Design Issues in Parallel Array Languages for Shared Memory *

James Brodman[1], Basilio B. Fraguela[2], María J. Garzarán[1], and David Padua[1]

[1] University of Illinois at Urbana-Champaign, Dept. of Computer Science
201 N. Goodwin Ave, Urbana, IL 61801 USA
{brodman2,garzaran,padua}@uiuc.edu
[2] Universidade da Coruña, Dept. de Electrónica y Sistemas
Campus de Elviña, s/n. 15071 A Coruña. Spain
basilio@udc.es

Abstract. The Hierarchically Tiled Array (HTA) is a data type that facilitates the definition and manipulation of arrays partitioned into tiles. The data type allows to exploit those tiles to attain both locality and parallelism. Parallel programs written with HTAs are based in data parallelism, and provide the programmer with a single-threaded view of the execution. In our experience, HTAs help to develop parallel codes in a much more productive way than other parallel programming approaches. While we have worked extensively with HTAs in distributed memory environments, only recently have we began to consider their adaption to shared memory environments such as those found in multicore systems. In this paper we review the design issues, opportunities and challenges that this migration raises.

Key words: parallel programming, data parallelism, tiling, shared memory

1 Introduction

Arrays are one of the most basic and useful data structures. The parallelism in the operations on their components, expressed as array operations and functions, has been exploited successfully since the days of the early array and vector processors [1]. The efforts to express the data parallelism in array operations have been often implemented as new languages or language extensions. The historical experience in the attempts to implant new languages with a focus on parallelism, coupled with the large base of existing legacy codes, makes us think that macros, and more in general, libraries, are a better vehicle to bring parallelism to mainstream computing. The advent of object oriented (OO) programming further supports our observation, as it enables to associate methods or tasks with sets

* This material is based upon work supported by the National Science Foundation under Awards CCF 0702260 and CNS 0509432. Basilio B. Fraguela was partially supported by the Ministry of Education and Science of Spain, FEDER funds of the European Union (Projects TIN2004-07797-C02-02 and TIN2007-67537-C03-02).

of data. In OO languages arrays can contain objects of any kind, and the operations on them need not be restricted to be the traditional simple mathematical operations as was the case for SIMD implementations. Rather, arbitrary tasks encapsulated as methods can be performed in parallel on the elements of the array. Integration of libraries and classes that express parallelism in OO languages is further facilitated by their polymorphic features and operator overloading, when available.

Tiling [2] is closely related to array processing. Tiles are used both to increase the locality of the accesses in sequential programs [3] and to describe data parallelism [4–7]. This led us to the development of the *Hierarchically Tiled Array* (HTA) data type [8]. HTAs represent arrays partitioned into tiles which can be further partitioned recursively. When parallelism is expressed using HTAs, programs have a single logical thread of execution. They express parallelism as array operations on HTAs, with the operations on the different tiles of an HTA taking place in parallel. This gives structure to parallel operations, which improves readability and maintainability over the SPMD (Single Program Multiple Data) approaches. The tiling allows to choose the granularity of the tasks with different purposes. For example, the number of tasks can be chosen so that the local working set fits in the memory of a node in a distributed memory environment. The tiles in an HTA can be recursively subtiled in order to subdivide the work to perform so that the data to process at each time fits in a given level of the memory hierarchy of the machine.

We have experimented with this data type for a number of years now [8, 9] using both typical parallel benchmarks such as NAS [10] and serial codes that benefit from tiling. Our experience is that HTAs allow to write these codes in a much more productive way than traditional approaches while achieving good performance. Still, we have always worked on distributed memory environments, and it is for them that we have defined the semantics of our data type. Given the growing importance of multicore systems, and the conviction that most HPC systems in the future will have a hybrid memory model, the moment to define the HTA implementation options and semantics for these environments and build an HTA library for hybrid memory models has arrived.

The rest of this paper is organized as follows. Section 2 is a brief introduction to the HTA data type. Section 3 reviews the design issues that an HTA implementation for shared memory systems poses. Finally, we present our conclusions in Section 4.

2 The Hierarchical Tiled Array

The Hierarchically Tiled Array (HTA) [8] is an array data type which can be partitioned into tiles. Each tile can be either a conventional array or a lower level HTA. Tiles in an HTA can conceptually be mapped on to different levels of the memory hierarchy. At the top-most level, tiles can represent portions of the array that map to different nodes in a cluster. Each of those tiles could then carry additional levels of tiling that then map to the various levels of cache

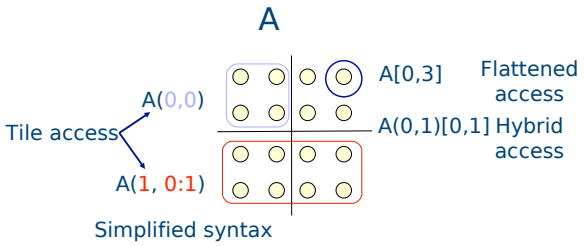

Fig. 1. HTA Indexing, () are used to index tiles, [] to index scalars

in a machine. One could then further partition the tiles to map the individual scalars to registers. Programmers see a single-threaded view of execution in HTA programs. Parallelism takes the form of concurrent operations across tiles. The tile size thus provides the granularity of parallel execution.

Figure 1 illustrates the three ways in which HTAs may be indexed. One can index HTAs at both the tile and scalar level and combine both indexing schemes. One can flatten the tiling structure to directly access the scalar elements of HTAs. A hybrid approach can also be used when the programmer wants to access one or more scalars found in one or more tiles. In the example, `A(0,0)` indexes the first tile in the first row. Likewise, `A(1,0:1)` indexes the entire second row of tiles. `A[0,3]` indexes the last scalar on the first row using the flattened notation. The same scalar can be indexed as `A(0,1)[0,1]` using the hybrid scheme.

HTAs support the three main constructs found in data-parallel computations:

- *Element-by-element operation*: A function is applied to each element of an array or corresponding elements of two or more conformable arrays.
- *Reductions*: These apply operations on an array to produce an array of lesser rank. For example, computing the sum of the elements of a one-dimensional array produces a scalar.
- *Scan*: A function computes a prefix operation across all the elements of an array.

These operations take the form of three methods in the HTA library: `hmap` (which implements element-by-element operations), `reduce`, and `scan`. The three constructs receive at least one argument, a function object that encapsulates the operation to be performed. In the case of `hmap`, the function may accept additional HTAs as parameters that must have the same tiling structure as the HTA instance on which the `hmap` is invoked. This effectively allows programmers to extend the library with new user-defined operations. A simple example of `hmap` can be seen in Figure 2. Here, two HTAs, `X` and `Y`, with ten tiles of ten elements are created. Function `F` is applied on them by `hmap`. In it, each tile in `X` assigns its elements the sum of their current values plus the values in the corresponding tiles of `Y` plus one, in parallel. HTAs overload the arithmetic operators (+,*,...) and also assignment so that these typical element-by-element operations can be

```
1  HTA X([10], [10])
2  HTA Y([10], [10])
3
4  ...
5
6  hmap(F(), X, Y)
7
8  F(HTA X, HTA Y) {
9    do i=1,10
10     X[i] = X[i] + Y[i] + 1
11 }
```

Fig. 2. hmap Example

expressed in a traditional array syntax instead of requiring the usage of `hmap`. Operations typical of array languages such as matrix multiplication, transposition, or stencil computations are also found in the library. All parallelism is explicit and takes the form of independent operations performed on tiles. Synchronization required by operations such as reductions is implicit and handled by the library.

The tiling structure of an HTA is normally specified at creation time. However, some problems are more naturally expressed in a dynamic or input-dependent fashion. The dynamic partitioning [9] feature enables the modification of the structure of an HTA after its creation by adding or removing partition lines, the abstract lines that separate the tiles in an HTA.

3 Design Issues for Multicore and Shared Memory Systems

As mentioned before, we have an implementation supporting HTAs that runs on distributed memory systems. An HTA program appears to the programmer as having a single thread of execution. Arrays are partitioned and distributed across a set of nodes and non-HTA data such as scalars and non-HTA arrays are replicated on all the nodes. When operating on an HTA each node works on its portion of the HTA. However, when operating on non-HTA data, our implementation uses the SPMD mode, and all the nodes execute the same code on its local copy of the non-HTA data. Synchronization is achieved implicitly, by the underlying send and receive messages used to communicate between the nodes.

When running an HTA program in a shared memory environment the situation changes quite a bit, and a wide space of design options can be explored, each of them resulting on different performance/productivity trade-offs. In this Section we discuss some of the issues that appear in this shared memory environment.

3.1 Dynamic Task Creation

In many parallel programs tasks can be identified before execution begins. In terms of HTAs, this means that HTAs can be created with a given partitioning

suitable for the algorithm that is being parallelized and that this partitioning does not need to be modified later. In the HTA library, the parallel tasks are determined by the tiles. In distributed memory systems, the parallelism is also determined by the distribution of tiles to processors. In these systems, dynamic task creation or, in other words, repartitioning of an HTA, involves an expensive redistribution of the data. For this reason, we did not implement dynamic partitioning in our distributed memory implementation. Instead, we leave programmers to solve the problem.

In shared memory systems, however, the ability to create tasks dynamically, that is, to define and spawn parallel subtasks from a parallel task, can be useful and even necessary to obtain good performance. Dynamic task creation is good for two reasons: it allows programmers to more elegantly write their algorithms and can be used to improve load balance.

In HTA programs, tasks are created by `hmap`. `hmap` can be implemented by a parallel loop where each iteration corresponds to a tile. If the parallel loops are implemented using Intel's Threading Building Blocks [11], each processor is assigned a range of iterations and idle processors can steal part of the range of another processor, splitting one task into two. Dynamic task creation by the library, rather than by the program, could help improve load balance. These issues will be further discussed in Section 3.2.

Task creation can also be hierarchical as illustrated by the example in Figure 3, parallel merging of two sorted sequences (`input1` and `input2`), where the partitioning is dependent on the input to be merged. A rough sketch of the algorithm states that one first splits the first input HTA in half. Next, the location of the first element greater than the midpoint element of the first input HTA is found in the second HTA and used to partition it. The output is partitioned such that its new tiles can fit the merged elements from the respective tiles of the input arrays. Finally, `hmap` recursively calls the Merge operation on the newly created left tiles of the two input arrays as well as the right tiles. Here, the `Merge` operation creates a tree of tasks during the course of its recursion.

```
1  Merge(HTA output, HTA input1, HTA input2) {
2  ...
3    if (output.size() < THRESHOLD) {
4      SerialMerge(output, input1, input2)
5    }
6    else {
7      i = input1.size() / 2
8      input1.addPartition(i)
9
10     j = h2.location_first_gt (input1[i])
11     input2.addPartition(j)
12
13     k = i + j
14     output.addPartition(k)
15
16     output.hmap(Merge(), input1, input2)
17   }
18 ...
19 }
```

Fig. 3. Parallel Merge

Note that in this example, dynamic partitioning enables the implementation of merge in this elegant manner. In this case, dynamic task creation for both algorithmic elegance and load balancing can be combined. The programmer could change the number of partitions created in each invocation of `Merge` to improve load balance.

3.2 Locality vs Load Balancing

Conventional notation for task parallelism does not provide a convenient mechanism to express locality [12, 13]. However, locality is very important to achieve good performance in multicore systems due to the existence of a hierarchy of private and shared caches, coupled with shared buses to memory systems which are much slower than the processors they feed. With HTAs we solved this problem in a very natural way for distributed systems. However, in shared memory, extensions are needed to achieve locality.

In shared memory we could promote locality by assigning tiles to processors when HTAs are created and maintaining this assignment throughout the program. This is similar to the way HTAs operate in distributed memory systems. However, the affinity between processors and tiles provided by this assignment would hinder load balancing.

To solve the problem of load imbalance a more dynamic strategy is necessary, and that could be to use task stealing. Task stealing provides a mechanism for dynamic task scheduling. With task stealing a processor places tasks into its own queue upon creation. Task queues could be implemented using parallel programming libraries like Intel's Threading Building Blocks [11] or written independently. Idle processors can then steal tasks and their associated tiles from the queue for execution. Task stealing has different implications for affinity depending on whether the parallelism resembles loops or whether it is hierarchical. The former case refers to common mathematical operations and `hmaps`. Here, in order to achieve locality, parallel operations performed on the same HTA should respect any prior affinity between processors and tiles, that is, execute operations on a given tile on the same processor that used it before. Task stealing should follow this approach when possible. However, task stealing can choose to change the affinity between tiles and processors when a load imbalance exists, trading better utilization for negative effects on locality.

The case of hierarchical task parallelism is shown in the Parallel Merge example in Figure 3. Here, the `Merge` function performs an operation on the input before partitioning. The new subtasks created by the subsequent invocation to `hmap` would ideally be placed in the queue of the processor that created the tasks because the data are in its cache. However, task stealing any of these dynamically created tasks can be necessary to balance the load, what would change the affinity of the tiles to processors. Care must be taken to properly address the consequences of affinity in our design.

In addition to the concerns about locality, load balancing could also have consequences on the correctness of HTA programs. Dynamic partitioning, as mentioned in the previous section, provides another alternative to load balancing

in HTAs. When load imbalance is detected, a run-time element of the library could decide to dynamically split the tiles in an operation, creating additional tasks with smaller granularity. For that, the library would have to provide a new `hmap` or the programmer would have to annotate the operation passed to `hmap` to inform the library that such splitting is legal, that is, that the parallel operation defined would be legal if the size or shape of the tiles changed. The library could choose from several partitioning strategies such as split on the largest dimension, split on the smallest dimension, quarter, etc. The library would also need to know if it is safe to permanently alter the tiling structure of an HTA or if the original tiling structure must be restored at the end of the parallel operation.

Dynamic scheduling of tasks in the distributed memory implementation of HTA was not feasible. Consequently, it was up to the programmer to distribute tasks to processors in such a way as to distribute the load as evenly as possible. However, the lack of communication involved in dynamically moving tasks in order to improve load balance in shared memory has led us to explore this option for our implementation on these systems. We must handle affinity in a proper fashion whether our parallelism comes from `hmap` or from hierarchical dynamic task creation. We must also ensure that if the library is allowed to dynamically create tasks to improve that load balance that it does so in a correct manner.

3.3 Execution Models

In choosing the execution model for our shared memory implementation, we have two choices:

- *Master Thread*: A single master thread executes all serial portions of the program, and creates tasks for worker threads to run in the parallel portions. There is a single copy of shared non-HTA data.
- *Thread Private (SPMD)*: Every processor executes the whole program. However, each processor only executes operations on its own data during parallel operations. Shared data is replicated across all processors.

Different programming environments have chosen different answers to this problem, e.g., UPC [6] follows the SPMD model and OpenMP [13] follows the master thread model.

The master thread approach is conceptually the simpler of the two. One processor executes the sequential portions of the code and only one copy of shared data exists. When a parallel operation occurs, the master thread spawns tasks that the other processors execute. This approach requires synchronization at the beginning and the end of the parallel operations. One can imagine a parallel operation in this model as a parallel loop that iterates over all the tiles in an HTA, applying the operation to each tile. The loop itself would be executed only by the master thread, with the execution of the parallel body being assigned to different threads for different iterations. In the SPMD model each processor would execute all iterations of the loop, but the operations on each tile would be executed by only one processor. Another difference between the SMPD and

```
1  int A[10]
2
3  B() {
4    do i = 1,10
5      H(i) = H(i-1)
6  }
```

```
7  C() {
8    do i = 1,10
9      A[i] = i;
10   hmap(F, H, A)
11 }
12
13 F(HTA H, int[] A) {
14   do i=1,10
15     H[i] = H[i] + A[i]
16 }
```

Fig. 4. Examples

the master thread approaches is that SPMD has a larger footprint due to the replication of the non HTA data. In either approach, threads do not explicitly communicate. Threads correspond to independent operations on tiles and no synchronization by the programmer is needed.

Figure 4 helps illustrate the differences between both models. In function `B`, we assign each tile the values of the previous tile. Using the master thread approach, this occurs serially on only one processor. However, using SMPD, this loop will run in parallel, although the dependences will result in a serialization of the code. The reason is that processor that owns tile `i` has to wait for a signal from the processor that owns tile `i-1` in order to perform its assignment. In shared memory, rather than using the *owner-computes* rule, SPMD can follow the more relaxed *Single Computation* rule. When an access to a tile of an HTA occurs, it is only performed once. Such an access could be handled by a single processor or even by multiple processors working on different sections if the tile is dynamically partitioned.

The handling of the shared data on these two execution models has implications on data locality. Remember that non-HTA data is shared in the master thread approach and replicated (and as result thread-private) in the SPMD model. For example, Function `C` in Figure 4 illustrates a shortcoming of the master thread approach. The master thread computes the values in array `A`, so when the processors perform function `F` on each tile of `H`, the non-HTA data, array `A`, is only in the cache of the processor that executed the serial portion of the code. However, in the SPMD model, array `A` is replicated across all processors. Thus, each processor assigns its own copy of the array `A`. This ensures non-HTA data will be in every processor's cache when `hmap` performs the parallel operation. Ultimately, performance should dictate which model we choose. We have not yet performed experiments to determine which model will provide better performance.

Finally, notice that under both execution models modifications of non-HTA data within `hmap` functions should not be allowed. The reason is that `hmap` is fully parallel. Thus, under the master thread model, synchronization would be necessary for correctness and this will result in `hmap` not being fully parallel. Under the SPMD model, such accesses will be a programming error, as different results could be obtained in different processors. Ideally, the library should disallow such

```
1 X = ...
2 hmap(... , X)
3 Z = X
4 X = ...
5 hmap(... , X)
```

Fig. 5. Copy Example

accesses, but the languages in which the HTA library can be implemented do not provide the necessary mechanisms.

3.4 Reference/Value Semantics

In our distributed memory implementation, assigning an HTA to another that is distributed in a different way implicated data copying. The copy could be immediate or delayed using a lazy implementation, but the semantics was always that of a copy by value. In shared memory we can choose between copies by value or by reference, also called deep and shallow copies, respectively. For example, a shared memory implementation could use a copy by reference model implemented through a copy-on-write strategy where a shallow copy that only copies the pointers is used until a write occurs, at which point a deep copy of all the data must occur. However, copy by reference introduces additional overhead into the library as proper reference counts must be kept to ensure that memory is de-allocated at the appropriate time. In addition, this scheme can potentially affect the affinity between tiles and processors as is illustrated by Figure 5. In this example, HTA `X` is initially assigned some values and then an operation is performed on it using `hmap`. Next, another HTA, `Z`, copies `X`. `X` is then changed again and another parallel operation occurs. Under the copy by value scheme, `Z` would be a new copy of `X`. When `X` is changed and then used by a computation, the tiles of `X` could still be in the caches of the processors that operated on them in the first `hmap`. However, under copy by reference, the second write to `X` would cause `X` to be the new copy, with `Z` continuing to point to the original data. The second `hmap` could then find that the tiles of `X` have changed affinity if the copy is not careful to preserve it. Intuitively, the easier implementation and lesser bookkeeping of copy by value leads us to believe that this is, on average, the faster strategy since one does not usually copy HTAs without modifying them afterwards. However, this conjecture would need to be experimentally validated.

3.5 New HTA Notations/Constructs

The greater flexibility of access to data by different threads in shared memory environments probably leads to programs with more complex patterns than those we have seen in distributed memory environments. As a result, it could be convenient to extend HTAs with notations to express these structures. For example, new ways to express task dependences, new operators (possibly domain-specific), etc. A very important question is whether these extensions would fit naturally in the clean semantics and array notation that characterize HTAs.

4 Conclusions

In this paper we have reviewed the different design issues that appear when considering a shared memory implementation for the HTA, a data type that allows to express data parallelism as well as locality. These issues can influence the performance and programming flexibility attained with the HTA. We are currently examining the trade-offs of the different options, considering several potential implementations. Our priorities are, in this order, to provide clear semantics to the programmer, to provide a notation as systematic as possible that enables most if not all HTA programs to run correctly in every kind of system, and finally to facilitate the effective parallelization of as many programs as possible using our class. In this process we should also consider their implications in hybrid memory systems.

References

1. Barnes, G.H., Brown, R.M., Kato, M., Kuck, D., Slotnick, D., Stokes, R.: The ILLIAC IV Computer. IEEE Transactions on Computers **8**(17) (1968) 746–757
2. McKellar, A.C., E. G. Coffman, J.: Organizing Matrices and Matrix Operations for Paged Memory Systems. Communications of the ACM **12**(3) (1969) 153–165
3. Wolf, M.E., Lam, M.S.: A Data Locality Optimizing Algorithm. In: Proc. of the Conf. on Programming Language Design and Implementation. (1991) 30–44
4. : High Performance Fortran Forum. High Performance Fortran Specification Version 2.0 (January 1997)
5. Chamberlain, B., S.Choi, Lewis, E., Lin, C., Synder, L., Weathersby, W.: The Case for High Level Parallel Programming in ZPL. IEEE Computational Science and Engineering **5**(3) (July–September 1998) 76–86
6. Carlson, W., Draper, J., Culler, D., Yelick, K., Brooks, E., Warren, K.: Introduction to UPC and Language Specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences (1999)
7. Numrich, R.W., Reid, J.: Co-array Fortran for Parallel Programming. SIGPLAN Fortran Forum **17**(2) (1998) 1–31
8. Bikshandi, G., Guo, J., Hoeflinger, D., Almasi, G., Fraguela, B.B., Garzarán, M.J., Padua, D., von Praun, C.: Programming for Parallelism and Locality with Hierarchically Tiled Arrays. In: PPoPP'06: Proc. of the ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. (March 2006) 48–57
9. Guo, J., Bikshandi, G., Fraguela, B.B., Garzarán, M.J., Padua, D.: Programming with Tiles. In: PPoPP'08: Proc. of the ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. (Feb 2008) 111–122
10. : NAS Parallel Benchmarks. Website http://www.nas.nasa.gov/Software/NPB/.
11. Reinders, J.: Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. 1 edn. O'Reilly (July 2007)
12. Butenhof, D.R.: Programming with POSIX Threads. Addison Wesley (1997)
13. Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., Menon, R.: Parallel programming in OpenMP. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2001)