http://dx.doi.org/10.3745/KIPSTA.2012.19A.3.129

클러스터 상에서 다중 코어 인지 부하 균등화를 위한 Chapel 데이터 분산 구현

구 본 근[†] · Patrick Carpenter[††] · Weikuan Yu[†††]

요 약

클러스터와 같은 분산 메모리 구조에서 각 노드는 전체 데이터의 일부분을 저장하고 있다. 이러한 구조에서는 데이터를 각 노드에 분산시키는 방법이 성능에 영향을 준다. 데이터 분산 정책은 데이터를 노드들에게 분산시켜 병렬 데이터 처리를 실현하는 정책이다. 클러스터 관리, 확장, 업그레이드 등 다양한 요인으로 인해 클러스터의 각 노드 성능이 동일하지 않을 수 있다. 이러한 클러스터에서 노드의 성능을 고려하지 않은 데이터 분산 정책은 데이터를 각 노드에 효율적으로 분산시키지 못할 수 있다. 본 논문에서는 각 노드의 성능을 나타내는 인자로 노드에 장착되어 있는 프로세서의 코어 수를 이용하고, 이를 고려한 데이터 분산 정책을 제안한다. 본 논문에서 제안하는 데이터 분산 정책에서는 전체 코어 수 대비 노드에 장착된 코어 수에 비례하여 데이터를 노드에 분산 저장하도록 할당을 한다. 또, 본 논문에서 제안하는 데이터 분산 정책을 Chapel 언어를 이용하여 구현하였다. 본 논문에서 제안하는 데이터 분산 정책이 효과적임을 입증하기 위해 이 정책을 이용하여 Mandelbrot 집합과 원주율을 계산하는 병렬 프로그램을 작성하고, 클러스터에서 실행하여 실행 시간을 비교한다. 8-코어와 16-코어로 구성되어 있는 클러스터에서 수행한 결과에 의하면 노드의 코어 수를 기반으로 한 데이터 분산 정책이 병렬 프로그램의 수행 시간 감소에 기여하였다.

키워드 : 데이터 분산, 부하 균등화, 다중 코어, 채플

Implementation of Multicore-Aware Load Balancing on Clusters through Data Distribution in Chapel

Bongen Gu[†] · Patrick Carpenter[††] · Weikuan Yu[†††]

ABSTRACT

In distributed memory architectures like clusters, each node stores a portion of data. How data is distributed across nodes influences the performance of such systems. The data distribution scheme is the strategy to distribute data across nodes and realize parallel data processing. Due to various reasons such as maintenance, scale up, upgrade, etc., the performance of nodes in a cluster can often become non-identical. In such clusters, data distribution without considering performance cannot efficiently distribute data on nodes. In this paper, we propose a new data distribution scheme based on the number of cores in nodes. We use the number of cores as the performance factor. In our data distribution scheme, each node is allocated an amount of data proportional to the number of cores in it. We implement our data distribution scheme using the Chapel language. To show our data distribution is effective in reducing the execution time of parallel applications, we implement Mandelbrot Set and π-Calculation programs with our data distribution scheme, and compare the execution times on a cluster. Based on experimental results on clusters of 8-core and 16-core nodes, we demonstrate that data distribution based on the number of cores can contribute to a reduction in the execution times of parallel programs on clusters.

Keywords : Data Distribution, Load Balancing, Multicore, Chapel

※ This work was supported by 2010 Visiting Scholar Program of Chungju University.
† 종신회원 : Professor in Dept. of Computer Engineering at Chungju University, Korea
†† 비 회 원 : Graduate Student in Dept. of Computer and Software Engineering at Auburn University, USA
††† 비 회 원 : Assistant Professor in Dept. of Computer and Software Engineering at Auburn University, USA
논문접수 : 2011년 12월 9일
수 정 일 : 1차 2012년 2월 27일
심사완료 : 2012년 2월 27일

1. Introduction

In the fields of seismic modeling, weather forecasting, bioinformatics, medical image analysis, etc., high performance computer systems are required to get results within the pre-determined time period. Parallel processing is one way to achieve high-performance computing. Parallel systems are implemented using various architectures, such as SMP, Cluster [1], GPGPU [2], etc. Configuring parallel processing hardware, extracting the parallelism from applications, representing parallelism, and data distribution across cluster nodes are all required in order to maximize the performance of parallel systems.

Modern parallel processing systems use distributed memory systems for storing data [3]. In these systems, each node stores a portion of data in its local memory. So, load balancing through data distribution is important to enhance the performance of parallel systems because efficient data distribution can increase data parallelism and reduce communication between nodes. Load balancing is a methodology to distribute tasks and data across nodes in a cluster and maximize task parallelism and/or data parallelism in order to enhance the performance of the cluster. In general, there are two ways to balance the load at the application program level. One is that the programmer explicitly allocates tasks and data at each node to balance the load among nodes. Another is a run time system which executes applications, automatically re-distributing the workload across cluster nodes.

In parallel applications using communication libraries such as MPI [4], PVM [5], etc., the programmer explicitly distributes the load across cluster nodes. Thus, the degree of load balancing depends on the programmer's skill. Moreover, this method may not properly adapt to changes in the cluster's configuration. However, for parallel applications that are implemented with a parallel programming language such as Chapel [6] or ZPL [7], the runtime system distributes data across nodes according to a data distribution scheme specified by the application program, and lets nodes process data stored in local memory. Thus, application programmers can then focus on implementing their algorithms without having to worry about data and task distribution. Such programs can also automatically adapt to changes on cluster configurations.

The data distribution scheme is the strategy for exposing the data parallelism of a parallel program. Therefore, data distribution determines how to distribute data across nodes in cluster, and lets nodes process data stored in local memory. Data distribution in a global address space language such as Chapel additionally supports the global address space model [6, 8, 9], which lets programmers access data without having to specify the node.

Well-designed parallel programs use data distribution schemes suitable to their data access patterns. They can attain much higher degrees of locality of access than programs which do not employ appropriate data distribution scheme. Higher locality of access means that communication time for receiving and sending data is reduced, and contributes to enhancing the performance of the system. Chapel, etc. provide data distribution schemes such as *Block, BlockCyclic*, and *Cyclic* [6]. These data distribution schemes evenly distribute data across nodes in clusters, so the amount of data stored in each node is almost the same.

As data distribution schemes such as Block, BlockCyclic and Cyclic evenly distribute data across nodes, these data distribution schemes may not be efficient in clusters where all nodes are not capable of the same performance. That is, low performance nodes are allocated the same amount of data as high performance nodes. In this case, the data processing completion time in the low performance nodes is delayed because the computing capacity in the nodes is not enough. These delayed times can delay the overall completion time of parallel programs executed on such clusters.

If data distribution schemes could distribute data across nodes according to their performance capabilities, the problem previously described can be mitigated. In such a data distribution scheme, the high performance nodes in the cluster could store and process more data than the low performance nodes. This can lead to a reduction in the delay caused by nodes with less performance capability. Ultimately such a data distribution scheme could lead to a reduction in the execution time of parallel programs on such clusters.

In this paper, we describe a load balancing scheme which distributes data based on the node performance capability, which we associate with the number of processor cores. Multi-core processors are microprocessors in which multiple cores are integrated. This is an evolutionary technology, such as cache systems, pipelines, superscalar and vector processor, etc. for enhancing the performance of computer systems. Currently, Intel and AMD are racing to integrate more cores and functions into one processor chip, and have announced a timeline for developing 8-core and 12-core processors. Currently, 4-core and 6-core processors are available on the CPU market.

The reason for their racing is that the number of cores in a processor determines the number of tasks/threads concurrently executed. For example, a 4-core processor can concurrently execute 4 tasks/threads. Therefore, a processor which has more cores can concurrently execute more tasks/threads. This results in an increase in concurrent data processing. That is, the amount of data concurrently processed can be in proportion to the number of cores in a processor.

The data distribution scheme described in this paper uses the number of cores as a raw metric to quantify node performance because the number of tasks/threads that can be concurrently executed is important to maximize parallel execution. Thus, our data distribution evenly distributes data across cores in cluster nodes to balance the load. In our data distribution scheme, nodes with more cores store and process more data. However, as the amount of data per core is almost the same, the processing completion time in a node is almost the same among nodes. Therefore, the delay in completion time due to under-performing nodes can be removed or reduced. This can result in the reduction of the execution time of parallel programs. This is not the case in other data distribution schemes such as Block, BlockCyclic and Cyclic.

To evaluate data distribution based on the number of cores, we develop a new data distribution module using the Chapel language, and also implement two parallel programs using this data distribution scheme. These parallel programs are executed on a cluster, and the execution times are compared. The cluster used in our experiments consists of 6 nodes. Two nodes in the cluster contain two Intel Xeon processors recognized as 8-core processors via Intel's hyper-threading technology, officially called HT [10]. These nodes are considered 16-core nodes by the application. Four nodes in the cluster contain two 4-core Intel Xeon processors, and are recognized as 8-core nodes.

This paper is organized as follows: Section 2 describes Chapel and data distribution. Section 3 gives motivation for this work, and describes a new data distribution scheme based on the number of cores. Section 4 describes the implementation of our data distribution scheme. In Section 5, we describe the results of an empirical performance evaluation. We discuss conclusions and future work in Section 6.

2. Data Distribution in Chapel

2.1 Chapel Language

Developed by Cray Inc., Chapel is a programming language for massively-parallel systems [6]. Chapel is a global-view language which supports general parallelism and provides better separation of algorithm and data structure [9]. The meaning of the term "global-view language" is that the programmer doesn't use any special index or descriptor to access array elements stored in remote nodes. Each node is called a locale in Chapel. The programmer uses an index defined in the domain, and does not consider where the array element is stored. Furthermore, Chapel supports general parallelism such as data parallelism, task parallelism, and nested parallelism at the level of the language specification [8]. Data and task parallelism are previously described. Nested parallelism means that we can use inner parallelisms in outer parallelisms. Chapel clearly separates the algorithm and the implementation [11, 12]. This means that the programmer makes his program from a global viewpoint, not in terms of the low level implementation. The low level implementation is the responsibility of the compiler.

2.2 Data Distribution

One of the characteristics of contemporary high-performance computing systems is a physically distributed memory. In such an architecture, efficient management of locality is essential to enhance the performance of the system [3]. The global view for distributed data across nodes is also supported, and tools for programmers are provided to encourage higher productivity. Chapel can support these features via data distribution classes.

Chapel provides *Block, BlockCyclic*, and *Cyclic* distribution schemes as default data distributions. Chapel also supports user-defined data distribution strategies [6, 11]. Chapel's default distributions evenly divide all the data, and distribute each data partition across nodes. In this case, all nodes store and process the same amount of data. We refer to such data distribution schemes as *locale-even.*

For example, (Fig. 1(a)) shows code which declares a 2-dimensional array to store complex numbers. In this code, *Block* distribution is used to distribute array data across locales. A data distribution object is created in line 2 and the domain is declared via the distribution object in line 3. Then, the array is declared in line 4. (Fig. 1(b)) shows the range of the array stored in each locale when this code is executed on a 4-locale cluster. As the same amount of data is allocated to each locale in the Block distribution, each locale stores 262,144 complex numbers.

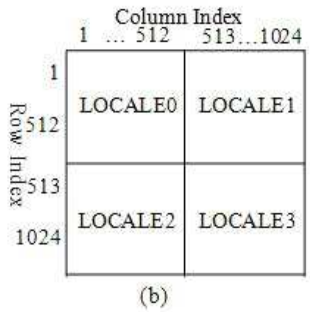
3. Data Distribution Based on the Number of Cores

3.1 Motivations

If the performance capabilities of all locales are the same, the completion times of data processing in each locale may be the same because the amount of data allocated to each locale is the same. However, if the performance capabilities of locales are not the same, the completion time of data processing in the lowest-performing locale is longer than the times in other locales. Therefore, the completion time in the lowest-performing locale determines the total time required to finish processing all the data. Data distribution schemes which account for the performance capacity of locales can enhance the performance of clusters by reducing the difference in per-locale completion times.

```
1 : use BlockDist;
2 : var dist=new dmap( new Block
          ( boundingBox=[1..1024,1..1024] );
3 : dom : domain(2 ) dmapped dist
          =[1..1024,1..1024];
4 : arr : [dom] complex;
```

(a)



(b)

(Fig. 1) Declaring a 2-dimensional array with Block data distribution in Chapel: (a) sample code and (b) portions of array allocated to each locale when the code is executed on 4-locale cluster.

There are many ways to assess the performance capacity of locales. In [13, 14], the BogoMIPS of Linux is used as the performance metric. But [13, 14] did not consider data distribution scheme and the number of cores. In this paper, we use the number of cores in a locale as the performance capability metric because the number of cores influences the number of tasks/threads concurrently executed in a locale, and processors with more cores have higher performance than processor with fewer cores (assuming similar clock frequencies). Even if many tasks are created on a locale to process data, the number of tasks concurrently executed depends on the number of cores. So, if the amount of data allocated to each locale is the same, the locale which has more cores can concurrently execute more tasks, and the time required to process the data is shorter. That is, data distribution based on the number of cores in a locale can contribute to a reduction in the execution time of parallel programs.

3.2 Data Distribution Method

The method of distributing data based on the number of cores is the following. Assume that the amount of data processed by a parallel program is D_{whole}, and the total number of cores in a cluster is C_{whole}. If data is evenly distributed to each core of this cluster, the amount of data allocated to each core is given by (1).

$$D_{core} = \frac{D_{whole}}{C_{whole}} \tag{1}$$

Assuming that the number of cores in locale i is L^i_{core}, the amount of data allocated to locale i, L^i_{data}, is given by (2). That is, data is allocated to locale i in proportion to the ratio of L^i_{core} to C_{whole}, and the amount of data allocated to each core in the locales of the cluster is almost the same. So, locales which have more cores are given more data. We refer to this data distribution scheme, and any others which evenly distribute data based on the number of cores, as core-even.

$$L^i_{data} = L^i_{core} \times D_{core} = \frac{D_{whole} \times L^i_{core}}{C_{whole}} \tag{2}$$

(Fig. 2(a)) shows the declaration of a 2-dimensional array with the data distribution based on the number of cores. *CoreEvenBlock*, which is a Chapel class for implementing the data distribution scheme based on the number of cores, will be described in the next section. This code differs from the code using Block data distribution shown in (Fig. 1(a)) at line 1. The statement at line 1 creates the instance of *CoreEvenBlock* class. Once this sample code is executed on a cluster which consists on three 8-core locales and one 16-core locale, (Fig. 2(b)) shows array regions allocated to each locale. A 16-core locale has twice the number of cores as an 8-core locale, so in our data distribution scheme, the

amount of data allocated to a 16-core locale is twice that of the data allocated to an 8-core locale.

To process data allocated to each locale in a parallel manner, statements like *forall* are required to use data parallelism supported by the system. In this case, locale i creates and executes L^i_{task} tasks concurrently. L^i_{task} is the number of tasks executed on locale i, and is given by (3). In (3), the *dataParTaskPerLocale* is Chapel's configuration constant for specifying the number of tasks created when executing a *forall* over a domain.

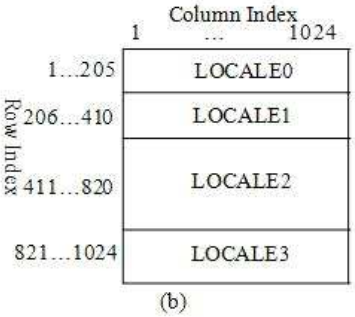
$$L^i_{task} = \max(L^i_{core}, dataParTaskPerLocale) \quad (3)$$

Basically, L^i_{task} is L^i_{core} because the user does not specify *dataParTaskPerLocale* without a special purpose, so as many tasks are concurrently executed as there are cores in the locale. At that time, the amount of data processed by task j on locale i, T^{ij}_{data} is given by (4).

$$T^{ij}_{data} = \frac{L^i_{data}}{L^i_{task}} = \frac{L^i_{data}}{L^i_{core}} = \frac{L^i_{core} \times D_{core}}{L^i_{core}} = D_{core} \quad (4)$$

```
1: use CoreEvenBlockDist;
2: var dist =new dmap( new CoreEvenBlock());
3: dom : domain(2 ) dmapped dist
          = [1..1024,1..1024];
4: arr : [dom] complex;
5: forall idx in dom do ... ;
6: forall ele in arr do ... ;
```

(a)



(b)

(Fig. 2) Declaring 2-dimensional array with our CoreEvenBlock distribution in Chapel: (a) sample code and (b) portions of array allocated to each locale when the code is executed on cluster which consists of three 8-core locales(LOCALE0, LOCALE1, and LOCALE3) and one 16-core locale(LOCALE2).

As described through (1)-(4), in our data distribution scheme, the amount of data allocated to a locale and the number of tasks concurrently executed on a locale are decided by the number of cores. Since each core executing a task processes the same amount of data, we expect the per-core processing times to be nearly the same.

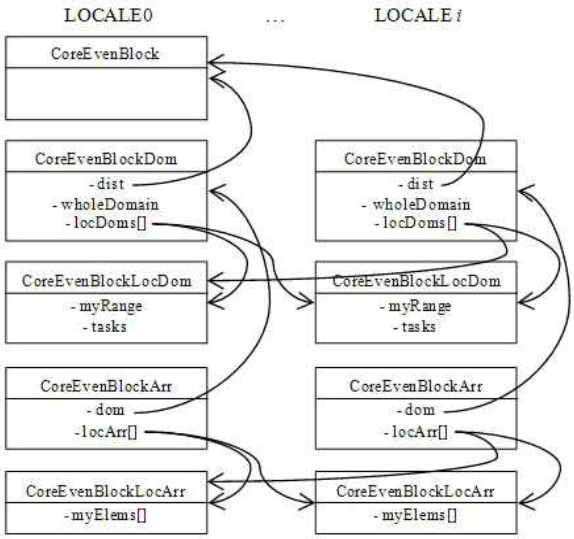
If the value of the *dataParTasksPerLocale* is greater than L^i_{core}, more tasks are created than the number of cores on a locale, and the amount of data processed by each task is reduced. However, the amount of data processed by each core executing tasks is almost the same, and the elapsed time for data processing is almost the same at the core level.

In the example shown in (Fig. 2), each 8-core locale stores 209,920 data elements and creates eight tasks. The 16-core locale stores 419,840 data elements and creates sixteen tasks. Each core on each locale in the cluster processes 26,240 data.

4. Implementation of CoreEvenBlock

To realize data distribution based on the number of cores, we describe the implementation of the *CoreEvenBlock* data distribution module in this paper. The *CoreEvenBlock* module consists of five classes, the relations between instances of which are shown in (Fig. 3).

CoreEvenBlock is the data distribution class, and inherits from *BaseDist* class. *CoreEvenBlock* does not implement all possible functions in our implementation. The runtime system uses this class to create class instances for the whole domain, e.g., *CoreEvenBlockDom* in our implementation. Instances of this class are created for the data distribution object at line 2 in (Fig. 2(a)). *CoreEvenBlockDom* stores all domain information, and partitions the whole domain among locales in the cluster. An instance of it is created on each locale when the domain is declared at line 3. At that time, the range of the domain is assigned, and this information is stored in the *wholeDomain* field. Then, the whole domain is partitioned among locales by using the number of cores. The size of partitioned domain for each locale is calculated by (2). Once the domain partitioning is complete, instances of *CoreEvenBlockLocDom* are created in all locales by using each locale's domain portion. References to these instances in locales are stored in *locDoms[]* fields in *CoreEvenBlockDom*. *CoreEvent BlockLocDom* stores the portion of the domain assigned to a locale in its *myRange* field, and the number of cores in its *tasks* field.



(Fig. 3) Relations between class instances in *CoreEvenBlock* distribution

The *CoreEvenBlockArr* class represents the array declared by element type and domain previously described, and instances of this class are created at line 4. At that time, by using the *CoreEvenBlockLocDom* instance stored in each locale, *CoreEvenBlockLocArr* instancs are created, the references to which are stored in the *locArr[]* field of *CoreEvenBlockArr*. The portion of the data corresponding to the domain stored in *CoreEvenBlockLocDom* is physically stored in its *myElems[]* field, and this is part of the whole array data.

To use data parallelism for processing data stored in arrays according to this data distribution, the *forall* statement in Chapel is used. In *forall* statements with domains like that at line 4 in (Fig. 2(a)), instances of *CoreEvenBlockDom* create as many tasks as cores. In the case of *forall* with array variable shown in (Fig. 2(a)). line 5, *CoreEvenBlockArr* instances create as many tasks as cores, and the amount of data processed by each core is D_{core} given by (1).

5. Performance Evaluation

5.1 Experimental Environment

In this section, we show that data distribution based on the number of cores contributes to a reduction in the execution time of parallel programs by evenly distributing data among cores, and can enhance the performance of clusters in terms of application execution time. For our performance evaluation, we use a cluster which consists of six locales. Four locales in our cluster are equipped with two Intel Xeon E5506 processors which have four cores. Therefore, these locales are detected as 8-core locales. Two locales are equipped with two Intel Xeon E5520 processors. This Xeon processor has 4 cores, but uses Intel HT technology, so this processor is treated as 8-core processor. So locales with two Xeon E5520 processors are detected as 16-core locales. Locales in the cluster communicate with each other via an Infiniband network.

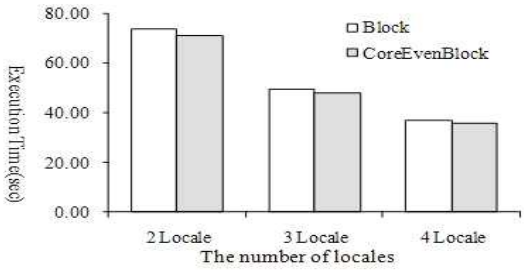
To evaluate the performance, we make two parallel programs, Mandelbrot Set [15] and π-calculation, and measure their execution times on our cluster. The Mandelbrot Set program repeatedly computes data stored in array elements, but the number of iterations is not identical for all data because the termination condition for each data element is repeatedly checked after each computation, and the iteration is terminated when the condition is matched. Therefore the elapsed execution time of each task may differ from that of each other. The π-calculation program uses a MonteCarlo [16] method, and the number of iterations for each array element is identical. So, the elapsed execution time of each task may be identical. At this point, the π-calculation program is more compute-intensive than the Mandelbrot Set program. The choice of two parallel programs with differing degrees of computational intensity is intentional.

5.2 Mandelbrot Set Parallel Program

(1) Experiment Results

The Mandelbrot Set program used in our evaluation declares a 1024x1024 array of complex numbers. To show that our data distribution is effective in reducing the execution time of parallel programs, we compare the execution times of two versions of this program. One uses *CoreEvenBlock* as the data distribution scheme, and the other uses *Block*. The programs differ only in terms of the data distribution, so the difference in execution times between them must be due to the data distribution. The execution times are measured using the *Timer* class defined in Chapel's *Time* module. Each program is executed ten times, and we use average of ten execution times to evaluate and analyze the performance.

(Fig. 4) shows the execution times in the case that the number of cores in all locales is identical. The X-axis and Y-axis represent the number of locales in the cluster and the execution times of two programs, respectively. Because the number of cores in all locales is identical, *CoreEvenBlock* distributes data across locales evenly. In this cluster configuration, the execution times of programs with *CoreEvenBlock* and *Block* distribution do not differ from each other.

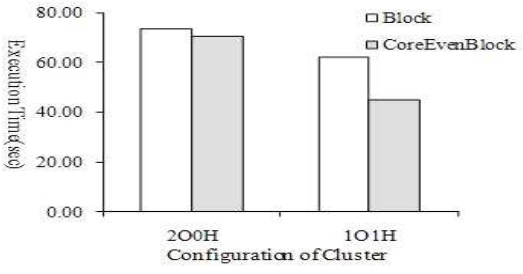


(Fig. 4) Execution Time of Mandelbrot Set Program on cluster which consists of all 8-core locales

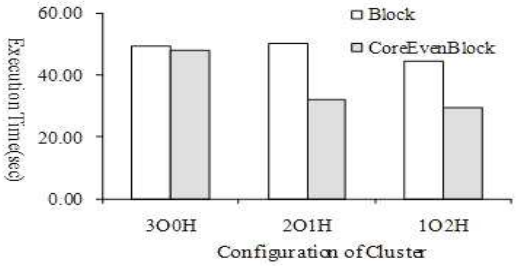
(Fig. 5) shows the execution times on a cluster which consists of two locales. The X-axis represents the configuration of the cluster. The configuration '2O0H' represents the cluster which consists of two 8-core locales. In this configuration, there is not a 16-core locale in the cluster. '1O1H' represents a cluster which consists of one 8-core locale and one 16-core locale. The Y-axis represents the execution time. In the '2O0H' configuration, the execution times of two programs do not differ from each other because *CoreEvenBlock* distributes data on locales evenly like *Block* as described above.

However, in the '1O1H' configuration, *CoreEvenBlock* reduces the execution time by about 27% compared to *Block*. The reason for this result is that *CoreEvenBlock* considers the number of cores on locale to decide the amount of data allocated to locale.

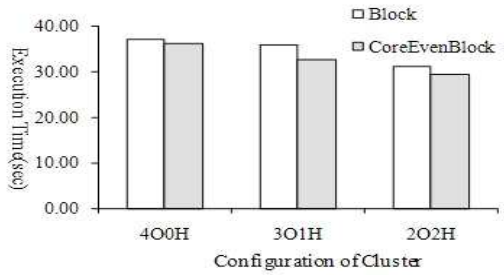
(Fig. 6) shows the execution times on cluster which consists of three locales. In '3O0H' configuration, as expected, the execution times of two programs do not differ from each other. In '2O1H' and '1O2H' configurations, *CoreEvenBlock* distribution reduces execution times by about 35% compared to that of *Block*. (Fig. 7) shows the execution times on cluster which consists of four locales. In the case of '3O1H' and '2O2H', our data distribution reduces the execution time by about 8% and 4.5%, respectively. In the case of a four-locale cluster, the performance results are similar to those of two- and three-locale clusters.



(Fig. 5) Execution Time on cluster which consists of two locales



(Fig. 6) Execution Time on cluster which consists of three locales



(Fig. 7) Execution Time on cluster which consists of four locales

(2) Contributions

In the case of clusters in which locales have the same number of cores such as '2O0H', '3O0H', '4O0H', the execution times of two programs with *CoreEvenBlock* and *Block*, respectively, do not differ from each other. However, in the case of clusters in which the number of cores in locales is not same, the execution time of program with *CoreEvenBlock* is shorter than that of program with *Block*. These results show that our data distribution scheme based on the number of cores can be effective in reducing the execution time of parallel programs.

(3) Strange Results

In *Block* distribution, data is evenly distributed to locales. Therefore, if the number of locales in the cluster is the same, the performance of the cluster may be the same regardless of the number of cores in locales. That is, the execution time of parallel program will be almost the same on clusters with '3O0H', '2O1H', and '1O2H'. As expected, the execution times on clusters with '3O0H' and '2O1H' are almost same, and that of '4O0H' and '3O1H' are also almost same. But in the cluster configuration '1O1H', '102H', and '2O2H', the execution time is reduced against our expectation. These results are common in the case of clusters in which the percentage of 16-core locales is greater than 50%.

From these results, we think the number of cores in locales influences executions of parallel programs. We are

not sure how the number of cores can influence execution but we anticipate that it influences the number of tasks created. However, *Block* distribution evenly allocates data to locales as described above. Nevertheless, *CoreEven Block* shows the shorter execution time than *Block*.

5.3 π-Calculation Parallel Program

(1) Motivations

To investigate the cause of the strange results described above, and the effect of our distribution on Intel HT technology processor once a parallel program is executed on it, we evaluate the execution times of a program which has more compute-intensive tasks. Intel HT technology enables multiple threads to run on each core because it uses processor resources more efficiently. Therefore the number of cores showed to programs and the runtime system is twice the number of physical cores.

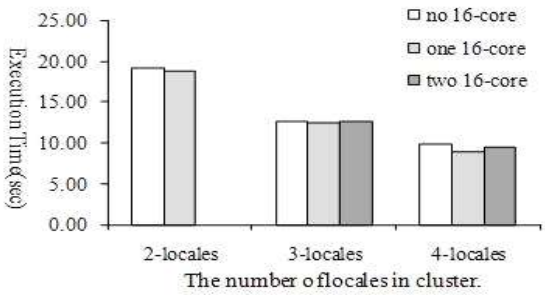
Even though Intel HT technology increases utilization and efficiency of resources, the computing capacity of the logical number of cores cannot exceed that of the physical number of cores. If our expectation is correct, the execution times of a more compute-intensive *Block*-enabled program on clusters configured by the same number of locales such as '3O0H', '2O1H', and '1O2H' are almost the same. Also, the execution times of a more compute-intensive *CoreEvenBlock*-enabled program are longer than that of *Block*-enabled program because our distribution allocates overloaded data to logically 16-core but physically 8-core locale. In this case, the utilization of processor is already very high, so Intel HT technology does not work well.

To verify our expectation, we make a π-calculation program which is significantly more compute-intensive than the one discussed earlier. To calculate the value of π, we use a MonteCarlo method.

(2) Experiment Results

In the π-calculation program implemented for our evaluation, we declare a 1-dimensional array with 1024 elements to distribute compute tasks across locales. We also use two versions of the program to evaluate the execution time. One uses *Block*, and another uses *CoreEvenBlock* as distribution. These programs are executed on clusters which have the same configurations used to execute the Mandelbrot Set program. Each program is also executed ten times, and we use the average of ten execution times to evaluate and analyze the performance.

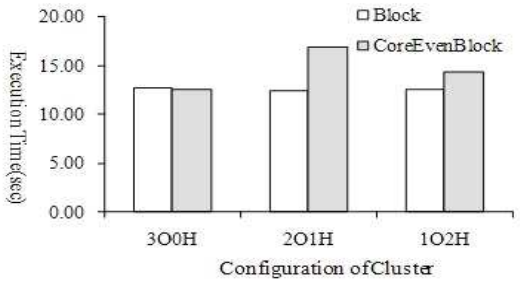
(Fig. 8) shows the execution times of *Block*-enabled program on 2-locale, 3-locale, and 4-locale cluster. As we expected, the execution times are almost the same if the number of locales on the cluster is the same regardless of the number of 16-core locales due to Intel HT.



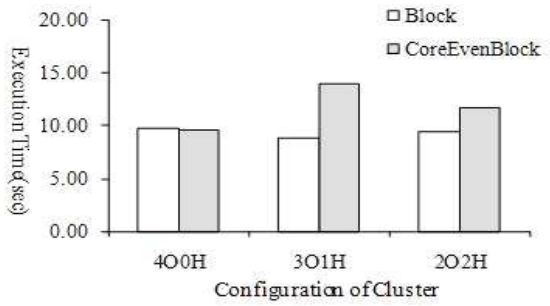
(Fig. 8) Execution Time of Block-enabled π-Calculation program on clusters

(Fig. 9) and (Fig. 10) show the execution times of programs on 3-locale and 4-locale clusters, respectively. As described above, the execution times of *Block*-enabled program are almost the same, but the execution times of *CoreEvenBlock*-enabled program are longer than that of *Block*-enabled version. The reason for these results is the following: *CoreEvenBlock* distributes data to locales by considering the number of cores. So, locales perceived as 16-core due to Intel HT are allocated data in proportion to the number of cores. But these locales have just 8 physical cores, so these locales become overloaded in the case of more compute-intensive tasks. The long data processing times on overloaded locales influence the whole execution time of the program.

In the case of '1O2H' shown in (Fig. 9), *CoreEven Block*-enabled program allocates 411 data elements to each 16-core locale. In the case of '2O1H', it allocates 511 data elements to one 16-core locale. So a 16-core locale on '1O2H' is less overloaded than on '2O1H'. Therefore the *CoreEvenBlock*-enabled program on '1O2H' has a shorter execution time than on '2O1H' configuration as shown in (Fig. 9). There exist the same results on 4-locale cluster shown in (Fig. 10).



(Fig. 9) Execution Time on cluster which consists of three locales



(Fig. 10) Execution Time on cluster which consists of four locales

As a result, the execution times are almost same if the number of locales on cluster is same regardless of the number of 16-core locale due to Intel HT. Table 1 shows the number of data elements per core and execution time when 1024 data elements are processed. The amount of data allocated to the physical core influences the execution time of more compute-intensive applications, like our π-calculation implementation. In the 3O0H configuration, each core processes about 42.66 data elements, and the execution time is about 12.64. In the 2O2H configuration, each core is allocated about 21.33 data elements by the *CoreEvenBlock*. In this case, the physical cores in locale recognized as 16-core due to Intel HT are allocated about 42.66 data elements. Thus, the execution time of the program depends on the execution times of the physical cores which process 42.66 data elements, which is about 11.72 seconds. Therefore the execution times of more compute-intensive applications depend on the amount of data allocated to the physical cores, not the locales or the logical cores

Assume that our π-calculation program, which declares 2048 data elements, is executed on a cluster which consists of two 8-core locales and two 16-core locales, 2O2H. If all cores in locales are physical, the execution time may be about 12 seconds because the number of data elements per core allocated by our *CoreEvenBlock* is 42.66. However, the number of data elements per locale allocated by *Block* is 512. In this case, the core on 8-core and 16-core locales process 64 and 32 data, respectively. Therefore, the execution time depends on that of 8-core locales, and is longer than the case of *CoreEvenBlock*. From our experimental results and analysis, our data distribution is effective on clusters which consists of locales with multiple physical core processors even in the case of more compute-intensive applications.

〈Table 1〉 The amount of data per core and execution time

Cluster	Recognized number of cores	Data per core	Data per Physical Core		Execution time(sec.)
			8-core locale	16-core locale by HT	
1O2H	40	25.6	25.6	51.2	14.30
3O1H	40	25.6	25.6	51.2	13.99
3O0H	24	42.67	42.66	-	12.64
2O2H	48	21.33	21.33	42.66	11.72

6. Conclusion

A main feature of high-performance computing architectures is the physically distributed memory. With such distributed memory, data locality may influence the performance of the system, and optimal data distribution to computing nodes is helpful in increasing data locality. It is important to maximize data parallelism based on distribution of data to nodes.

In this paper, we proposed data distribution based on the number cores in nodes-defined as locales in Chapel-and programmed the *CoreEvenBlock* module for implementing our data distribution scheme. Our implementation is based on the Chapel language. To demonstrate that our data distribution can contribute to a reduction in the execution time of parallel programs, we also implemented two programs: Mandelbrot Set and π -Calculation. From evaluation results, the contributions of our work are the following:

(1) If the numbers of cores in nodes in a cluster are not identical, our data distribution based on the number of cores can reduce the execution time of parallel programs using core-level load balancing.

(2) In the case of less compute-intensive applications, our data distribution is effective on clusters which consist of nodes equipped with Intel HT-enabled processors.

(3) As a result of our analysis, we expect that our data distribution is effective on clusters which consist of nodes equipped with physically multiple-core processors, even in the case of more compute-intensive applications.

We plan to conduct many more studies in efficient data distribution. Data distribution considering CPU cores and GPGPU cores on clusters which consist of CPU+GPU hybrid nodes can reduce the execution time of parallel programs. Also, if we can detect data access patterns of parallel programs, data distribution considering these patterns can also reduce the execution time and increase system efficiency. In the future, we will study data distribution schemes adapting to different data access patterns on CPU+GPGPU clusters.

References

[1] B. Wilkinson and M. Allen, 'Parallel Programming: Technique and Applications Using Networked Workstations and Parallel Computers', Prentice Hall, 1999.

[2] A. Sidelnik, B.L. Chamberlain, M.J. Garzaran and D. Padua, "Using the High Productivity Language Chapel to Target GPGPU Architectures," Technical Report , IDEALS, 2011.

[3] R.E. Diaconescu and H.P. Zima, "An Approach to Data Distributions in Chapel," International Journal of High Performance Computing Applications, Vol.21, No.3, pp.313-335, Aug., 2007.

[4] http://www.mpi-forum.org/

[5] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam, 'PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing. Scientifice and Engineering Computation', MIT Press, 1994.

[6] Cray Inc., *Chapel Specification,0.795 ed.*, Cray Inc., Seattle, 2010.

[7] L. Snyder, A Programmers Guide to ZPL, http://www.cs.washington.edu/research/zpl/, 1999.

[8] S.J. Deitz, B.L. Chamberlain, S.E.Choi and L. Prokowich, "Five Powerful Chapel Idioms," CUG2010, 2010.

[9] B.L. Chamberlain, D. Callahan and H.P. Zima, "Parallel Programmability and the Chapel Language," International Journal of High Performance Computing Applications, Vol.21, No.3, pp.291-312, Aug., 2007.

[10] http://www.intel.com

[11] B.L. Chamberlain, S.J. Deitz, D. Iten and S.E. Choi, "User-defined distributions and layouts in chapel: philosophy and framework," Proc.HotPar10, 2010.

[12] C.D. Krieger, A. Stone and M.M Strout, "Mechanisms that Separate Algorithms from Implementations for Parallel Patterns," Proc. ParaPLoP'10, pp.1-12, 2010.

[13] C.A. Bohn ang G.B. Lamont, "Load balancing for heterogeneous clusters of PCs," Future Generation Computer Systems: The International Journal of Grid Computing and eScience, Vol.18, No.3, pp.389-400, 2002.

[14] B.G. Gu, "TSCcp Load Sharing Algorithm for the Heterogeneous Cluster," Journal of Korean Institute of Information Technology, Vol.4, No.5, pp.29-36, Oct., 2006.

[15] http://en.wikipedia.org/wiki/Mandelbrot_set

[16] http://en.wikipedia.org/wiki/Monte_Carlo_method

구 본 근

e-mail : bggoo@ut.ac.kr

1991년 인제대학교 전자학과(학사)

1993년 부산외국어대학교 컴퓨터공학과 (공학석사)

1998년 경북대학교 컴퓨터공학과(공학박사)

1998년~현 재 한국교통대학교(구. 충주대학교) 컴퓨터공학과 교수

관심분야 : 고성능 컴퓨팅, 병렬 시스템, 고성능 스토리지 등

Patrick Carpenter

e-mail : carpept@ auburn.edu

2010년 Auburn University, Dept. of Computer Science and Physics (Bachelor's degree)

2010년 Present Auburn University, Dept. of Computer Science and Software Engineering(Graduate Student)

관심분야 : Scientific Computing, Computational Physics, theoretical computer science, etc.

Weikuan Yu

e-mail : wkyu@auburn.edu

1995년 Wuhan University, Genetics(B.S.)

1998년 Chinese Academy of Sciences, Molecular & Cellular Biology (M.S.)

2001년 The Ohio State University, Developmental Biology(M.S.)

2004년 The Ohio State University, Computer Science & Engineering(M.S.)

2006년 The Ohio State University, Computer Science & Engineering(Ph.D)

2009년~PRESENT Assistant Professor, Auburn University

2008년~2009년 Research Staff Member, Oak Ridge National Laboratory

2008년~2009년 Adjunct Assistant Professor, University of Tennessee at Knoxville

2006년~2007년 Research Associate, Oak Ridge National Laboratory

2004년~2004년 Summer Intern, Los Alamos National Laboratory

관심분야 : high-end computing, parallel and distributed networking, storage and file system, etc.