INTEGRATING TASK AND DATA PARALLELISM USING PARALLEL I/O TECHNIQUES*

BHAVEN AVALANI†, ALOK CHoudhARY‡, IAN FOSTER§ and RAKESH KRISHNAIYER†
Syracuse University and ANL

Abstract

Task and data parallelism are traditionally seen as disjoint modes of parallelism. But a lot of multi-disciplinary applications can benefit from both types of parallelism. The development of an integrated task/data parallel programming system introduces challenging technical issues in the areas of language design, compilation, and runtime system design. This paper concentrates on communication issues in an integrated task/data parallel system. The paper first presents a semantic model for communication between two data-parallel tasks. Various implementation approaches to realize these semantics are then discussed. We evaluate these different approaches using an image processing example as a case study.

1 Introduction

In data-parallel programming, programs apply a sequence of similar operations to all or most elements of a large data structure; in task-parallel programming, programs consist of a set of (potentially dissimilar) parallel tasks that perform explicit communication and synchronization operations. Both paradigms are supported by a number of parallel languages. High Performance Fortran (HPF) [9] is an example of a data-parallel language and Fortran M (FM) [7], CC++ [4] are examples of task-parallel languages.

An integrated task and data parallel system requires data parallel tasks to communicate and synchronize with each other. This paper presents approaches to achieve this synchronization and communication. This paper also addresses issues related to heterogenous computing in an integrated task/data parallel system. The paper presents a semantic

*This work was supported in part by NSF Young Investigator Award CCR-9357540, a grant from Intel SSD, and IBM.
†CIS, Syracuse University, Syracuse, NY 13244
‡ECE, Syracuse University, Syracuse, NY 13244
§Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439

model for communication between data-parallel tasks. CHANNELS are introduced as modes of communication and synchronization between data parallel tasks. Various implementation approaches to realize the semantics of a CHANNEL are presented. These different approaches are evaluated using an image processing example as a case study.

The organization of this paper is as follows. Section 2 presents a CHANNEL as a mode of communication between communicating data-parallel tasks and outlines the semantics associated with it. Section 3 discusses different approaches to implement a CHANNEL. Section 4 presents a case study along with performance results for each implementation model described in Section 3. Section 5 discusses the conclusions and presents related work.

2 Channel Semantics

We introduce CHANNELS as a mode of communication between data parallel tasks in an integrated task/data parallel system. A CHANNEL provides a uniform mode of communicating data between two data parallel tasks. Programs are constructed by using CHANNELS to plug together concurrent tasks. This provides a many-to-many communication model between different processes of the communicating tasks. Fig 1 shows two tasks T1 and T2 connected by a CHANNEL C. The two tasks are assumed to be data-parallel executing on m and n processors respectively. The semantics of a CHANNEL connection C between two data parallel tasks T1 and T2 are the following:

- Distribution Independence: Tasks on the two ends of a CHANNEL may have different data distributions. A CHANNEL provides a uniform mode of communicating data which is distribution independent. For example, data in task T1 may be distributed in a block fashion and data in task T2 may be distributed in a cyclic fashion. The CHANNEL implementation manages
Figure 1: CHANNEL connection between two tasks

Figure 2: Shared File Model

T1 and T2 are two tasks connected by a CHANNEL C. The CHANNEL is uni-directional with T1 as the sending task and T2 as the receiving task. Data D is communicated over the CHANNEL. The distribution of D on T1 is T1.DISTR and on T2 is T2.DISTR. The distribution of D on the shared file is F.DISTR. The shared file consists of a number of regions R_i. Process P_i of task T1 writes to region R_i of the shared file. This region may be contiguous or striped as illustrated in Fig 3. Shared file 1 in Fig 3 has data distributed in a contiguous format for each process. Shared file 2 in Fig 3 however has data distributed in a striped format. There is also a special region in the file called SYNC.global which is used for synchronization.

The mapping function MW provides a one-to-one mapping between the processes P_i executing task T1 and the regions R_i of the shared file. The mapping function MR maps the regions R_i of the shared file to the processes P_i executing task T2. MR can be a one-to-many mapping as we allow many processes to read from the same file region even though only one process can write to a given region. Similarly MR can also be a many-to-one mapping.

The characteristics of the SFM are the following:

Multiple-Readers/Multiple-writers are allowed in this model for implementing CHANNELS. It is up to the compiler to generate the correct code so that the multiple writers do not overwrite each other. Each reading process has its own file pointer which can read any part of the file. The mapping function MW could map a process either to a contiguous region or a striped region on the file. The information about the mapping is specified at the beginning of the file. The mapping function MR is defined by using this

3 Implementation Approaches

In this section we present two different approaches to implement a CHANNEL described in the previous section. The two models are named Shared File Model and Distributed File Model. These file models apply the parallel I/O techniques described in [10] to achieve the communication and synchronization between data-parallel tasks.

3.1 Shared File Model - SFM

In this model, a CHANNEL is implemented using a shared file as illustrated in Fig 2. This model is explained using the following notations:
information at the top of the file.

Synchronization is achieved through a synchronization variable at the beginning of the file. The reads and writes to the synchronization region of the file are atomic. Ports for a given CHANNEL are defined by the file pointers opened at each end of the CHANNEL. The files are opened either in read-only or write-only mode. Hence the CHANNELS are unidirectional in nature.

The files need not be stored on disks. They are stored in memory buffers and when the buffers overflow, they are transferred to disks. The files are reclaimed once they are read. On reading the data from the file the receiving task resets the synchronization flag and removes the file. We now discuss some of the advantages and disadvantages of this model.

ADVANTAGES:

This approach handles different types of distributions at either end of a CHANNEL. That is, communication over the CHANNEL is distribution independent. Task at the receiving end of a CHANNEL need not be concerned with the distribution of data on the task at the sending end and vice versa. The CHANNEL has to perform the required transformations on the data and present it to the receiving task. This property is important to enable information hiding between the two communicating tasks.

This approach allows dissimilar sets of processors to communicate as long as the file formats are the same. Hence this approach extends easily to a heterogeneous environment. In a heterogeneous environment we can have the tasks running on dissimilar machines having their own file systems. As long as the file system is able to convert the other machine's

file format to its own format the files can be used to communicate data across the two machines.[10]

DISADVANTAGES:

Having a single file tends to form a bottleneck in the system. Typically this file will be striped over multiple disks. But even in this case, one can have a large number of processes accessing data from the same disk.

The synchronization may be implemented independent of the file system. Performing a file read/write to synchronize may be time consuming.

3.2 Multiple File Model - MFM

One of the limitations of the previous model is that it does not allow the data to be communicated in a pipelined fashion. If we have to transfer a large data structure from one task to another, we can pipeline the transfer by breaking the data structure into a number of smaller data sets. Each set has a synchronization variable associated with it. Multiple files can provide this abstraction of communication. This model implements a CHANNEL using a multiple file system. This model is similar to the previous model except that it offers multiple files as an intermediate storage facility. This model is further illustrated in Fig 4. The main distinctions between the SFM and

the MFM are as follows:

- In the MFM, the mapping function MW gives a file name as well as a file region to map the data of a process in a task. A process can typically have more than one file on which its data is mapped.

- Synchronization variables are associated with each file or a set of files in the MFM.
Some of the advantages and disadvantages of the MFM are discussed below.

**ADVANTAGES:**

Since there are multiple synchronization variables, the communication over the CHANNEL can be pipelined. The files in this case are smaller as the data is distributed over multiple files.

**DISADVANTAGES:**

Multiple files require multiple variables for synchronizing accesses to the files. Accessing multiple synchronization variables introduces extra overheads for the implementation. The mapping functions $M_W$ and $M_R$ can become complicated as different mappings for different files may be required.

4 A CASE STUDY

In this section, we present an example program implemented in both the models discussed above. The example program we discuss is a 2D convolution program. We first describe this program in the context of an integrated task and data parallel language.

4.1 2D Convolution: A Case Study

![Figure 5: 2D Convolution Algorithm Structure](image)

Two dimensional convolution is a simple example of a problem that can benefit from both task and data parallelism. As shown in Fig 5, 2-D convolution involves two 2-D FFTs, an element-wise multiplication, and an inverse 2-D FFT.

A data-parallel convolution algorithm can be formulated in HPF as follows. Images are computed one at a time, with data distribution statements used to indicate how data parallelism is to be exploited within each FFT. The 2-D FFT uses a TRANPOSE intrinsic to transpose the input array so that 1-D FFTs can be performed in each dimension without communication. Each FFT operates on all processors. The data parallel version of the code is given in Fig 6. A mixed data/task parallel algorithm can use disjoint subsets of processors for each FFT. Data flows between components on a CHANNEL. We explain the task parallel part of the program using the syntax defined by the task parallel language Fortran M. The integration of FM and HPF is further discussed in [1]. This mixed algorithm can be formulated in HPF/FM by using the FM code in Fig 7 to implement the structure illustrated in Fig 5. The HPPCALL statements create the boxes in the figure and the CHANNEL statements create the connections between the boxes. There are three data-parallel tasks, with each box corresponding to a task. For simplicity, input and output is not considered. The SUBMACHINE annotations control resource allocation. All the processes are fired simultaneuously and the processes communicate with each other on a CHANNEL.

Each data-parallel task is passed a port which is either the reading or the writing end of a CHANNEL. The task fft gets the output end to the CHANNEL through the output outs. Note that we define the number of processors on which the tasks execute explicitly using the submachine concept. We implemented this example using the two implementation models of a CHANNEL described in the previous section. The experimental results along with a discussion of each implementation is presented in the following section.

```c
HPF Code
program data_parallel
  complex A(512,512), B(512,512)
  complex C(512,512)
  !HPFS processors p(32)
  !HPFS align B, C with A
  !HPFS distribute A(BLOCK, *)
  do i = 1, nimages
    call read(A,B)
    call fft(A)  ! FFT first input array
    call fft(B)  ! FFT second input array
    C = A*B
    forall(i=1:512, j=1:512)
      C(i,j) = conjg(C(i,j))
    call fft(C)  ! Inverse FFT
  enddo
```

Figure 6: Data-parallel code for 2D convolution.
4.2 RESULTS

4.2.1 Shared File Model - SFM

The 2D convolution example described in the previous section was implemented on the Intel Delta. We used the Concurrent File System (CFS) provided by the Intel Delta to implement the shared file. The results of our experiments on the Intel Delta are presented in the graph in Fig 8. The graph gives the total time for a complete 2D convolution. We implemented the shared file model for the following processor configurations on the Intel Delta:
- 4 processors for each of the three tasks.
- 2 processors for each of the three tasks.
- 2 processors each for the two ffts and 4 processors for the ifft.
- 4 processors each for the two ffts and 2 processors for the ifft.

Each configuration was executed with data sizes of 256 \times 256, 128 \times 128 and 64 \times 64. A brief discussion of our experiments is presented below:

For a small data size the performance of all the four configurations was almost the same. The file I/O time is predominant when the data size is small as the computation time is relatively less significant. The amount of data written/read from the file is independent of the number of processors.

The computational requirements of the two ffts is lower than the computational needs of the ifft. The ifft includes point matrix multiplication, fft and scaling. Hence for a large data size, the (2,2,4) configuration performs as good as the (4,4,2) configuration although the latter case has more number of processors allocated to it. This shows that the choice of processor configuration is very important for performance.

As the data size increases, the single shared file is striped across a number of disks on the Delta. Small data size problems were not scalable as the file I/O time plays a predominant factor in the performance. However for large data sizes the problem is more scalable. For a 256 \times 256 array, the performance improved by a factor of 1.6 from the (2,2,2) configuration to the (4,4,4) configuration.

4.2.2 Multiple File Model - MFM

For this model, we used one file per sending processor in the sending tasks - the two fft modules. The receiving task ifft had the same number of processors as the sending tasks. Hence there was a one-to-one mapping between the files and the receiving task processors. We performed the experiments on the Intel Delta with the following configurations:
- 4 processors for each of the three tasks.
- 2 processors for each of the three tasks.

The performance results are shown in the graph in Fig 9. Some of the salient features observed in the experiments are given below:

The multiple file performance was in general better than the shared file performance. However the performance improvement was not significant as the data size transferred over the CHANNEL was small.

The performance improved by a factor of 1.75 from the (2,2,2) configuration to the (4,4,4) configuration. The MFM should be more scalable than the SFM as the I/O bottleneck to one single file is avoided in this case. We can observe this scalability improvement from 1.6 to 1.75.
5 Related Work

In a parallel context, most work has focused on task-parallel coordination frameworks for data-parallel components. Foster, Avalani et al. presented the language integration issues involved in mixed paradigm programming[1]. Cheng et al. propose the use of the AVS dataflow visualization system to implement multidisciplinary applications, in which some components may be data parallel programs [6]. Similarly, Quinn et al. describe work on iWARP in which a configuration language is used to connect Dataparallel C computations [8]. The Dataparallel C programs use specialized versions of C I/O libraries for communication. Process and communication structures are static, and all communication passes via a central communication server. Subhlok et al. describe a compile-time approach for exploiting task and data parallelism on the iWarp mesh-connected multicomputer [11].

Chapman et al. [5] have recently proposed the addition of “spawn” and “shared data abstraction” constructs to data-parallel languages to support multidisciplinary programming. The shared data abstraction, a form of monitor, is used to control interactions between tasks created using spawn.

6 Conclusions and Future Work

We present two modes for communication between data parallel tasks. We have defined the semantics of a CHANNEL for this purpose. The important properties of a CHANNEL: are information hiding and distribution independence. An image processing application has been used to evaluate the performance of each implementation approach; experimental studies confirm the semantic correctness of our models and compare the performance among the models. We have listed the advantages and disadvantages of each implementation model and their extendibility to support heterogenous computing.

The development of this prototype is just the first phase in a research program designed to investigate the language constructs, compiler techniques, and runtime mechanisms required to support mixed paradigm programming. The next step in our research would be to tightly integrate the task and data parallel paradigms.

References