Scientific Programming 18 (2010) 169–181
DOI 10.3233/SPR-2010-0307
IOS Press

Enabling locality-aware computations in OpenMP *

Lei Huang [a], Haoqiang Jin [b], Liqi Yi [a] and Barbara Chapman [a]
[a] *University of Houston, Houston, TX, USA*
[b] *NASA Ames Research Center, Mountain View, CA, USA*

Abstract. Locality of computation is key to obtaining high performance on a broad variety of parallel architectures and applications. It is moreover an essential component of strategies for energy-efficient computing. OpenMP is a widely available industry standard for shared memory programming. With the pervasive deployment of multi-core computers and the steady growth in core count, a productive programming model such as OpenMP is increasingly expected to play an important role in adapting applications to this new hardware. However, OpenMP does not provide the programmer with explicit means to program for locality. Rather it presents the user with a "flat" memory model. In this paper, we discuss the need for explicit programmer control of locality within the context of OpenMP and present some ideas on how this might be accomplished. We describe potential extensions to OpenMP that would enable the user to manage a program's data layout and to align tasks and data in order to minimize the cost of data accesses. We give examples showing the intended use of the proposed features, describe our current implementation and present some experimental results. Our hope is that this work will lead to efforts that would help OpenMP to be a major player on emerging, multi- and many-core architectures.

Keywords: Locality of computation, memory programming, OpenMP, programming models

1. Introduction

It has long been recognized that one of the key means to achieve good performance on large-scale non-uniform memory access (NUMA) computers, including cache-coherent distributed memory platforms, is to coordinate the mapping of computations and data across the system in a manner that minimizes the cost of data accesses. Many of today's desktop systems are multi-core NUMA computers. Application performance on such systems also depends to some extent on the relative placement of computations and data. As the number of cores configured increases, the need for consideration of locality in a parallel application targeting multi-core systems will grow. Moreover, the trend towards building systems with heterogeneous cores, with a high cost of data transfer between memories associated with different kinds of cores, will further drive the need to carefully consider the mapping of an application's data and computation to a given platform. We further note that careful data mapping will be required to help reduce energy consumption on large-scale resources.

OpenMP is a collection of compiler directives and library routines that was developed to support the creation of shared memory parallel programs. The API is owned by the OpenMP Architecture Review Board (ARB), which defines the language specification and debates enhancements to its feature set. The directives, which have bindings for C, C++ and Fortran, may be inserted into an existing application in order to enable it to exploit multiple processors or cores in a shared memory system. They enable the application developer to identify regions of the code that should be executed by multiple threads and to specify strategies for assigning work in those regions to the threads. Moreover, data may be shared among the threads or it may be privatized, in which case each thread has its own individual copy of the data. Whereas the application developer provides the parallelization strategy in this manner, it is up to the OpenMP implementation to determine the details of the parallel execution. OpenMP-aware compilers translate the directives into an explicitly multithreaded code that makes calls to its runtime system to manage the subsequent execution. This runtime sys-

*This material is based upon work supported by the National Science Foundation under Grant No. CCF-0833201 and CCF-0917285 as well as the Department of Energy under Grant No. DE-FC02-06ER25759.

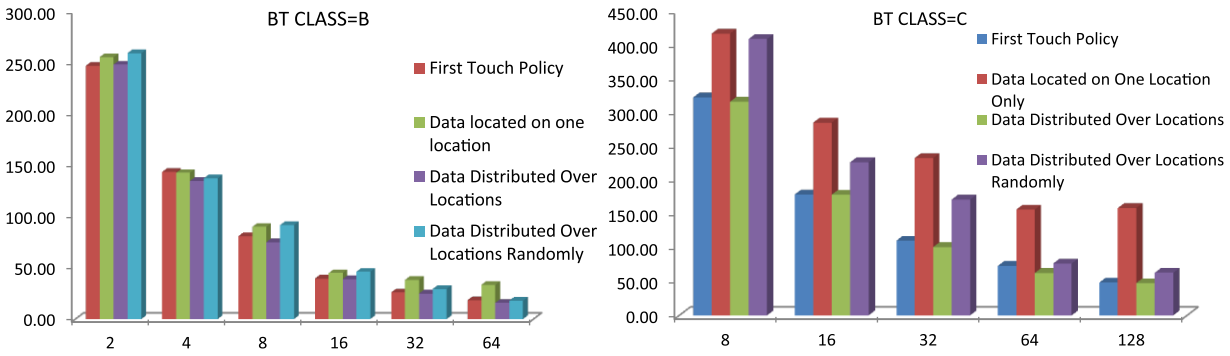

Fig. 1. Performance study of the NPB BT benchmark. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2010-0307.)

tem will typically include routines to start up threads and to manage them during the course of a program's execution. Most often it relies on the operating system to perform crucial services, including assigning work associated with OpenMP's user-level threads to hardware resources, and allocating memory for the shared and private data. Hence OpenMP programs rely on the runtime or OS to bring data to a task or vice versa.

OpenMP enjoys wide vendor support on today's shared memory architectures. However, it does not provide features to enable explicit programming for locality. Thus it is generally not known in advance where a piece of work, such as a set of loop iterations or a task, will be executed. Similarly, the location of data objects is typically not known a priori. Yet even on a small multi-core platform the relative placement of data and computations may lead to clearly observable variations in execution time. Figure 1 shows the performance differences between several executions of the NPB benchmark BT Class B (left) and BT Class C (right) that arise as the result of different data layouts on up to 128 processors of the SGI Altix, a distributed shared memory computer. The first column shows the program performance when the default "first touch" policy is applied: in this case, the system attempts to store data so that it is local to the thread that first accesses it. In this case, we have analyzed the program's memory access pattern and then applied the first touch policy to implicitly distribute data at the beginning of the program in an appropriate manner. Since the program has a fixed memory access pattern, the first touch policy achieves good performance. The second to fourth columns display the performance achieved when applying different data distributions. The second one allocates data into one location only (the concept of location will be introduced later), the third one distributes data evenly over all locations and the fourth distributes all data randomly over all locations. The performance varies significantly due to the different data layouts.

Given the fact that OpenMP assumes a flat memory (uniform memory access), while memory systems are increasingly hierarchical and NUMA, we believe that it is important to introduce features for explicit locality control into OpenMP. Our goal is to develop an approach to introduce a pervasive locality concept in OpenMP. We moreover aim to design language features that may be used in conjunction with this concept to enable a scalable OpenMP-based programming model that will cater not only to the needs of emerging multi-core and many-core platforms, but may form the basis of a high-level programming model for heterogeneous systems. We envisage a notion of locality that may even help shared memory computations to scale to satisfy the demands of emerging petascale and future exascale architectures.

In this paper, we proposes the following additional features to OpenMP.

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

The paper is organized as follows. We discuss the relationship between OpenMP features and the locality of computations in the next section below. We then describe the syntax of the proposed location and data layout in Section 3, and present our initial implementation in Section 4. The results of several experiments that illustrate the performance differences observed based

upon different data layouts are given in Section 5. Related work is briefly discussed in Section 6 and finally, we draw some conclusions in Section 7.

2. Locality in OpenMP programs: Past and present

Traditionally, high-level shared memory programming models have encouraged the programmer to ignore any structuring of a system's memory and data access time inequalities. This encourages truly platform-independent programming since the application developer may focus on describing the parallelism of the application, determining what data needs to be shared and when this sharing should take place. Moreover, many early shared memory architectures did indeed provide uniform memory access, so that the impact of data placement could safely be ignored. OpenMP is built on top of prior work that provides for this kind of programming style [12].

Nevertheless, the need to consider data locality in OpenMP programs has been around almost as long as the standard itself. One of the early platforms that was used to run OpenMP code was the SGI Origin [13] a distributed memory system that provided global memory and cache coherency at relatively low cost across the nodes of the machine. Since the relative placement of computations and data could have a major impact on performance of OpenMP applications on this system, SGI defined several extensions to OpenMP that were designed to allow the user to provide information related to locality. They created two different notations for specifying data distributions, a means for describing how the data should be mapped across the hardware. They also introduced an affinity clause that could be appended to loop nests. One form of SGI data distributions was a hint to the system to describe how pages of data should be spread across the nodes of the DSM architecture that SGI provided. The other kind was an HPF-style data distribution that specified a precise element-wise mapping of data to the system. In addition to these, SGI provided a default policy to be applied when the application developer did not specify explicit distributions: this was the first touch policy deployed in the example above, where data will be migrated at the page level to the memory associated with the location executing the first thread that accesses it.

The first touch policy works well if a program does not change the data access pattern during its execution. If no data is available in the corresponding local memory, this can become inefficient, however. The next touch policy has been proposed to address a shortcoming of the first touch policy; the idea is to allow data to be redistributed during program execution. However, both first/next touch policies operate at the page level, which does not provide precise data distribution. Moreover, it relies on the manner in which threads access data and also on the mapping of the thread computations. However, tasking was introduced in OpenMP 3.0 and is expected to be heavily used. Tasks are scheduled dynamically and there is no means to influence how (where) they are scheduled, nor to influence the memory location of the data that they access. This may greatly affect the usefulness of applying a first/next touch policy. Explicit data layout needs to be introduced to OpenMP in order to enhance the performance of OpenMP tasking. Unfortunately, however, the manner in which it was introduced by SGI, and proposed by other vendors, does not facilitate compiler optimizations. For this reason, we need to take a fresh look at how this can be accomplished in the OpenMP context.

3. Proposed features

In order to achieve good performance on multi-core platforms, one needs to pay a great attention to the memory access. Some programs may get better performance if data is allocated as close as possible to tasks since the data access has small latency; while others may get better performance if data is spread out to avoid memory contention and to better utilize the memory bandwidth. It requires the developers to be able to manipulate the data locality and thread affinity. For emerging platforms including heterogeneous systems, it may even be more important to avoid moving data around due to the high latency.

However, OpenMP does not support thread and data affinity, and relies 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 will be able to control where data is located as well as to manage tasks to be executed close to their associated data. We also present some code examples to illustrate how to use these features.

3.1. Definition of location

The location concept is adopted from X10's Place [7] and Chapel's Locale [8] notations. We give the definition of location in OpenMP as follows.

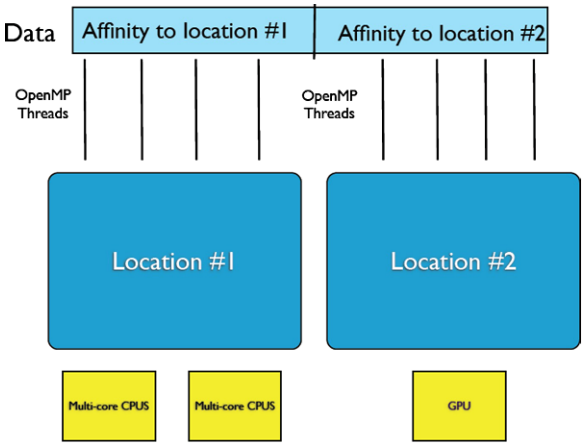

Fig. 2. Two locations map with different hardware resources. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2010-0307.)

Location. *Location is a logical execution environment, where OpenMP implicit and explicit tasks are executed and data is stored.*

Location provides an additional logical layer between the current OpenMP programming model and underlying hardware. Figure 2 illustrates the additional logical layer of OpenMP programs using *location* to apply thread binding and data affinity. In this case, four OpenMP threads are bound to location #1 while the other four OpenMP threads are bound to location #2. The data is still shared but they have the affinity to one of these locations. With the concept, OpenMP user can specify a parallel region mapping with a collections of locations, determine an OpenMP worksharing construct to be executed by a set of locations and allocate an OpenMP task on a specific location. By specifying the locations in a program, user can control where a task is executed, bind threads with hardware and set data layout. User can use locations to further optimize their OpenMP code by specifying data and task affinity within a location. The fundamental assumption of the concept is that a task will be able to access data at the same location faster than data at other locations.

A location is mapped at runtime onto hardware that includes computing resources and memories. In NUMA systems, a location may map to a NUMA node or to a set of NUMA nodes. Although we experimented the concept for homogeneous systems only in this paper, we believe the concept can be applied to heterogeneous systems such as GPU/accelerator devices as well. In heterogeneous systems, a location can be mapped with a device that may have separate memory and different instruction set. The location provides users a way to allocate different kind of tasks to different devices that suit most for their computation requirements. This will be explored in the near future.

We introduce a new runtime environment variable `OMP_NUM_LOCS` that defines the number of locations, similar to the number of threads in the current OpenMP specification. If the environment variable has not been set, the program assumes one location by default. We introduce a parameter `NLOCS` as a pre-defined variable for the number of locations in an OpenMP program. It is similar to the `THREADS` parameter defined in the UPC language [3]. The parameter remains constant during an execution. The parameter is necessary in the definition of data layout, as well as in compiler and runtime implementation. Each location is uniquely identified with a number `MYLOC` in the arange of `[0:NLOCS-1]`.

3.1.1. Syntax of location

At our design, we limit the location usage as a clause associated with OpenMP `parallel`, OpenMP `worksharing` 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])
```

where m and n are two integer numbers ranging 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 construct is executed in a specified location/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 indicates how the location clause is used to associate with an OpenMP parallel region, which forks and binds the number of threads on locations from 0 to 4 and executes its enclosed implicit tasks on these locations. The second Example 2.2 indicates that the OpenMP task will be executed at location 1 only.

We consider the *location* clause as a hint, instead of a command for compiler implementation. It means that a compiler may ignore the clause, and the code result is still expected to be correct. There should not be an exceptional or error happening even when the parameters of the clause are wrong. For example, if a location number specified by programmer does not exist during runtime, e.g., location 4 is specified but there are only 2 locations for the execution, then these tasks specified running on the location will still be executed on a random location.

3.1.2. Threads and locations mapping

The default mechanism to map threads to locations is by block distribution fashion. 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, and so on. The block fashion maps threads with locations compactly to increase the data access locality. We also define the cyclic mechanism to map threads with locations, which distributes threads in a scatter fashion that can be used in the case of increasing memory access bandwidth. A user can modify the mapping rule by calling the `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.

3.1.3. Location inheritance rule

The location inheritance rule for parallel regions and tasks without the "`location`" clause is hierarchical, that is, it is inherited from the parent in term of nested parallelism. In the beginning of a program execution, the default location association is to the entire collection of 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. For nested parallelism, the inner parallel region will start by default at the same location where its parent thread is associated.

If a task has been assigned to a particular 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.

3.1.4. Examples

In principle, instead of relying on compiler to analyze where data is and map tasks with data, we follow the OpenMP design principle to let the programmer explicitly specify where a task is executed, instead of by the compiler.

After the locations are defined either by the environment variable `OMP_NUM_LOCS` or by the runtime, a parallel region starts with forking and binding threads to all locations in the block distribution fashion. It is possible to move a nested parallel region or a task into a different set of locations, even if these locations are excluded in the current parallel region.

The following is a legal example of nested parallelism. The task and nested parallel region can be mapped to a location that may or may not belong to the original set of locations #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 start 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
  //thread 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 thread according to the inheritance rule.

It is quite simple and natural to achieve good data locality using location to nested parallelism, similar to what the Subteam proposal [6] designed for.

3.2. Defining data layout

With the concept of location, we can further introduce a mechanism 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 it with hierarchical memory systems. The data layout attribute is applied to shared data and needs to be specified right after the shared data is declared. The data layout does not change during its lifetime of the entire program. In other words, we do not consider data redistribution/migration at this time.

3.2.1. Data layout

We borrow the data distribution syntax from SGI to express data layout as a directive in OpenMP as follows.

```
#pragma omp
    distribute(dist-type-list: variable-list)
    [location(m:n)]
```

"`dist-type-list`" is a comma-separated list of distribution types for the corresponding array dimensions in the `variable-list`. Variables in the `variable-list` should have the same dimension that matches with the number of distribution types listed. Possible distribution types include "`BLOCK`" for a block distribution across a list of locations given in the `location` clause and "`*`" for a non-distributed dimension. If the `location` clause is not present, it means all locations. We could consider other types of data distribution, but will not discuss them here for the sake of simplicity.

The distributed data still keeps its global address and is accessed in the same way as to other shared data in the program. With distributed data, the user can control and manage shared data to improve data locality in OpenMP programs. The following example shows how the first dimension of array A gets distributed across all locations.

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

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 following the first touch policy for data locality. 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.

3.2.2. Location replicated data

We also introduce a location replicated data attribute that indicates a list of read-only variables to be replicated to a set of locations. The location replicated data implies that a copy-in operation will be applied at the beginning of a parallel region to each defined variable. Multiple copies of the variable will be created and each of them will be allocated to one location. We define its syntax as a standalone directive in the following.

```
Syntax as directive:
#pragma omp locationreplicated(variablelist)
    [location(m:n)]
```

In the following example, we use *locationreplicated* as a directive to define a portion of A allocated in each location. A is invariant during the execution and there are replicated copies for A on each location. It is used when A is a small size of array and it is accessed repeatedly over a program. By replicating A, the program may achieve better performance due to reducing the latency of accessing it.

Example 2.6.

```
double A[M];
#pragma omp locationreplicated(A)
   location(0:NLOCS-1)
```

3.3. Mapping with locations

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. In this section, we introduce the syntax of mapping OpenMP worksharing and tasks with locations.

3.3.1. Mapping worksharing with locations

To map a worksharing construct or 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 speci-

fied data, i.e., it assigns a task to a location where the specified data is located.

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

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

The following example illustrates how to use a distributed array A as an indication to schedule a parallel loop by considering the data layout over the location. The implicit tasks generated in the parallel loop will be executed on a list of locations where the variable A is distributed to, and be scheduled according to where data is located. For example, if A[0] is located on location #1, then the iteration $i = 0$ will be executed on location #1 too.

Example 2.7.

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

The OnLoc clause changes the original OpenMP scheduling by introducing additional factor. The OnLoc clause determines how iterations are distributed to different locations, while inside each location, the original OpenMP scheduling applies. For example, in Example 2.7, the iteration space is determined first for each location, and these iterations will be further distributed to multiple threads (if any) bound to a location using static scheduling.

3.3.2. Mapping tasks with data

The following example illustrates how to map a task to a location where $A[i]$ is located. In this case, it is the programmer's responsibility to define where the task will be executed by specifying the location where $A[i]$ is stored.

Example 2.8.

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

Compared to the `location` clause, `OnLoc` maps 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.

3.4. Runtime functions

We 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 allow users to modify the mapping of threads to locations by the runtime routine `omp_location_policy([BLOCK, CYCLIC])`. Other runtime support are under consideration. For example, we may provide runtime functions to allow programmers to query the memory hierarchy and get the neighboring location (`get_myloc_neighbor()`) or get a list of locations sorted by their memory accessdistance to a specific location (`sort_locations()`).

4. Implementation

We have implemented the proposed location feature in the OpenUH compiler [14]. 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.

4.1. Runtime

We have extended the OpenMP runtime to support location and data layout management. We built the 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 dis-

tances 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 [9] 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.

4.1.1. *Mapping location with hardware resources*

To accomplish location mapping with hardware, our runtime first queries the topology of the machine, and then decides how to map the user-specified locations to the computing nodes based upon the topology. For example, in a cc-NUMA system, neighboring nodes may be grouped together if the number of locations defined is smaller than the number of available computing nodes in the system. We bind threads to locations in either block or cyclic distribution fashion. We do not bind a thread to a specified CPU/core; instead, we bind a group of threads to a location to enable further scheduling within the location. After binding, the thread, and its descendant threads, may only migrate between cpu cores within the same location. Figure 3 shows an example of thread binding and location mapping on a hypothetical NUMA machine with 16 nodes. In this case, 8 neighboring NUMA nodes are grouped together and mapped to one location; there are 16 threads bound to the location.

We use the following algorithm to figure out how to map locations with NUMA nodes when the number of NUMA nodes is larger than or equal to the number of locations.

```
1 Start with a list of all the NUMA
  nodes available;
2 for each location that contains
  k nodes
3 pick the first available node,
  say X, from the node list;
4 sort the rest nodes in the list
  by NUMA-distance from X;
```

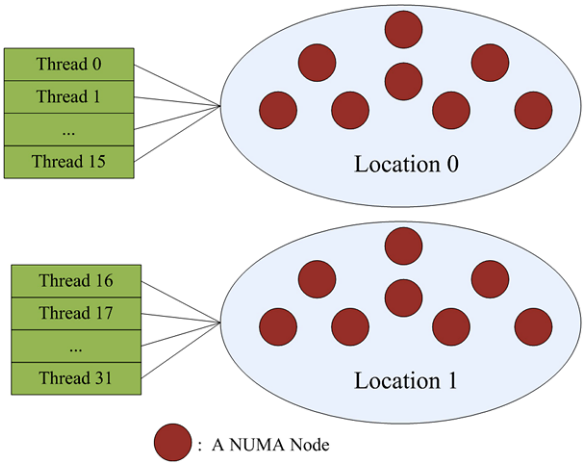

Fig. 3. Mapping 32 threads to 2 locations on a NUMA machine with 16 nodes. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2010-0307.)

```
5 pick the first k-1 node from the
  sorted list;
6 end for
```

At the case when the number of locations is larger than NUMA nodes, more than one location needs to share the same NUMA node. We map the locations to NUMA nodes as evenly as possible.

4.1.2. *Binding threads with locations*

There are two types of threads and locations binding supported in the runtime: blocked and cyclic. When we use blocked mapping, the location_id for thread #k will be determined by location_id $= k/N$, where N is the number of locations in the runtime. When we are using cyclic mapping, the location_id for thread #k will be determined by the following formula: location_id $= k \% N$.

Apparently, if the total number of threads is not an integral multiple of N, some location sill got extra thread here.

4.2. *Compiler translation*

The OpenUH 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 may require the compiler to change a global array to distributed dynamic allocated pointers, which may jeopardize the compiler loop nested optimizations.

In our implementation, we keep the global index of a distributed array intact, while using the first touch policy to move data to the local memory of different NUMA nodes. We generate an additional parallel loop to touch data at the beginning of a program to mimic the data distribution. The advantage of this approach is to keep the rest of program intact without jeopardizing any compiler optimizations. The drawback is that the data is distributed at a page granularity.

After the data distribution, the runtime keeps a table registered with the data distribution to different locations. The OnLoc clause is implemented with the data distribution function to assure that, inside a parallel loop, each thread will get the iterations which contains data item on the same location with it. This is done by searching the data and its index in the registered table. After the iteration space is determined for each location, the original OpenMP scheduling is applied to all threads inside each location.

5. Experiments

The goal of these proposed extensions is to improve the scalability of OpenMP performance on modern architectures. We have tested the implementation on an SGI Altix NUMA system (part of the NASA Columbia supercomputer) and an 48-core AMD workstation for selected NAS Parallel Benchmarks (NPB) [10]. In the following we first briefly describe some characteristics of the two selected benchmarks (BT and SP) considering the data and task affinity. We then discuss the performance results.

5.1. Benchmark implementation

Both BT and SP employ an alternating direction implicit (ADI) solver in three spatial dimensions. Each direction (x, y or z) involves five-stencil operations in the corresponding dimension (I, J or K) on the 3D data. The x solve has the shortest stride access of data in memory, while the z solve involves the longest stride access of data in memory. As a result, the z solve has the worst cache access for both local and remote data on a NUMA system.

In order to fully utilize the memory bandwidth and to align data with most of the work in loops, the OpenMP versions of BT and SP from NPB3.3 include a data touching loop for the K dimension of the data, effectively distributing the data in the K dimension. This approach works in concert with the parallelization of the K loops for both the x and y solves. However, for the z solve, the parallelization is on the J loop–data accesses and loop iterations are not aligned.

In our implementations of BT and SP with the newly proposed locality extensions, we replace the data touch loop with the `distribute` directive for the K dimension, such as:

```
   double precision u(5,0:imax,
                      0:jmax,0:kmax-1), &
                    rhs(5,0:imax,
                        0:jmax,0:kmax-1)
!$omp distribute(*,*,*,BLOCK: u,rhs)
```

then specify the "`OnLoc(u(1,0,0,k))`" clause on the `OMP DO` (K) loops to align with the data access in the K dimension. When considering the J loop in the z solve, the situation is different. If we apply `onloc` on the J loop based on the data distributed in the K dimension, the J loop would effectively be serialized. So we have adopted a data transposing approach, which involves declaring a new array distributed in the J dimension, transposing data to the new array, and performing computation on the new array. The code snippet looks like the following:

```
   double precision rhsz(5,0:kmax,
                         0:imax,0:jmax-1)
!$omp distribute(*,*,*,BLOCK: rhsz)
   . . .
   ..copy rhs to rhsz..
!$omp do OnLoc(rhsz(1,0,0,j))
   do j=1,ny
      do work on rhsz(*,*,*,j)
   end do
   ..copy rhsz back to rhs..
```

The `OnLoc` clause now aligns the J iterations with `rhsz`. To prepare for the next time step, array `rhsz` has to be copied back to the original array `rhs`.

Effectively, the above process is equivalent to a data redistribution except that the current approach always keeps two copies of the data around. In the actual implementation, the dimensions "`i`" and "`j`" of the array `rhsz` can be contracted into a 2D local array to reduce memory footprint and improve memory traffic. We need a template declaration to define the data distribution along the J dimension, as:

```
    double precision rhsz(5,0:kmax),
                     temp(jmax)
!$omp distribute(BLOCK: temp)
    . . .
!$omp do private(rhsz) OnLoc(temp(j))
```

```
do j=1,ny
  do i=1,nx
     copy of a slice of K
        from rhs(*,i,j,*)
        to rhsz(*,*)
     do work on rhsz(*)
     copy rhsz(*,*)
        back to rhs(*,i,j,*)
  end do
end do
```

`rhsz` is now a private variable and copying between `rhs` and `rhsz` is performed inside the loop nests.

5.2. Results

We used the OpenMP versions of NPB3.3 as a baseline for performance comparison. In order to examine contributions from different components described in Section 5.1, we created two versions: the first one applied the data transposing without the `OnLoc` clause (*data transposition*) and the second one applied the `distribute+OnLoc` as described (*loop affinity*). The OpenUH compiler was installed on both the SGI Altix and the 48-core AMD system. For additional comparison we also manually created a *loop affinity*$*$ version to mimic the scheduling of loop iterations to threads based on where the data resides and then used the Intel compiler to compile the translated code.

Figures 4, 5 and 6 show the percentage performance improvement of the new versions over the baseline version for the Class B and C problems at various thread counts. The "aggregate" values in the figures are those accumulated from the two components. Negative values indicate performance degradation. On the SGI Altix using the Intel compiler, we observe as much as 10% performance improvement at large thread counts

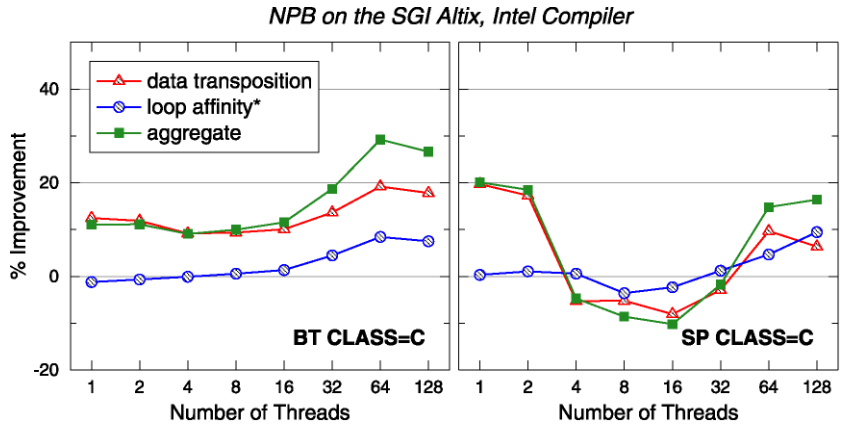

Fig. 4. Performance comparison on the SGI Altix using the Intel compiler. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2010-0307.)

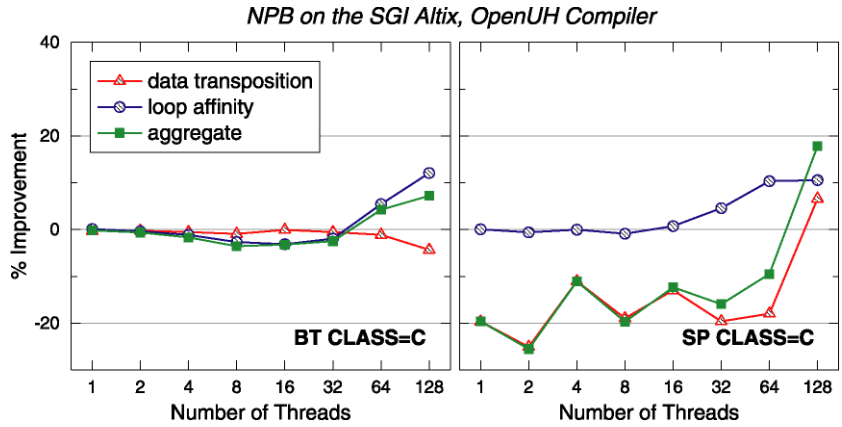

Fig. 5. Performance comparison on the SGI Altix using the OpenUH compiler. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2010-0307.)

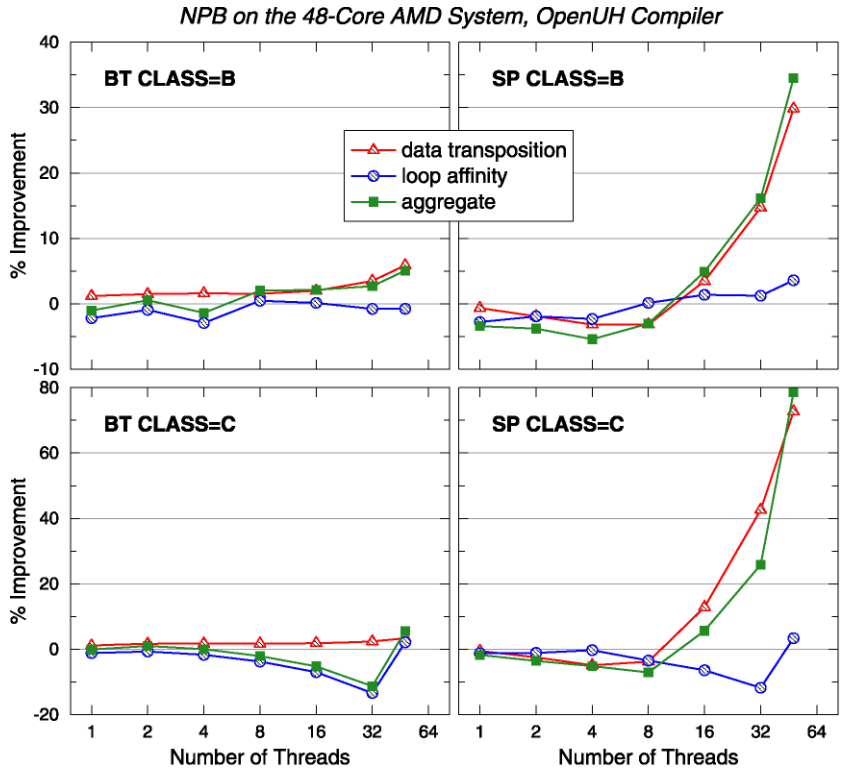

Fig. 6. Performance comparison on the 48-core AMD system using the OpenUH compiler. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2010-0307.)

from loop affinity. Data transposition improves the BT performance by 10–20%, but has variable effects on SP. This seems to be related to the balance between the cost of extra data copies and the improvement from better data alignment and cache access in the computation. The results using the OpenUH compiler (see Fig. 5) show performance improvement at large thread counts ($\geqslant$32) from loop affinity. However, we observe substantial performance degradation ($\sim$20%) from data transposition for SP and no improvement for BT.

On the 48-core AMD system (Fig. 6), there is no performance gain from applying loop affinity; in fact, negative effects are observed for the Class C problem. Such results are somewhat counter-intuitive. One possible explanation is that the current OpenUH runtime is experimental and may not handle data and thread binding optimally. On the other hand, we do observe performance improvement from applying data transposition for both BT and SP. The improvement for BT is less than 5%, but for SP it increases substantially when the number of threads is larger than 8. The larger problem (Class C) exhibits close to 80% performance improvement over the baseline version at 48 threads.

From the experiments, we observe significant performance impact from different data layouts on the NUMA system, especially for larger data sets. The notion of data layouts via `distribute` and affinity with loop iterations via `OnLoc` allows a user to carefully optimize data layout with the data access pattern and, thus, achieve performance gain on large NUMA systems.

6. Related work

In contrast to HPF [5], our design of the data location feature is following the same design principle in OpenMP, i.e., 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 [4] 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 [7] 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 [11] 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 [15] 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.

7. 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 appears to be potential enough 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.

Although the basic concept has been established, we need to further consider a number of implications of the feature for the existing OpenMP standard in addition to synchronization extensions that will give it additional power. The challenges include how to make the concept backward compatible with the OpenMP specification, and how to apply the concept to all platforms, including SMPs. Furthermore, it needs to be extended to cover hierarchies of locations and heterogeneous systems. For example, we may define a location for an SMP and then other locations for accelerators. Other questions arise with regard to how to specify their data environments and most importantly, how to enable interactions among them, since they derive much of their power from their persistence. We must also consider the implications of whether to allow threads to migrate between locations; whether to introduce the ability to dynamically change the number of locations; what happens with tasks that are not explicitly mapped to a location; how to deal with untied tasks, and whether the runtime should decide how to map locations with the underlying hardware. Moreover, the single level of locations defined in the current work is still not sufficient to map to modern hierarchical systems or very large platforms. We need to extend the concept to introduce hierarchies of locations, which may include both horizontal and vertical data layout specifications. Hierarchies may provide suitable levels of locality for very large platforms, including those with nodes that may support hundreds of thousands of threads.

Acknowledgements

We would like to thank to Piyush Mehrotra at NASA Ames Research Center for valuable discussions on this topic and are grateful for feedback from our colleagues within the OpenMP ARB on the proposed extensions.

References

[1] E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, S. Ryu, G.L. Steele 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, February 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 specification, Technical report, Center for Computing Sciences, 1999.

[4] Chapel specification, available at: http://chapel.cray.com/.

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

[6] 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, 2005.

[7] P. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, V. Saraswat, V. Sarkar and C. Von 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*, ACM SIGPLAN, 2005, pp. 519–538.

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

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

[10] NPB – NAS Parallel Benchmarks, available at: http://www.nas.nasa.gov/Software/NPB.

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

[12] Parallel Computing Forum, PCF parallel Fortran extensions, V5.0, *ACM Sigplan Fortran Forum* **10**(3) (1991), 1–57.

[13] Silicon Graphics Inc., MIPSpro 7 FORTRAN 90 commands and directives reference manual, 2002.

[14] The OpenUH compiler project, available at: http://www.cs.uh.edu/~openuh, 2005.

[15] K.B. Wheeler, R.C. Murphy and D. Thain, Qthreads: An API for programming with millions of lightweight threads, in: *IPDPS*, IEEE, 2008, pp. 1–8.