicitly. In contrast, the shared memory programming paradigm offers the user a global address space, making it unnecessary to specify data partitioning and communication. Indeed, many parallel programs written using the shared-memory paradigm are much shorter than their counterparts in the message-passing paradigm. Until recently, cache-coherent shared address space architectures were largely based on a global shared bus, making them inherently unscalable beyond a few dozen processors. More recently, scalable directory-based cache-coherent shared address space architectures are being offered by many vendors. This seems to make the message-passing programming style obsolete. In this article, we argue that the style of message-passing programming makes it much easier to develop efficient parallel programs even on cache-coherent shared address space architectures.

The message-passing programming paradigm is often called the assembly language of parallel computers. The paradigm requires the user to partition the data among processors and specify interaction among processors explicitly. In contrast, the shared memory programming paradigm offers the user a global address space, making it unnecessary to specify data partitioning and communication. Indeed, many parallel programs written using the shared-memory paradigm are much shorter than their counterparts in the message-passing paradigm.

Until recently, cache-coherent shared address space architectures were largely based on a global shared bus, making them inherently unscalable beyond a few dozen processors. More recently, scalable directory-based cache-coherent shared address space architectures are being offered by many vendors. Nearly all of the architectures are NUMA; *i.e.*, they have physically distributed memory, but they offer a cache-coherent view of the entire memory to all the processors. Since access to local memory is faster than access to non-local

*This work was supported by NSF CCR-9423082, by Army Research Office contract DA/DAAH04-95-1-0538, by Army High Performance Computing Research Center cooperative agreement number DAAH04-95-2-0003/contract number DAAH04-95-C-0008, by the IBM Partnership Award, and by the IBM SUR equipment grant.

University of Minnesota, Department of Computer Science
University of Minnesota, Department of Computer Science
Purdue University, Department of Computer Science

memory, it is still useful to partition the data intelligently among processors. However, directory-based cache coherent hardware makes it possible to program these machines without specifying explicit communication.

Technological trends indicate that in future generations of NUMA machines, the gap between local and non-local accesses will be substantially smaller, making these machines closer to UMA architectures. If the ratio of local and non-local access is close to one, then even data partitioning and placement becomes irrelevant, and these machines can be programmed in the true shared-memory paradigm. Considering the technological and economical feasibility of such machines and the apparent difficulty of programming in message-passing, it is reasonable to ask "Is message-passing obsolete?".

It turns out that a good message-passing program specifies much more than partitioning of data and explicit communication of information among processors. Since the user is responsible for explicitly specifying interactions among the processors, he/she can ensure that there are no conflicts in the accesses made by different processors. That is, the programmer is fully aware of the costs involved in performing various message-passing operations. In a shared-memory paradigm, such conflicts can happen inadvertently. For example consider the following shared-memory parallel program for multiplying two matrices $A \times B = C$.

> Let p be the number of processors and n the order of the matrices. All three matrices are shared and are a blocked in $\sqrt{p} \times \sqrt{p}$ blocks each of size n^2/p. Each one of the processors is assigned to compute on of the blocks of C, *i.e.*, processor $P_{i,j}$ computes $C_{i,j}$. Thus, processor $P_{i,j}$ performs the following:
>
> $$C_{i,j} = \sum_{k=0}^{\sqrt{p}-1} A_{i,k} * B_{k,j}$$

Looking closer at this example we see that at any one of the $\sqrt{p}$ steps, $\sqrt{p}$ processors will be accessing the same block of A and B. In particular, all the processors that work on the same row of C will be accessing the same block of A. Similarly, all the processors working on the same column of C will be accessing the same block of B.

A typical message-passing program to perform this computation is by using Cannon's [1] algorithm as follows.

In Cannon's algorithm, the processors are arranged in a $\sqrt{p} \times \sqrt{p}$ two-dimensional topology, and processor $P_{i,j}$ stores the (i, j) block of matrices A, B, and C. Processor $P_{i,j}$ is responsible for computing $C_{i,j}$. The following code is executed by each processor $P_{i,j}$. The arrays a, b, and c store the portion of A, B, and C assigned to each processor.

1. Send a to $P_{(i-j)\%\sqrt{p},j}$ and receive from $P_{(i+j)\%\sqrt{p},j}$ in a.
2. Send b to $P_{i,(j-i)\%\sqrt{p}}$ and receive from $P_{i,(j+i)\%\sqrt{p}}$ in b.
3. Initialize c to 0.
4. for $(k = 0; k < \sqrt{p}; k++)$ {
5.   c += a*b
6.   Send a to $P_{(i-1)\%\sqrt{p},j}$ and receive from $P_{(i+1)\%\sqrt{p},j}$ in a.
7.   Send b to $P_{i,(j-1)\%\sqrt{p}}$ and receive from $P_{i,(j+1)\%\sqrt{p}}$ in b.
8. }

Cannon's algorithm performs an initial alignment of the blocks of A and B (lines 1-2), and after that the computation proceeds by performing the matrix-matrix multiplication of the blocks in a and b followed by a leftward shift of a and an upward shift of b. Note that '%' indicates a module operation.

Comparing the two programs we can easily see that the message-passing one is much longer. However, it does not suffer from the potential contention of the shared-memory program. Thus, Cannon's algorithm is appropriate even for the shared memory architectures even though it appears inherently a message-passing algorithm. However, a better approach is to take Cannon's message-passing algorithm and write a shared-memory program for it. This can be done by eliminating from the algorithm any primitives for sending and receiving data and focus on the order in which the operations are performed. In particular, Cannon's algorithm computes $C_{i,j}$ by multiplying the required blocks of A and B in the following order:

$$C_{i,j} = \sum_{k=0}^{\sqrt{p}-1} A_{i,(i+j+k)\%\sqrt{p}} * B_{(i+j+k)\%\sqrt{p},j}$$

If we modify our earlier shared-memory program to use the order indicated by Cannon's algorithm, we see that the new program does not suffer from the potential memory contention problems of the earlier program.

Another major advantage of message-passing algorithms is that they allow the user to optimize data movement. Consider the situation in which it is necessary to compute a global sum of some M values computed by each processor. In a message passing paradigm this will be computed by the reduction operation, and will take $O(M)$ or $O(M \log p)$ time, depending upon the size of M [2]. But in a shared memory paradigm, it will be natural to write the following at each processor

lock(global-var)
Add the elements in the global-sum array.
unlock(global-var)

This will clearly take $O(Mp)$ time, and will be much more inefficient compared to the message passing solution. We must also add that the "collective" operations such as reduction do not need to be implemented explicitly, and are available as convenient functions calls in libraries such as MPI.

Even the explicit data partitioning aspect of message-passing is quite relevant in the context of NUMA machines in which the local and remote access latencies are quite similar. The reason is that even in these machines the bandwidth between a processor and its local DRAM is usually much higher than the effective bandwidth between a processor and non-local memory, especially for configurations with large number of processors. Hence, in such machines data placement is relevant for most parallel programs.

We must note that programming these NUMA cache-coherent machines using a message-passing library such as MPI can have some disadvantages as well. Note that in the message-passing paradigm, sharing of data requires explicit copying, whereas a shared memory paradigm will allow the data to be used in place by all the processors. Also, a communication using message passing libraries requires cooperation by both sending and receiving processors, which can lead to many wasted cycles.

Hence, even if a message-passing library is not the most efficient and convenient way to use a shared-address space architecture, the message-passing paradigm is very useful for conceptualizing parallel programs. Of course, on a cluster of workstations connected via some high latency LAN, message-passing is the only reasonable way to program them.

References

[1] L. E. Cannon. *A cellular computer to implement the Kalman Filter Algorithm*. PhD thesis, Montana State University, Bozman, MT, 1969.

[2] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cummings Publishing Company, Redwood City, CA, 1994.