Graph Scaling: A Technique for Automating Program Construction and Deployment in ClusterGOP

Fan Chan, Jiannong Cao, and Yudong Sun

Software Management & Development Lab, Department of Computing
The Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong

Abstract. Program development and resource management are critical issues in large-scaled parallel applications and they raise difficulties for the programmers. Automation tools can benefit the programmer by reducing the time and work required for programming, deploying, and managing parallel applications. In our previous work, we have developed a visual tool, VisualGOP, to help visual construction and automatic mapping of parallel programs to execute on the ClusterGOP platform, which provides a graph-oriented model and the environment for running the parallel applications on clusters. In VisualGOP, the programmer needs to manually build the task interaction graph. This may lead to scalability problem for large applications. In this paper, we propose a graph scaling approach that helps the programmer to develop and deploy a large-scale parallel application minimizing the effort of graph construction, task binding and program deployment. The graph scaling algorithms expand or reduce a task graph to match the specified scale of the program and the hardware architecture, e.g., the problem size, the number of processors and interconnection topology, so as to produce an automatic mapping. An example is used to illustrate the proposed approach and how programmer benefits in the automation tools.

1 Introduction

Programming with parallel applications is a difficult task involving programming details, processors information and network configurations. A method to simplify parallel program development is to abstract and represent the parallel programming structure by logical graphs.

Task graph is a graphical representation of a parallel program. It describes the logical structure of the program in which the nodes represent the computational tasks and the edges denote the communication links and precedence relationships among the nodes. Varieties of task graph have been proposed. Directed acyclic graph (DAG) [1, 2] and task interaction graph (TIG) [3, 4] are two ordinary types of task graphs. TIG is a concise representation of parallel program. The edges can represent any relationships between the nodes, for example, communication, synchronization, and execution precedence. The edges

X. Zhou et al. (Eds.): APPT 2003, LNCS 2834, pp. 254–264, 2003.

can form loop to represent the iterative operations. Thus, TIG is more flexible to describe different program structures. Designing programs in TIG can simplify many programming details. However, there is a restriction for programming. TIG may have difficulty in handling complex relationships between tasks in a large-scaled graph. For the implementation, network configurations such as nodes-to-processors and LPs(local programs)-to-nodes mapping are time consuming tasks and difficult to handle manually. The programmer needs some tools for designing the TIG efficiently and managing the task mapping automatically.

In our previous work, we have developed tools for supporting the development of parallel applications, based on the graph-orient programming (GOP) model [5, 6]. We have developed a visual programming tool, VisualGOP [7], for designing the GOP program graphically. VisualGOP has a highly visual and interactive user interface, and provides a framework in which the design and coding of GOP programs, and the associated information can be viewed and modified. It also facilitates the compilation, mapping, and execution of the programs. Programs constructed in VisualGOP are deployed to the ClusterGOP, a high-level parallel computing platform for the cluster [8]. However, in VisualGOP, the programmer needs to manually build the task interaction graph. This may lead to scalability problem for large applications. In this paper, we describe the improvement to VisualGOP with an automation tool for constructing the graph and deploying the application in an efficiently way. In the tool, the graph scaling algorithms adapts a basic graph to the parameters of the application and determines the mapping between the programs and the processors automatically.

Section 2 introduces the ClusterGOP framework. Section 3 discusses the graph scaling approach. Section 4 presents the implementation of the task scaling and mapping tool in VisualGOP and the experiment using an example. Section 5 concludes the paper with the discussion of our future work.

2 The ClusterGOP Framework for Programming on Clusters

2.1 The ClusterGOP Model and Architecture

ClusterGOP is based on the GOP model, in which parallel/distributed program is defined as a collection of *local programs* (LPs) that may execute on several processors. Parallelism is expressed through explicit creation of LPs and communication between LPs is solely via message passing. The distinct feature of GOP is that it allows programmers to write distributed programs based on user-specified graphs, which serve the purpose of naming, grouping and configuring LPs. The graph construct is also used as the underlying structure for implementing uniform message passing and LP co-ordination mechanisms.

The key elements of GOP are a logical graph construct to be associated with the LPs of a parallel/distributed program and their relationships, and a collection of functions defined in terms of the graph and invoked by messages traversing the graph. As shown in Figure 1, the GOP model consists of the following:

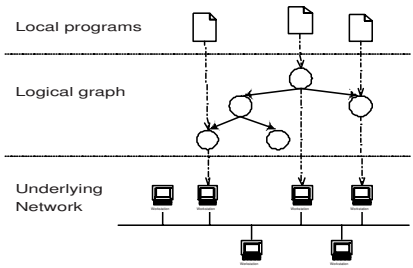

Fig. 1. The GOP conceptual model

- A *logical graph* (directed or undirected) whose nodes are associated with local programs (LPs), and whose edges define the relationships between the LPs.
- An *LPs-to-nodes mapping*, which allows the programmer to bind LPs to specific nodes.
- An optional *nodes-to-processors mapping*, which allows the programmer to explicitly specify the mapping of the logical graph to the underlying network of processors. When the mapping specification is omitted, a default mapping will be performed.
- A library of language-level graph-oriented programming primitives.

The GOP model provides high-level abstractions for programming distributed programs, easing the expression of parallelism, configuration, communication and coordination by directly supporting logical graph operations. It is important to note that GOP is independent of any particular language and platform. It can be implemented as library routines incorporated in familiar sequential languages and integrated with programming platforms such as PVM and MPI [9, 10].

ClusterGOP is an implementation of the GOP framework on MPI. The ClusterGOP software environment is illustrated in Figure 2. The top layer is a visual programming environment, VisualGOP, which supports the design and construction of parallel/distributed programs. A set of GOP API is provided for the programmer to use in parallel programming, so that the programmer can build application based on the GOP model, ignoring the details of low-level operations and concentrating on the logic of the parallel program. The GOP library provides a collection of routines implementing the GOP API. The goal in the GOP library implementation is to introduce a minimum number of services with a very simple functionality to minimize the package overhead.

The runtime system is responsible of compiling the application, maintaining structure, and executing the application. In the target machine, there exists two runtimes. The first one is the GOP runtime, a background process that provides graph deployment, update, query and synchronization. When deploying and updating the graph, it will block other machines to further update the graph and synchronize the graph update on all machines. Another runtime is the MPI

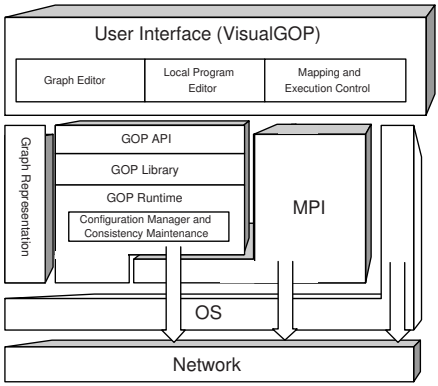

Fig. 2. The ClusterGOP Framework

runtime, which provides a complete set of parallel programming library for the GOP implementation.

3 Graph Scaling and Mapping

In this section, we will first introduce the basic patterns of the task graph, and then discuss the graph scaling method and the graph mapping strategy.

3.1 Regular Graphs for Parallel Application

The programmer needs to create a realistic graphical representation for parallel applications. The task graph should be defined as an abstraction of program structure. In addition, it should be high scalability to adapt to the parameters such as the problem size and number of processors. Our graph scaling approach is based on TIG by which the graph scaling algorithms and the graph mapping strategy will be implemented. In graph scaling, the nodes in the graph can be decomposed or merged, and the edges are reconstructed based on the original graph structure to produce a new task graph to match the parameters. TIG provides a concise topology to describe process-level computation and communication. It has a flexible structure for graph scaling.

Task graphs may have an arbitrary structure. More often, however, a task graph can take a regular topology such a tree, mesh, hypercube, etc., as parallel algorithms are often developed based on a regular topological model [11]. The following are typical topologies of task graph:

Tree. A *tree* has one root node and multiple leaf nodes (leaves). Each node except the root node has one parent. The edges are acyclic. If a graph satisfies these conditions, it can be identified as a tree.

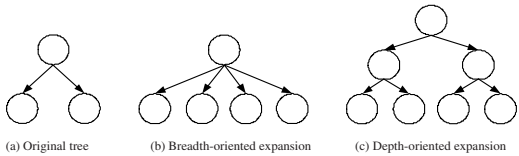

Fig. 3. Graph expansion for tree structure

Mesh. In an $m \times n$ *mesh*, there are four corner nodes with two neighbors each, $2(m\text{-}2)+2(n\text{-}2)$ boundary nodes with three neighbors each, and $(m\text{-}2)(n\text{-}2)$ inner nodes with four neighbors each.
Hypercube. In a *hypercube* with 2^n nodes, each node has n neighbors.
Arbitrary topologies. A task graph can present any other arbitrary topology.

The topology of a task graph is identified by the programmer. If the size of the graph, measured in the number of tasks that can be executed in parallel, does not conform to the parameters specified at runtime such as the problem size and the number of processors, graph scaling is required to derive a new graph from the original one to match the parameters.

3.2 Graph Expansion and Compression

Graph scaling can be made in two modes. If the number of parallel tasks is less than the required problem size or the number of processors, graph expansion will be performed to generate more tasks. On the other hand, if the number of parallel tasks is greater than the problem size, the graph should be compressed to include fewer nodes, although this is a rare situation in task graph. In addition, if the number of parallel tasks is greater than the available processors, the graph may be compressed.

In *graph expansion*, some nodes are decomposed and the edges are re-linked between the nodes based on the graph topology. The tasks should be redistributed among the expanded nodes. Figure 3 shows the expansion of a tree structure. The tree in Figure 3(a) can be expanded in two directions. One is breadth-oriented expansion as shown in Figure 3(b), in which all expanded nodes are attached to the root. The other is depth-oriented expansion in which the nodes spawn children beneath as shown in Figure 3(c).

Figure 4 shows the graph expansion for a mesh. The original 2×2 mesh is expanded to 2×4 and then 4×4 meshes by redeploying the decomposed nodes. If there are n nodes in a mesh, they are deployed as an $\sqrt{n} \times \frac{n}{\sqrt{n}}$array. The nodes are linked by edges according to the mesh topology.

A hypercube is expanded in a similar way. The decomposed nodes are linked based on the hypercube topology. That is, each node is linked to k neighbors if there are totally 2^k nodes. Figure 5 shows the expansion of a 4-node hypercube to 8-node and 16-node hypercube.

Graph compression can use the same approach as the clustering in task scheduling [1, 2, 12]. It merges the nodes of a graph to clusters when the number

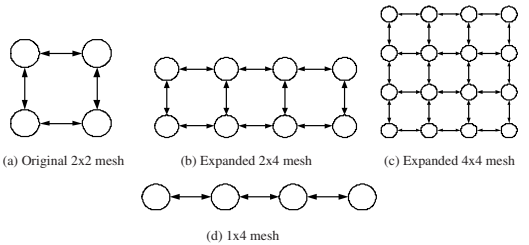

Fig. 4. Graph expansion for mesh structure

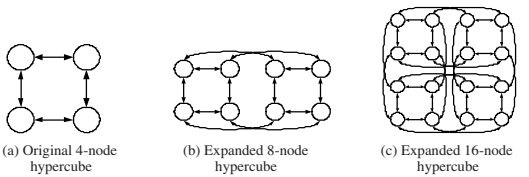

Fig. 5. Graph expansion for hypercube structure

of nodes is greater than the available processors. Graph compression is particularly useful to the cyclic graphs such as mesh and hypercube. The criterion of graph compression is the reduction of communication between the nodes. It analyzes the edges between the nodes and determines the neighboring nodes to be merged; meanwhile the topology of a compressed graph can be maintained. To compress the 16-node mesh in Figure 4(c), for example, the analysis can determine that the merge of two adjacent rows or columns has the same effect in reducing the inter-node communication. Thus, the compression can be made on either direction. If we choose row-oriented compression, the graph can be compressed into an 8-node in Figure 4(b). To compress the 8-node mesh to a 4-node one, the row-oriented compression will produce a one-dimensional mesh as shown in Figure 4(d). The column-oriented compression will result in a 2×2 mesh as shown in Figure 4(a). Using the criterion of communication reduction, it can be decided that the row-oriented compression can result in a graph with minimal the communication (denoted by three edges), in contrast to the result of column-oriented compression (i.e., the 2×2 mesh with four edges).

3.3 Graph Mapping

After the graph analysis and the graph scaling, LPs will be mapped to the task graph which has expanded or compressed. In SPMD (Single Program Multiple Data) model, all the nodes share the same copy of the program, so the mapping is simple. In the MPMD (Multiple Program Multiple Data) model, each node may work on different tasks. The programmer can choose a set of rules to do the LPs-to-nodes mapping automatically. There are rules for classifying the LP into different groups, e.g., a range of node ID, similar node names, and node

types. Finally, a task graph will be mapped to processors. Each processor is responsible for executing a node of in the task graph; i.e., there is a one-to-one correspondence between a processor and a node.

4 Implementation

We have developed algorithms for graph scaling as described in the previous section and implemented in VisualGOP. In this section, we first describe the scaling algorithms for graph expansion and then use our example to show how the proposed algorithms help the program design.

4.1 Scaling Algorithms

In the following parts, we will introduce three basic scaling algorithms for graph expansion. We assume that Gn is a 2-D graph and not weighted. We also assume that |Nn| (the number of nodes) = |Np| (the number of processors), and there is a one-to-one mapping between Nn and Np.

Tree. This structure can have two expansion types, the breadth-oriented and the depth-oriented expansion. We choose the binary tree as the example, which belongs to the depth-oriented expansion. The graph of binary tree has 2^n-1 nodes, where n starts from 2. During the graph expansion, the function ExpDepBTree() updates the graph according to the input node number. In each loop of the graph expansion, Search_Leaf_Nodes() will be invoked for adding the leaf nodes into a list. Each tree leaf node in the list will be expanded to produce a new level of leaf nodes. The psedueo-code segment for this algorithm is shown below:

```
public void ExpDepBTree( graph Gi, int nodenum ) {
  Until Gi's nodenum >= input nodenum
    initialize the graph and nodelist for saving the result
    Search_Leaf_Nodes(Gi, root, nodelist_t)
    For each node k in nodelist_t
        add new_left_node, new_right_node to Gi
        add new_left_node, new_right_node to Gi.alist[k]
    End Loop
    update Gi
}
```

Mesh. We use a 2x2 mesh as the original graph. This graph has 2^n nodes, where n starts from 2. In order to get a better communication performance, the expanded mesh should be maintained in a regular shape, so that the communication paths among the nodes are short in average. By identifying the node number, the graph expands vertically if n is an odd number; otherwise, the graph expands horizontally. Graph expansion is done by joining two identical graphs to form a new one. After that, all nodes are re-ranked

for getting new node IDs. The psedueo-code segment for this algorithm is shown below:

```
public void ExpMesh( graph Gi, int nodenum ) {
  initialize the graph for saving the result
  Until Gi's nodenum >= input nodenum
    Gj := Graph_Copy(Gi)
    Find_Coloumn_Row_Num(Gi's nodenum, col_num, row_num);
    calculate n_multiple for graph expands direction
    If n_multiple is odd Then
      For n=0 to col_num-1
        u:= node nodeID(col_num*(row_num-1)+n) Gi's last row
        v:= node nodeID(n) Gj's first row
        add v to Gi.alist[u]
        add u to Gj.alist[v]
      End Loop
    Else
      For n=0 to row_num-1
        u:= node nodeID(col_num*(n+1)-1) Gi's last column
        v:= node nodeID(col_num*n) Gj's first column
        add v to Gi.alist[u]
        add u to Gj.alist[v]
      End Loop
    End If
    rerank Gi and Gj
    join Gi and Gj
}
```

Hypercube. We choose a 2x2 hypercube as the original graph. This graph has 2^n nodes, where n starts from 2. Hypercube expands by connecting corresponding nodes from two identical graphs. All nodes are re-ranked after the creation of the new graph. The psedueo-code segment for this algorithm is shown below:

```
public void ExpHypercube( graph Gi, int nodenum ) {
  initialize the graph for saving the result
  Until Gi's nodenum >= input nodenum
    Gj := Graph_Copy(Gi)
    For n=0 to Gi's nodenum-1
      add v to Gi.alist[u]
      add u to Gj.alist[v]
    End Loop
    rerank Gi and Gj
    join Gi and Gj
}
```

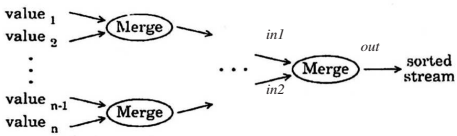

Fig. 6. The Merge Sorting Algorithm

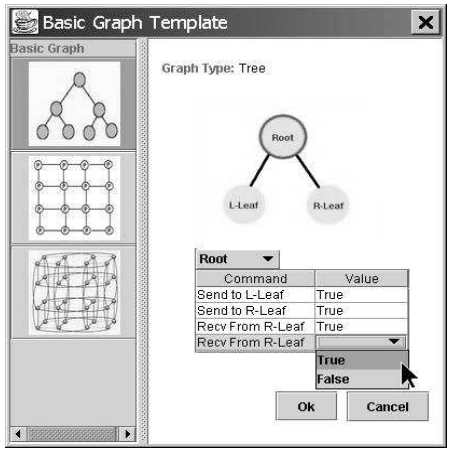

Fig. 7. The Graph Template for the Basic Graph Diagram

4.2 An Example: The Parallel Merge Sorting

In our example, we use VisualGOP for the programming demonstration, which involves the program design, the graph scaling and the graph mapping. The programmer first uses VisualGOP to design the logical graph and LPs, then a processor list is created and the graph will be expanded or decompressed. Finally, VisualGOP applies the LPs-to-nodes and the nodes-to-processors mappings automatically.

We use a typical parallel algorithm, the parallel merge sorting, to show how programmer benefits by using the VisualGOP as an automation tool for designing parallel applications. The idea behind a merge sorting is to merge two sorted lists into a longer sorted list repeatedly in parallel (see in Figure 6). The graph is constructed out of instances of merge program. Each merge process receives values from two ordered input streams, *in1* and *in2*. It merges these values to produce one ordered output stream, *out*.

Step 1: Define the Basic Graph

The first step for the programmer is to define the graph for the parallel application (see in Figure 7). VisualGOP will ask for the graph pattern, which is required for the basic graph of the application. In this example, we choose the binary tree template for the merge sorting application.

Step 2: Program LPs and Define the LPs-to-nodes Mapping

In the next step, the programmer needs to define programs for LPs-to-nodes mapping. The example uses MPMD programming model, which contains three source codes (root.c, transfer.c and leaf.c) for different tasks in the merge sorting application. The first LP (root.c) is mapped to the root node, for collecting the final result. The second LP (transfer.c) is mapped to the transfer nodes (non-root and non-leaf nodes). Their major tasks are accepting the input from child nodes, merge and sort the data, and then transfer the results to their parent nodes. The last LP (leaf.c) is mapped to the leaf nodes, they are the nodes for retrieving the input list and pass the data to their parent nodes (transfer nodes or root nodes). After defining the LPs, VisualGOP provides mapping rules for the LPs-to-nodes mapping automatically.

Step 3: Prepare the Processors

The next step is the creation of the processor list for nodes-to-processors mapping. Programmer edits the processor list through adding, removing, or modifying the existing processor information. VisualGOP restricts the processor number input according to the graph pattern to prevent the programmer input unaccepted processor number to the algorithm.

Step 4: Graph Expansion

After receiving the processor number, VisualGOP will choose graph expansion if the available processor number is larger than the graph node number, otherwise the graph compression is used. The programmer can also make a preview on the expanded or compressed graph, for accepting or rejecting the changes.

Step 5: Graph Mapping

Finally, the expanded graph is created and the graph mapping starts automatically. Different LPs will be mapped to the specified nodes. The nodes-to-processors mapping helps programmer mapping the nodes to the processors automatically. Programmer can make the final changes to the mapping if needed. For example, the programmer can change the nodes-to-processors mapping to use more powerful processor to increase the performance of a specified node. After the graph mapping, the graph scheduling for the merge sort application has finished. Programmer can compile and execute the application directly.

5 Conclusions and Future Work

We have introduced the ClusterGOP system, which provides high-level abstractions for programming parallel applications, easing the expression of parallelism, configuration, communication and coordination by directly supporting logical graph operations. We also provide a visual programming environment, VisualGOP, to provide a visual and interactive way for the programmer to develop and deploy parallel applications. We propose the scaling algorithms for the task graph, which supports graph expansion and compression to match the specified parameters. The graph scaling realizes the draw-once run-variedly feature of task graph that contributes to the practicability of task graph based scheduling in parallel computing.

In our future work, we will take into account the computation load on the nodes and the communication costs on the edges when map the nodes to processors. One target of the mapping is to reduce inter-processor communication, and another one is to balance the workload among processors so as to minimize the execution time of entire program.

Acknowledgement. This work is partially supported by the Hong Kong Polytechnic University under the research grant H-ZJ80.

References

1. Kwok, Y.K., Ahmad, I.: Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Computing Surveys **31** (1999) 406–471
2. El-Rewini, H., Lewis, T.G., Ali, H.H.: Task Scheduling in Parallel and Distributed Systems. Prentice-Hall (1994)
3. Hui, C., Chanson, S.: Allocating task interaction graphs to processors in heterogeneous networks. IEEE Transactions on Parallel and Distributed Systems **8** (1997) 908–925
4. Senar, M.A., Ripoll, A., Cortes, A., Luque, E.: Clustering and reassignment-based mapping strategy for message-passing. In: 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing, Florida, United States (1998) 415–421
5. Cao, J., Fernando, L., Zhang, K.: Programming distributed systems based on graphs. Intensional Programming I, World Scientific (1994)
6. Cao, J., Fernando, L., Zhang, K.: Dig: A graph-based construct for programming distributed systems. In: Proceedings of 2nd Int'l Conference on High Performance Computing, New Delhi, India (1995)
7. Chan, F., Cao, J., Chan, A.T., Zhang, K.: Visual programming support for graph-oriented parallel/distributed processing. Submitted for publication (2002)
8. Chan, F., Cao, J., Sun, Y.: High-level abstractions for message-passing parallel programming. To appear in Parallel Computing (Elsevier Science) (2003)
9. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V.S.: PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing. MIT Press, Cambridge, MA, USA (1994)
10. Snir, M., et al: MPI: the complete reference. MIT Press, Cambridge, MA, USA (1996)
11. Kumar, V., Grama, A., Gupta, A., Karypis, G.: Introduction to Parallel Computing: Design and Analysis of Algorithms, 2nd Ed. Addison Wesley (2002)
12. Darbha, S., Agrawal, D.P.: A fast and scalable scheduling algorithm for distributed memory systems. In: Proceedings of the 7th IEEE Symposium on Parallel and Distributed Processing, San Antonio, TX (1995) 60–63