A Locally Cache-Coherent Multiprocessor Architecture

Kevin Rich
Computing Research Group
Lawrence Livermore National Laboratory
Livermore, CA 94551

Norman Matloff
Division of Computer Science
University of California at Davis
Davis, CA 95616

Correspondence Author: N. Matloff
matloff@heather.cs.ucdavis.edu
(916) 752-1953, (916) 752-7004

Abstract

Recently there has been considerable interest in cache coherency protocols in shared-memory multiprocessor systems, particularly in protocols which are scalable, i.e. suitable for very large systems. However, cache coherency scalability (CCS) entails heavy performance overhead and system cost, so a critical examination of the assumptions underlying the quest for CCS is undertaken here. A non-CCS architecture which provides only "locally, but not globally, coherent" hardware support is proposed, and evidence is presented which shows that this architecture does well in large classes of application. Special emphasis will be placed on loop calculations, due to their prevalence in scientific applications.

1 Introduction

In the last few years, scalability of cache coherency protocols has been one of the most active topics of research in shared-memory multiprocessor systems. A nice survey of this area, with a good list of references, was presented in [9], and new papers have continued to appear since then. However, cache coherency scalability (CCS) requires complex hardware, which carries significant overhead and inhibits system performance, both in absolute terms and with respect to performance/price ratios. It is thus worthwhile to examine the fundamental question of whether CCS is a sufficiently desirable property to warrant the heroic efforts which are being made to achieve it. This question is addressed here, and evidence toward an at least partially negative answer is presented.

The key point involves the type and extent of hardware which should be devoted to support for coherent access of globally shared data, i.e., data which are at least potentially shared by all processors in the system. It is found here that in many large application classes, we can replace the major shared monitor variables (MVs) by data shared only by a fixed number of processors, with that fixed number being independent of overall system size. This obviates much of the need for CCS. Based on this idea, an architecture which provides only "locally, but not globally, coherent" hardware support is proposed which is useful in many application domains, and is particularly suited to loop applications which are common in the scientific computing field.

Section 2 discusses distributed task allocation (DTA), and presents evidence of its efficiency. Section 3 then introduces the architecture which is based on DTA.

2 Decentralized Task Allocation in Loop Contexts

A common problem in the design of efficient parallel processing software is the assignment of loop tasks to processors.[1] We choose to examine loop applications for two reasons. The first is that the concepts here will be easier to explain in the loop setting. Second, loops are a common paradigm in scientific applications. It should be noted, though, that DTA can be applied profitably to many other types of applications.

To illustrate the concepts involved, the simpler the example the better, so let us consider the usual matrix multiplication problem, in which an NxN matrix is to multiply an Nx1 vector. Taking as our basic task the computation of the inner product of the Nth row of the multiplier with the multiplicand, a **for**-loop then consists of N tasks to be allocated to the processors.

2.1 Some Non-DTA Task-Allocation Methods for Loops

Let us consider the various allocation methods compared in [4], and then discuss their relation to DTA. First, under self-scheduling (SS), there would be a global variable NextRow, recording the number of the next row which is available for multiplication. Whenever a processor finishes the multiplication of a row, it performs

```
R = NextRow++;
```

and then does the multiplication of row number R. (Of course, the incrementing of NextRow must be atomic.) This approach is very efficient in terms of load-balancing, but the variable NextRow can become a hot spot, leading large number of cache invalidations, for example. Indeed, [1] found that most accesses of atomically-protected variables such as NextRow led to cache coherency transactions.

Another approach would be to use the *static chunking* (SC) method proposed by Kruskal and Weiss [5]. Here tasks are assigned in "chunks" of K, instead of singly. In other words, the code above would become

```
R = NextRow;
NextRow += K;
```

This reduces the number of accesses of NextRow by a factor of K, but is less efficient in terms of load-balancing.

In an effort to get the load-balancing efficiency of SS and the reduction in accesses of NextRow which stems from SC, a number of hybrid methods have been proposed, based on the idea of guided self-scheduling (GSS) [4]. The latter is a variable-sized chunking method, with smaller chunks being used in later iterations. A variant of GSS proposed in [4], which we will call Factoring, was shown to do quite well in the **for**-loop setting (it is specifically designed for that setting), with performance superior to SS and SC.

Another queue-access method (not limited to task allocation) that has received considerable attention recently is the use of *software combining* (SWC) to reduce contention at the particular memory module that contains the global task queue [3, 10]. The task-allocation software uses a binary tree data structure, with the root node being the only one with direct accesses to the global task queue. Processor requests for additional work are combined at the nodes of the tree whenever possible, distributing the memory accesses across many nodes, and through careful planning, many memory modules, thus reducing contention. In the matrix example here, the code would look like

```
R = Fetch&Add(NextRow,1);
```

Another method similar to GSS, called trapezoid self-scheduling was presented in [11], but appeared too late for us to include in our empirical evaluations. The aim of this method was the same as the others, to reduce the number of accesses to the global task allocation queue. Our analysis showed that in "typical" problems it produces many more global accesses than DTA.

[1] In the context considered here, we focus on processors, rather than processeses.

2.2 DTA in the Loop Context

The method on which we will focus here is distributed task allocation (DTA). This method would approach loop problems such as matrix multiplication in the following way: Processors are considered to belong to groups, of size G.[2] Usually tasks are allocated locally, i.e. within groups, hence the term *distributed* in "DTA." However, occasionally one member of a group must go to a global variable to acquire a batch of new tasks for the group to process.

Specifically, the direct analogy of NextRow is now an NG-element array LocalNextRow, where the group size NG is equal to NPROCS (the number of processors in the system) divided by G. There is also a variable GlobNextRow and another NG-element array LocalLastRow. The I-th processor is considered to be in group GN = I mod NG. When this processor finishes the multiplication of a row, it performs

```
R = LocalNextRow[GN]++;
if (LocalNextRow[GN] > LocalLastRow[GN])  {
   LocalNextRow[GN] = GlobNextRow;
   LR = GlobNextRow += K;
   LocalLastRow[GN] = LR - 1;
}
```

What is happening here is the following: Initially each processor group is allocated a set of K consecutive rows of the matrix. The processors in that group process these rows one-by-one, just as in the centralized case, with LocalNextRow[GN] playing the role corresponding to NextRow in the centralized versions. When one of the processors discovers that these K rows have all assigned to processors, it goes to GlobNextRow to get K more rows for its group. This is in contrast to the SC method, in which each *processor*, rather than each group, is assigned K tasks at a time. Note that though the entire arrays LocalNextRow and LocalLastRow are global variables visible to all processors—this is a shared-memory system, after all—the code is written so that a processor in one group will never access the array element intended for another group.[3]

K is a design parameter, which in the experiments here is taken to be equal to G.

2.3 Experimental Results

As mentioned earlier, our focus will be on DTA. We are interested in DTA because of its broad range of applicability:

- DTA is applicable in open-ended problems in which the total number of tasks to be done is not known in advance. This is in contrast to, for example, the Factoring approach, which is applicable only in **for**-loops.
- DTA also applies to other problems commonly arising in the parallel processing area, such as barrier synchronization.[4]
- Most importantly, though we are in this section viewing DTA as a software technique, DTA forms the justification for the architecture which we will propose in the next section, and which is the main subject of this paper.

What we gain with DTA is the low number of accesses of the central resource NextRow (or in this case, the new central resource, GlobNextRow) enjoyed by SC, but with higher processor utilization than SC. The question, though, is how *much* higher that utilization will be, compared to the optimal method in terms of utilization, SS.

This question was studied by Ni and Wu [8], but not in the scalability context of interest here. Work in the latter context was done in [7], in which a theoretical analysis was presented which showed that DTA can indeed be done efficiently with a fixed group size, even with arbitrarily large overall system size. Moreover, it was found that in a certain sense a "universal" group size exists, information which then can be incorporated into hardware design.

[2] Taking DTA as only a software technique, as in [8], this grouping is just symbolic, not physical. However, it will take on a physical embodiment when we propose the LCSMA architecture in the followig section.

[3] Clearly, this concept can be extended in a hierarchal manner, for extremely large systems.

[4] Again, SWC can be used for this.

Now in the current work, this efficiency is demonstrated empirically by running three specific applications, and comparing them to SS, SC, Factoring and SWC. Note, by the way, that the empirical study is also important in that it accounts for the overhead a processor spends in actually acquiring a task from a task allocator, e.g. in row-assignment in the matrix problem, which the previous theoretical analyses [8, 7] did not do.

A summary of the results is:

- For the smaller system sizes DTA had performance comparable to that of Factoring.
- For the larger systems DTA was superior to Factoring, sometimes substantially so.
- DTA was superior to both SS and SWC at all levels.

Here is some more detail. The experiments reported here were conducted on a BBN TC2000 (a.k.a. BBN Butterfly) shared-memory multiprocessor consisting of 128 processor/memory nodes. The shared memory consists of the totality of memory modules at all these nodes, with internode access being via a multistage network. Local operating system structure allowed us to run programs on a dedicated-machine basis, i.e., with all other jobs suspended, except for certain interactive jobs which run on a reserved set of four nodes. For each combination of parameters, at least 15 (and as many as 65) runs were conducted, with the timings graphed here being the average values so obtained. All of the DTA experiments reported here used a group size of G = 16, and thus the numbers of processors used were various multiples of 16. Since four of the 128 processors are unavailable, the maximum number of processors we used was 112. The graphs presented here plot program timings t against numbers of processors p.[5]

Figure 1 presents the data for the matrix-multiply experiments, in which an NxN matrix multiplies an Nx1 vector.

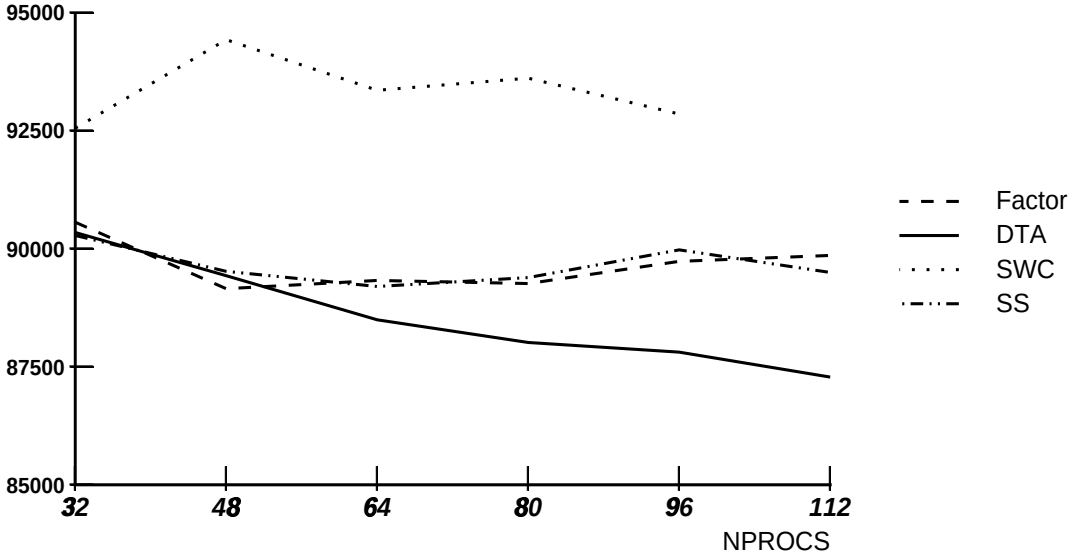

Figure 1: Matrix Multiply (N = 300)

Figures 2 and 3 give the results for the sorting experiments on matrices of size NxRS, in which a basic task is to sort one matrix row.[6] DTA, SS and Factoring were roughly equal for the smaller numbers of processors, but with DTA having a decided advantage for NPROCS $>=$ 64.

Though not discussed here, we also considered a **while**-loop application. Factoring cannot be used in this context, so we compared DTA to SS and SWC, and found DTA to yield very strong improvements over the other two methods.

[5] Note carefully the values on the vertical axis. They generally do not start at 0, and thus tend to exaggerate the differences.

[6] A standard C-library Quicksort routine was used for the sorting.

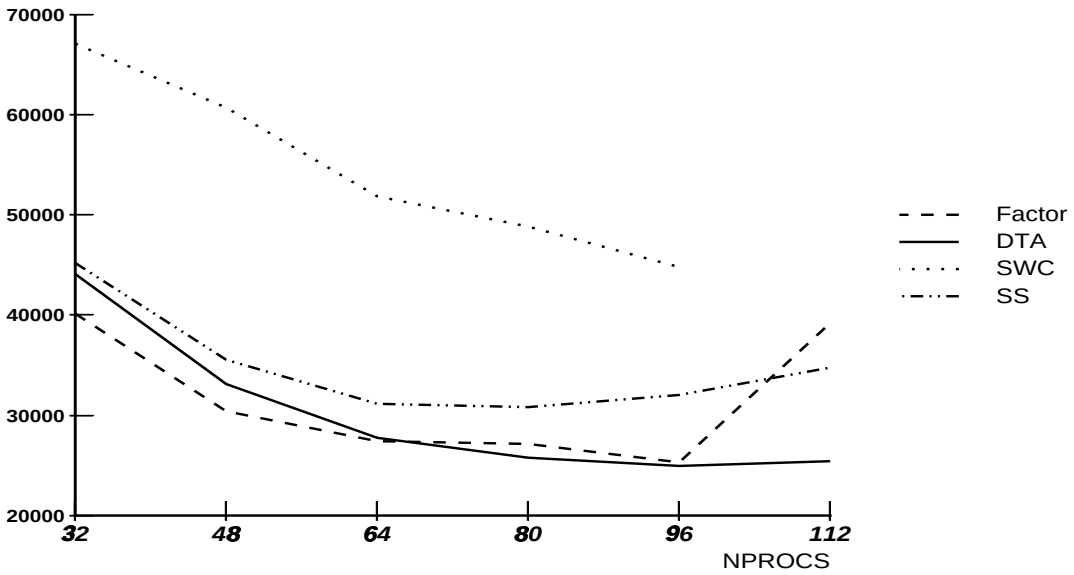

Figure 2: Sort (N= 500; RS = 25)

3 A New Class of Shared-Memory Multiprocessor Architecture

Recall our previous statement that [1] found that in the systems using an invalidation-based cache coherency policy, most accesses to variables like NextRow[7] in the last section caused cache invalidations. As system size grows, more and more processors are accessing NextRow, and the problem gets worse and worse. DTA solves this problem, because each variable LocalNextRow[GN], is accessed only by at most a fixed number of processors (GN). The variable GlobalNextRow does get accessed by all processors, not just those in one group, but the point is that such accesses are rare. This has a very profound implication for the CCS question, in that neither the variables LocalNextRow[GN] nor the variable GlobalNextRow need CCS hardware:

- The 'S' (for *scalability*) in "CCS" is irrelevant to the variables LocalNextRow[GN], since each such variable is accessed only by the processors in group GN.
- The variable GlobalNextRow is accessed so rarely that special hardware to make atomic access efficient is not justified.

The second point here is central. As mentioned earlier, CCS hardware adds greatly to system cost, and inhibits system performance. These problems can be avoided in loop applications by the use of DTA, and the empirical results of the last section and the mathematical analysis in [7] indicate that this can be done without inducing load-balancing problems.

In this regard, consider a concept of *locally coherent shared-memory architectures* (LCSMA), which we define to mean shared-memory multiprocessor systems which provide hardware support for cache coherency within groups of processors, but provide no hardware support for systemwide cache coherency. Note that machines such as DASH [6] are not LCSMAs. Though their use of processor clusters seems to have some similarity to our processor-group concept, the point is that they do have hardware devoted to systemwide cache coherency. LCSMA has no such hardware.

LCSMA denotes a broad class of architectures, with many possible implementations. For example, an interconnect structure such as that of Figure 4 could be used. (For the purposes of readability, this figure depicts a very small system, with an unrealistically small value of 4 for the group size G.) Here processor elements (PE) contain the processor, cache, and some of the global memory. PEs are connected to other PEs via the familiar Ω-network. What is different is that PEs are partitioned into groups, and processors within any given group are connected via snooping

[7] Or to lock variables guarding variables like NextRow.

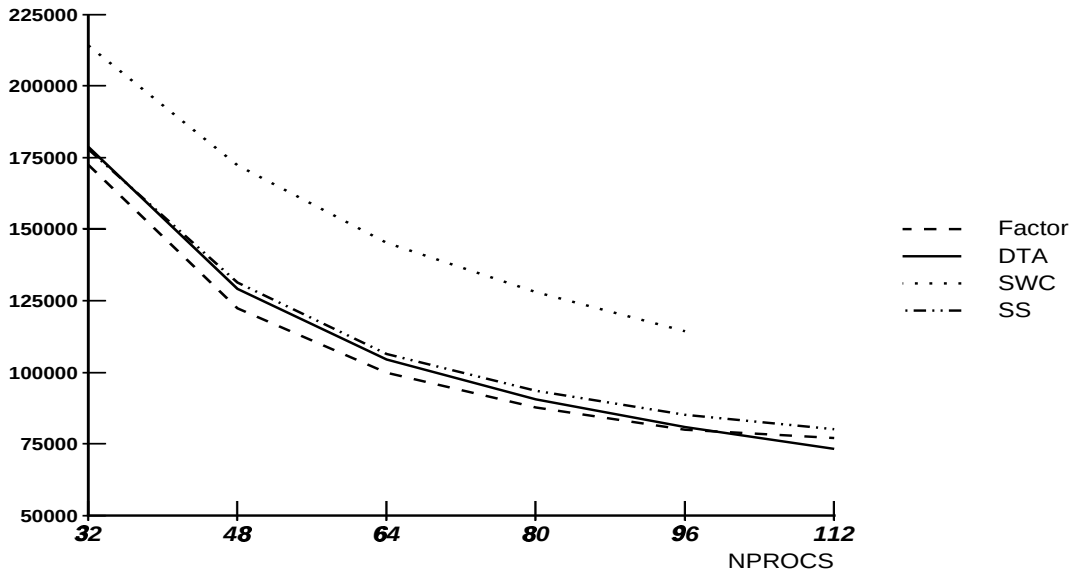

Figure 3: Sort (N= 1000; RS = 50)

caches [2] and a local bus. These provide hardware support for local MVs, i.e., variables which are shared only by processors within a given group, such as LocalNextRow[GN] in our examples in the last section.

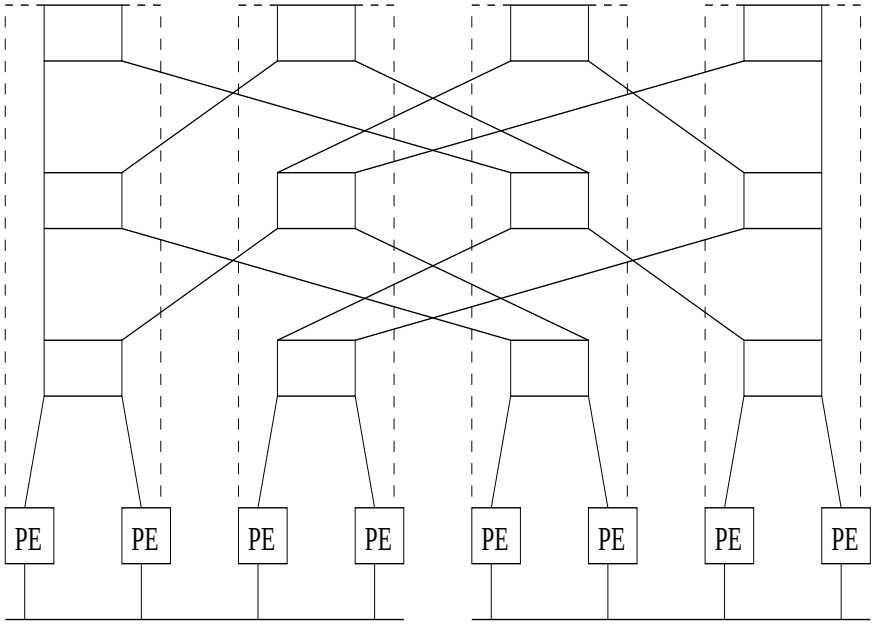

Figure 4:

Under LCSMA the software must, where possible, avoid creation of systemwide MVs. The previous section indicated how to do this with DTA in the case of **for**-loops, and DTA can be extended—still with efficiency from the load-balancing point of view—to many other kinds of task-allocation mechanisms, e.g. task queues [7]. Atomic access to group-specific variables like LocalNextRow[GN], and the problem of contention for them, are handled via hardware, e.g. the "locally-snoopy" buses in Figure 4. Any MVs which remain, e.g. such as GlobalNextRow, are handled in hardware at the group level, and then in software above that level, such as with software fetch-and-add as in [3, 10].

In addition, LCSMA should have program-controlled cacheability of individual blocks, to allow the best usage, with the program having the ability to dynamically set the cacheability mode (cacheable, read-only cacheable, non-cacheable) for the block containing a given address/variable, especially for variables which are not MVs.

We will not present details here, but as a simple illustration, consider Gaussian elimination. Since each row is accessed by a processor only once per iteration, caching is not useful, and the rows should be made noncacheable. In many other matrix applications, rows are re-used within an iteration, but only for reading, so read-only caching would be appropriate.

4 Discussion

Though for simplicity and conciseness we have limited our focus here to loops, it is important to note that DTA can be used in a much wider variety of applications. For example, problems with general task queues can be converted to having local task queues, rather than using one central queue. The theoretical work in [7] lends support to this. Thus we believe that LCSMA is a good choice for general parallel processing applications.

On the other hand, no machine can be optimal for all applications, and we note that LCSMA does not provide any special help in, for instance, synchronous algorithms with very short times between successive synchronizations, such as parallel root-finding problems. However, it is clear that approaches based on systemwide cache coherenecy are not good solutions to this problem either. The overhead due to relaying of cache update messages throughout an entire large system would be too heavy. Thus other mechanisms would be needed if this type of application were to be targeted, say adding a separate broadcast channel to Figure 4.

References

[1] A. Agarwal and M. Cherian. "Adaptive Backoff Synchronization Techniques," *Proceedings of the 16th Annual International Symposium on Computer Architecture*, 1989, 396-406.

[2] J. Goodman. "Using Cache Memory to Reduce Processor-Memory Traffic," *Proceedings of the 10th Annual Symposium on Computer Architecture*, 1983, 124-131.

[3] J. Goodman, M. Vernon and P. Woest. "Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors," *ACM Supercomputing*, 1989, 64-75.

[4] S. Hummel, E. Schonberg, L. Flynn. "Factoring: A Method for Scheduling Parallel Loops," *Communications of the ACM*, August 1992, 90-101.

[5] C. Kruskal and A. Weiss. "Allocating Independent Subtasks on Parallel Processors," *IEEE Transactions on Software Engineering*, SE-11 (10), 1001-1016, 1985.

[6] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta and J. Hennessy. "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor," *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990, 148-158.

[7] N. Matloff "On Decentralized Cache Coherency and Scalable Cache Coherency," *Technical Report, University of California at Davis*, 1991.

[8] L. Ni and C.-F. Wu. "Design Tradeoffs for Process Scheduling in Shared Memory Multiprocessor Systems," *IEEE Transactions on Software Engineering*, 15, 1989, 327-334.

[9] P. Stenstrom. "A Survey of Cache Coherency Schemes for Multiprocessors," *Computer*, June 1990, 12-24.

[10] P. Tang and P.-C. Yew. "Software Combining for Distributing Hot-Spot Addressing," *Journal of Parallel and Distributed Computing*, 10, 1990, 130-139

[11] T. Tzen and L. Ni. “Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers” *IEEE Transactions on Parallel and Distributed Systems*, January 1993

[12] P.-C. Yew, N.-F. Tzeng and D. Lawrie. “Distributing Hot-Spot Addressing in Large-Scale Multiprocessors,” *IEEE Transactions on Computers*, C-36, 1987, 388-395.