# Introducing Locality-Aware Computation into OpenMP

Lei Huang[1], Haoqiang Jin[2], Barbara Chapman[1] *
[1] *The University of Houston, Houston, TX*
[2] *NASA Ames Research Center, Mountain View, CA*

**Abstract**

This paper presents our idea to introduce data locality feature into OpenMP. Given the facts that the memory systems are hierarchical while OpenMP is at, we believe that it is important to introduce new features to OpenMP to provide OpenMP programmer capability to manage the data layout and align tasks and data as close as possible in modern architectures. We present the syntax and examples of the proposed features in this paper, and hope to enable further discussion of useful language features to keep OpenMP scalable in emerging architectures.

## 1 Introduction

OpenMP is still based on flat memory assumption that does not match with the modern architectures. Moreover, the memory system in modern architectures are increasingly becoming hierarchical and complex. For example, AMD HyperTransport, Intel QPI, IBM Power and Sun Niagara processors based systems have complicated memory systems with private and shared cache, data interconnect switch, memory controllers, which create non-uniform memory access (NUMA) latencies. Without considering the NUMA effect of these systems, OpenMP may not scale well when the number of cores/processors keeps increasing to medium size or even more. Managing data layout and maintaining task and data affnity are essential factors to achieve scalable performance on such systems. We believe it is necessary to introduce data locality feature into OpenMP nowadays. We cannot rely on compiler to do the work for us in the near feature since it focuses on instruction optimizations, but not too much to data layout so far. Programmers need to manage data layout and align task to data as close as possible. However, they cannot do it with current OpenMP standard.

Our goal of the paper is to introduce data locality features into OpenMP and allow OpenMP developers to manipulate data location in the hierarchical

*This work is supported by NSF.

memory systems of modern architectures. Furthermore, this work is to provide mechanisms to developers to map OpenMP tasks with data as close as possible, which is the key to achieve scalable performance. In this work, we present the following new features to OpenMP.

1. Define the concept of location, discuss how it is used with current OpenMP and how to map locations with hardware.
2. Define data layout among the locations.
3. Define task location feature to map tasks with data based on the above two features.

We describe the syntax of the proposed location and data layout in Section 2, present our initial implementation in Section 3, illustrate the performance difference with different data layout in Section 4, as well as discuss the related work in Section 5 and conclusion in Section 6.

## 2 Proposed Features

In this section, we introduce the syntax and implication of our proposed features on data locality in OpenMP. We also give some code examples to illustrate how to use these features.

### 2.1 Introducing Location

As we know, one of key points to achieve good performance is to align tasks together with their data as close as possible. However, OpenMP programs rely on runtime or OS to bring data to task or vice versa. In this section, we introduce a new feature called "*location*" into OpenMP so that programmers are able to control where data is stored as well as to manage tasks associated with their data.

#### 2.1.1 Definition of Location

The location concept is adopted from X10's Place [6] and Chapel's Locale [7] notations. We give the definition of location in OpenMP as follows.

**Location:** *Location is a logical execution environment that contains data and OpenMP implicit and explicit tasks.*

A location is a logic concept that can be viewed as a collection of tasks and data. It is mapped at runtime onto hardware that includes computing resources and shared memory, which we assume are close to the computing resources. It is mapped at runtime with specified hardware, and they may be heterogeneous, such as shared-memory nodes, NUMA nodes, or accelerator devices. The

OpenMP tasks allocated on a location may be executed concurrently. We assume that tasks running on a location accessing its local memory faster than the memory on other locations.

As a first attempt, we confine our definition of locations for homogeneous systems only. We introduce a parameter `NLOCS` as a pre-defined variable for the number of places in an OpenMP program. It is similar to the `THREADS` parameter defined in the UPC language [3]. The parameter keeps constant during a single run. It will help the definition of data distribution later, as well as compiler and runtime implementation. Each location is uniquely identified with a number `MYLOC` in the range of `[0:NLOCS-1]`.

The runtime environment variable `OMP_NUM_LOCS` defines the number of locations, similar to the number of threads in the current OpenMP specification. If the environment variable is not set yet, the runtime will detect the topology of hardware and determine the number of locations based on its memory hierarchical. For example, Linux based NUMA platform relies on libnuma to detect the number of NUMA nodes, each of which contains a set of CPUs and has the same memory latency to access the local memory of the node.

#### 2.1.2 Syntax of Location

At our first design, we limit the location usage as clause associated with OpenMP `parallel` and `task` directives. The clause may be used together with task groups in the future. The syntax of location clause is as follows:

```
  location(m[:n])
  integer: m, n
```

where $m$ and $n$ are two integer numbers from 0 to `NLOCS`-1. A single number "$m$" represents the location number where the associated OpenMP construct will be executed on. Two numbers separated with a colon, such as $m$:$n$, represent a range of locations where the associated OpenMP construct will be executed on. $m$ and $n$ are the lower bound and upper bound of the range.

The location clause indicates that the associated OpenMP constructs are executed in the specified location / a range of locations. The following are two examples of using the location clause.

```
Example 2.1:
  #pragma omp parallel location(0:4)

Example 2.2:
  #pragma omp task location(1)
```

The first example (2.1) is to use the location clause associated with OpenMP parallel region, which indicates that the parallel region will fork the number of threads in locations from 0 to 4 and executed its enclosed implicit tasks on these locations. The second example (2.2) indicates the OpenMP task will be executed on location 1.

### 2.1.3 Threads and Locations Mapping

We plan to use the block distribution to map threads to locations as the default rule. For example, if we have 16 threads and 4 locations, then the first location holds threads 0–3, the second location has threads 4–7, etc. If necessary, we will consider the cyclic distribution of threads to locations. We may allow users to modify the mapping rule by calling an `omp_location_policy([BLOCK,CYCLIC])` runtime routine. If `CYCLIC` is specified, the threads will be mapped with locations in a cyclic fashion, i.e. threads 0, 4, 8 and 12 are placed on location 0, thread 1, 5, 9 and 13 are placed on location 1, and so on in the above example.

### 2.1.4 Location Inheritance Rule

The location inheritance rule for those parallel regions and tasks without the "`location`" clause is hierarchical, that is, it is inherited from the parent. In the beginning of a program execution, the default location association is all locations. Thus, when there is no location associated with a top-level parallel region, the parallel region will be executed across all locations in a block distribution fashion for all threads if possible.

If a task has been assigned to a location, all of its child tasks will be running on the same location if no other location is specified. On the contrary, if a location is specified to one of its child tasks, the task will be executed on the specified location. If a location number specified by programmer does not exist during runtime, for example, location 4 specified but there are only 2 locations for the execution, then these tasks specified running on the location will generate an exception and be handled by an error handler or abort.

### 2.1.5 Examples

In principle, instead of relying on compiler to analyze where data is and map tasks with data as HPF does, we plan to follow OpenMP design principle to let the programmer decide where to run a task.

For the current OpenMP, we can assume that a parallel region starts with only one location. After the locations are defined, the parallel region starts with allocating threads to all locations in the block distributed fashion. It is possible to assign a parallel region or a task into another location, even if the location is not used for the current parallel region.

The following is a legal example for nested parallelism. The task and inner parallel region can be mapped to a location that may or may not in location 0–4.

```
Example 2.3:
 #pragma omp parallel location (0:4)
 {
   #pragma omp task location(4)
   {
     Task(A[i])
   }
   #pragma omp parallel location(5:7)
   {
     ...
   }
 }
```

To achieve good data locality in nested parallelism, we can use the MYLOC parameter to determine the current thread location and starts the inner parallelism within the same location. The following example shows a scatter threads distribution at the outer level of parallelism, and a compact threads distribution at the inner level of parallelism.

```
Example 2.4, nested parallelism:
  #pragma omp parallel  // spread to all locations
  {
    ...
    #pragma omp parallel location(MYLOC)
    //starts the inner parallel region within its parent location
    {
      ...
    }
  }
```

In fact, we do not even need to specify location for the inner level parallel region since it will be inherited from its parent. It is quite simple and natural to achieve good data locality using location, similar to what our Subteam proposal [5] designed for.

## 2.2 Defining Data Layout

With the concept of location, we can further introduce the way to express data layout into OpenMP. The goal of this feature is to allow OpenMP programmers to control and manage the data layout and map with the hierarchical memory systems as they wish. The data layout attribute is applied to shared data and needs to be specified right after where the shared data is declared. The data layout does not change during the lifetime of the entire program. In another word, we do not consider data redistribution/migration at this time.

### 2.2.1 Data Layout

We borrow the data distribution syntax to express data layout. It is subject to change if necessary. The data distribution is defined as a directive in OpenMP and its syntax is as follows.

```
Syntax:
  #pragma omp distribute(*/BLOCK [, */BLOCK]: variable-list) \
            [location(m:n)]
```

The variables in the list should have the same dimension declaration as the distribution specified. The symbol $*$ means that the indicated dimension keeps intact, while `BLOCK` indicates that the dimension will be distributed across a list of locations. The location clause indicates a list of locations for the data to be distributed. If location is not present, it means all locations. We only include the BLOCK data distribution at this time, and will consider other types of data distribution and impacts in the future. The distributed data still keeps its global address and is accessed in the same way as to other shared data. If no data distribution is specified for a shared variable, it is allocated in the shared memory space and it follows the current OpenMP implementation, mostly likely using a first touch policy. The only difference between distributed data and non-distributed shared data is that user controls the physical locations of the distributed data so as to improve data locality in OpenMP programs. In our current design, the data is only distributed on the main memory (associated with CPU), not to any accelerator devices.

In the following example, we distribute a shared array $A$ in a block distribution to all locations.

```
Example 2.5:
  double A[N];
  #pragma omp distribute(BLOCK: A) location(0:NLOCS-1)
```

### 2.2.2 Location Private Data

We also introduce a "LocationPrivate" data attribute that indicates a list of variables to be private to a location, but they are shared among tasks executed on the location. "LocationPrivate" is a clause used together with OpenMP parallel and task directives, or can be used as a standalone directive as well.

```
Syntax as directive:
  #pragma omp LocationPrivate(variable-list)

Syntax as clause:
  LocationPrivate(variable-list)
```

In the following example, we use LocationPrivate as a directive to define a portion of $A$ allocated in each location. It is similar to the example 2.5 in term of memory usage that uses distribute directive to distribute $A$, while in this example, the size of $A$ is only `N/NLOCS` and it is private to tasks running on other locations. In another word, the array $A$ in example 2.5 is in global address space, while the array $A$ in exmple 2.6 is in local address space.

```
Example 2.6:
  double A[N/NLOCS];
  #pragma omp locationprivate(A)
```

### 2.2.3 Mapping Tasks with Data

To achieve greater control of task-data affinity, we can map OpenMP implicit tasks (from parallel region) and explicit tasks to locations based on either the location number or the association with distributed data. To map a task with a location, one can simply specify the location number using the "`location`" clause. However, it is much more intuitive to use a distributed data element location to determine where to run a task, instead of using the location number directly. For this purpose, we define the "`OnLoc`" clause that maps a task with specified data, which assigns a task to a location where the specified data is located.

```
Syntax:
  OnLoc(variable)
```

Only distributed variables are allowed in the `OnLoc` clause. The variable can be either an entire array for the `parallel` construct or an array element for the `task` construct.

The following example illustrates how to use the distributed array $A$ to map tasks with locations for a parallel region. The implicit tasks generated in the parallel region will be executed on a list of locations that the variable $A$ is distributed to. However, these implicit tasks still follow the OpenMP scheduling (`static` in this example) and are distributed over these OpenMP threads binded on the list of locations. It does not specify how a task can be tied to the location where its data is stored. In another word, the owner compute rule does not apply to the case.

```
Example 2.7:
  #pragma omp parallel for OnLoc(A)
  for(i=0;i<N;i++)
  {
    foo(A[i])
  }
```

The following example illustrates how to map a task to a location where $A[i]$ is located. In this case, the programmer is responsible to define where the task will be executed by using the predefined location where $A[i]$ is stored.

```
Example 2.8:
  for(i=0;i<N;i++)
  {
    #pragma omp task OnLoc(A[i])
    {
      foo(A[i])
    }
  }
```

Comparing with the `location` clause, `OnLoc` is to map a task or tasks to a location or a set of locations based upon where a variable is located or distributed. The `location` clause uses explicit location number(s), while the `OnLoc` clause derives location information from the distribution of a variable, which is more closely related to the task-data affinity.

### 2.3 Runtime Functions

We plan to introduce `omp_get_myloc()` for a task to query its execution location. The use of the parameter `MYLOC` as a unique location number to identify a thread running location might be more convenient for programming purpose. We may allow users to modify the mapping of threads to locations by the runtime routine `omp_location_policy([BLOCK,CYCLIC])`. Other runtime supports are under consideration. For example, we may provide runtime functions to allow programmers to query the memory hierarchical and get a neighbor location (`get_myloc_neighbor()`), or get a list of locations sorted by their memory access distance to a specific location (`sort_locations()`).

### 2.4 External Deployment Tool

In our proposal, we do not introduce any hardware information directly into source code, but introduce a logical concept for locations. In order to map the logical locations to underneath hardware, especially on a large scale system, we believe that an external deployment tool is useful. We will introduce `omprun` external tool similar to `mpirun` used in launching MPI programs to deploy OpenMP programs and map with hardware execution environment. The tool will be able to allow users to describe the machine architecture information and configure how to map locations with hardware. It will interact with OS/job schedulers to allocate the required resources to OpenMP programs. In the future, we also plan to explore how to describe the different machine hierarchies with uniform format.

## 3 Implementation

We are currently under development of the proposed location feature in the OpenUH compiler[10]. The OpenUH compiler is a branch of Open64 compiler and is used as a research infrastructure for OpenMP, compiler and tools research at University of Houston. It supports C/C++, Fortran 77/90 with complete OpenMP 2.5 and partial OpenMP 3.0 features.

### 3.1 Runtime

We are extending our OpenMP runtime to support location and data layout management. We are currently building our runtime on top of libnuma to implement thread binding and location creation, and look for portable library support in next step. We first detect the number of locations specified by environment variable `OMP_NUM_LOCS`, as well as the CPU set and the number of numa nodes allocated to an OpenMP program. When an OpenMP program is launched, OS or job scheduler software allocates the computing resources to it, which determines the available CPU set. Once the CPU set is determined, the runtime maps the number of locations with the set of CPUs based on the distance between these CPUs. It may not be straightforward to determine the distance in different platforms, and we rely on existing runtime libraries to calculate the memory latency between different CPUs. For example, the latest `libnuma` on Linux has a function to calculate the distance between NUMA nodes, which can be used to determine the affinity of locations and NUMA nodes. Sun Solaris Lgrp library [11] also has the similar functions to return the different latencies between hardware resource groups. The runtime calculates the number of NUMA nodes contained in a location, and then put the neighboring nodes together and allocate them as one location. If the number of locations is larger than the number of NUMA nodes, then it will map a NUMA node with multiple locations. However, the number of locations should never exceed the number of OpenMP threads. The location creation and thread binding are done at the initialization of the first OpenMP parallel region.

The data layout is managed by the runtime as well. We use `libnuma` to store the distributed data in the local memory of each location as much as possible. We also create an additional data structure for a distributed array to store its local index and global index, and create `get/put` functions to access the array. The key point is to keep the data access overhead as low as possible.

### 3.2 Compiler Translation

The compiler translation implementation is currently still under development. The compiler handles the syntax parsing at its front end and translates the OpenMP directives and clauses to multithreaded code based on the runtime. The challenge of the compiler translation is how to generate efficient code without sacrificing the existing compiler optimizations. For example, the data layout feature requires the compiler to change a global array to distributed dynamic allocated pointers, which may jeopardize the compiler loop nested optimizations. The translation between global index and local index of such an array will create additional overhead. We are implementing the translation at a late stage after compiler performs loop nested optimizations to reduce the side effects to compiler optimizations.

## 4 Experiments

The ultimate goal of these proposed extensions is to improve the scalability of OpenMP performance on modern architectures. Since the implementation is not completed yet, we are not able to demonstrate the performance gains using sufficient number of benchmarks. We, however, revised the NPB BT benchmark

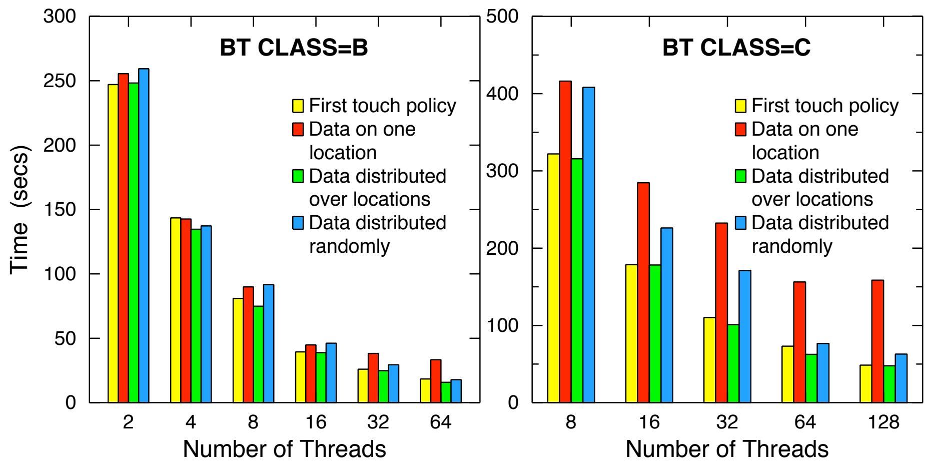

Figure 1: Performance Study of the NPB BT benchmark

manually and run it on NASA Columbia Altix NUMA system to illustrate the performance differences using different data layout.

Fig. 1 shows the performance differences of BT Class B (left) and BT Class C (right) by using different data layout. The first column shows the performance without using the proposed location feature. In this case, the performance is good since we applied thread binding and relied on the first touch policy to manage data layout. The rest of three cases use the proposed location feature with different data layout. The second column shows the worst performance where we allocated all data into one location. The third one shows the best performance where we distributed all data evenly over all locations and follow the data access pattern closely. The last one shows the performance of the case when we distributed data evenly but did not follow the data access pattern. In this experiment, we can see that the performance varies significantly with different data layout. Although we did not observe significant performance gain comparing with the case using the first touch policy, we believe that it is because we did not map the data layout with the data access pattern optimally, and it can be further improved when we do so after the compiler implementation is completed.

## 5 Related Work

In contrast to HPF[4], our design of the data location feature is following the same design principle in OpenMP, e.g. we let programmers decide and control the data layout, allocate work align with data, while the compiler just follows the programmers' decision, instead of applying sophisticated analysis to make conservative decision.

All HPCS languages allow users to associate computation with data in a more abstract way. Chapel[8] introduces “locales” to represent a unit of the parallel architecture (e.g. a node of a cluster). It allows users to distribute a domain and associated data to the locales. If a parallel loop iterates over a domain, this will distribute the iterations to the locales. Standard distributions are provided via an extensible library. Fortress[1] has also adopted a library approach to support data mappings; it distributes an array by default. X10[6] provides “places” to allow users to specify an affinity between data and activities, an abstraction of threads. Places may be mapped to physical locations at the deployment stage; this mapping may be changed during execution. Regions, collections of array elements, may be distributed in block or cyclic fashion. PGAS languages such as UPC[3], CAF[9] have data distribution feature too, but they target to distributed memory systems. Portable Hardware Locality (HWLOC) [2], formerly called `libtopology`, is a useful library that provides a portable abstraction of the hierarchical topology of modern architectures. Qthreads[12] is a light weight thread library with data locality awareness and distributed data support. We are exploring these two libraries and may consider to build our OpenMP runtime on top of them to provide portable OpenMP runtime with data locality features.

## 6 Conclusion

In this paper, we introduce the concept of “location” and the syntax to express data layout over the locations in OpenMP. We believe it would be a useful feature to scale OpenMP to many-core, medium size of SMPs. This feature has potential to be extended to heterogeneous systems with CPU and accelerators, as well as for distributed memory systems in the future. We may also consider how to express data movement/redistribution among locations in the future.

## Acknowledgments

We would like to thank to Piyush Mehrotra at NASA Ames Research Center for valuable discussion and feedbacks from OpenMP ARB meeting for the proposed extensions.

## References

[1] E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, S. Ryu, G. S. Jr., and S. Tobin-Hochstadt. The Fortress language specification, version 0.785, 2005.

[2] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-*

*Based Processing (PDP2010)*, Pisa, Italia, Feb. 2010. IEEE Computer Society Press.

[3] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and language specication. Technical report, Center for Computing Sciences, May 1999.

[4] B. Chapman. HPF features for locality control on ccNUMA architectures. In *HUG2000: The 4th Annual HPF User Group meeting*, October 2000.

[5] B. M. Chapman, L. Huang, G. Jost, H. Jin, and B. R. de Supinski. Support for flexibility and user control of worksharing in OpenMP. Technical Report NAS-05-015, National Aeronautics and Space Administration, October 2005.

[6] P. Charles, C. Donawa, K. Ebcioglu, C. Grotho, A. Kielstra, V. Saraswat, V. Sarkar, and C. V. Praun. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 519–538. ACM SIGPLAN, 2005.

[7] R. Diaconescu and H. Zima. An approach to data distributions in chapel. *Int. J. High Perform. Comput. Appl.*, 21(3):313–335, 2007.

[8] C. Inc. Chapel specification.

[9] R. W. Numrich and J. K. Reid. Co-Array Fortran for parallel programming. *ACM Fortran Forum*, 17(2):1–31, 1998.

[10] The OpenUH compiler project. `http://www.cs.uh.edu/~openuh`, 2005.

[11] I. Sun Microsystem. Memory and thread placement optimization developer's guide. `http://dlc.sun.com/osol/docs/content/MTPODG/lgroups-2.html`, 2007.

[12] K. B. Wheeler, R. C. Murphy, and D. Thain. Qthreads: An api for programming with millions of lightweight threads. In *IPDPS*, pages 1–8. IEEE, 2008.