MPI/C Advanced Pattern-based Parallel Programming System

Paras Mehta
Department of Computing Science
University of Alberta, Edmonton, Alberta, Canada

1 Description

MPI/C Advanced Pattern-based Parallel Programming System (MAP_3S) [4] is a system based on previous work on Correct Object-Oriented Pattern-based Parallel Programming System (CO_2P_3S) [3], but built upon an MPI/C codebase.

A pattern-based parallel programming system in the style of CO_2P_3S uses patterns of solutions to common parallel programming situations as a basis for generating program frameworks. A pattern user will then build on such a generated framework, completing it to suit a particular problem. A pattern, then, in the CO_2P_3S sense, is a collection of frameworks with similar overall structure, but differences in code.

A user chooses the appropriate framework for his problem by selecting a pattern and setting a number of pattern parameters, and the system will generate that framework automatically. Finally, the user adds his own application-specific code to the framework generated, and the program is complete.

At present, the code base completed toward this goal for MAP_3S contains prototypes for the frameworks necessary for each pattern. These prototypes are programs that have been designed with abstraction and encapsulation in mind to accomodate code changes required to generate the different frameworks required when parameters are changed. The prototypes also contain indications of the sites of parameter-dependent code changes; an indication of where frameworks for a particular pattern differ.

This document, then, contains a description of each prototype in the context of the pattern that it is being used for. Moreover, there are details of the implementation of each of these prototypes necessary to understand how they work and, potentially, how further work and modification can be done.

2 Common Design Decisions

Throughout the course of this work, design decisions were made for a variety of reasons, including efficiency and generality. These decisions are often made in one pattern and repeated in other patterns for consistency and simplicity. Some of these decisions and their rationales are outlined below.

2.1 Macros

In CO_2P_3S, the user's application specific code is inserted into the program via a series of `hook methods`, which are instance methods in user-written classes. Because MAP_3S uses C rather than Java, the nature of these `hook methods` cannot be completely replicated. Instead, a C equivalent must be used.

One can use functions to replicate this functionality. During early work, however, it was found that compilers do not always inline these functions when it is desirable for performance, causing degradation in performance, most notably in the parallelization of the program. To solve this problem, we implemented the framework hooks as C macros.

Unfortunately, C macros have their own problems, including debugging difficulties and the higher possibility of syntactic errors. Future work would look at finding a better solution to this problem.

2.2 C Structures

In order to accomodate users' requirements for a wide range of possible data for application specific code, CO_2P_3S allows users to define their own classes in Java and defines hook methods within those classes. The user-defined classes then become an integral part of the object-oriented framework.

In MAP_3S, a pattern requires the user to define C structures rather than Java objects to hold the necessary data for the application. The key factor for this decision is generality of the design; structures can be changed with minimal impact to the program, apart from the application specific sections of the code. Moreover, C structures represent a logical means to organize and encapsulate data.

2.3 MPI Packs

There is one part of the framework where the redefinition of an application-specific C structure can have an immediate impact: message passing. Because MPI requires the size of a message to be known when it is sent and received, the program must have some knowledge of what it is sending. The best way to provide this information without loss of generality is to use MPI packs as the datatype of a passed message.

MPI packing and unpacking functions allow structure data to be combined into a single piece of data and subsequently passed in messages between processes. At the point of packing and unpacking, however, specific function calls must be redefined whenever a structure changes. Macros that encapsulate the packing and unpacking functions for a key structure in a pattern are used to ensure that these calls are made properly. These macros are currently defined manually, but an automated code generator would likely be able to generate the macros for the user.

3 Mesh Pattern

3.1 Description

Mesh problems are those which involve multi-dimensional arrangements of regular elements whose values are iteratively recomputed. Moreover, the recomputation of these values depends not only on the previous value of the element, but also that of its neighbours.

3.2 Details

The mesh pattern is implemented by mapping a structure containing a subset of elements, a `ProcessBlock`, on top of the complete collection of elements that are to be iterated over. A process may have a number of these blocks, depending on how they are distributed. Each element in the mesh is represented by a C structure (`Elem`) which is defined by the user, but must have a guaranteed maximum compile-time size in order to be passed in MPI messages during the computation.

In addition to the subset of elements, each `ProcessBlock` contains information necessary to communicate with the rest of the blocks in the mesh. This information includes the process the block belongs to, an index of the block with respect to all of its process' blocks, and its coordinates in the two-dimensional mapping of blocks that overlays the entire mesh. Additionally, the block retains information necessary to communicate with each of its neighbours; information is stored in an `nb_info` structure. This includes whether communication is necessary in a particular direction, the number of elements to communicate, which process the neighbour belongs to and more. Finally, the block has a void* pointer, which should be used by the user to reference application-specific data above and beyond what is in each `Elem` that may be necessary for the computation.

The framework hooks for this pattern are fairly straightforward, and borrowed from the CO_2P_3S pattern of the same name. They include iteration functions (*e.g.,* `interior_node`), as well as functions for the preparation and postprocessing of elements, indications of computation completion, and data output functions. The hooks are contained in `framework_hooks.h`.

The processing of a mesh begins with the execution of the `runMesh` function. Each process determines how many and which `ProcessBlock`s of the mesh it will need to run, and then initializes them. If necessary, initial data is distributed from the root process to the others. Next, while the process' blocks have not completed their computations, the process iterates through fringe synchronization and block computation. Finally, when the iteration is complete, final data is collected from the processes.

3.3 Example Problem

The example problem implemented in the code provided is Conway's Game of Life; a simple simulation of a number of cells. At each generation, a cell is alive if it has three living neighbours, or if it has two living neighbours and was previously alive. Otherwise, the cell is dead. The game runs for an arbitrary number of generations, and makes for a fairly simple mesh computation.

4 Distributed Queue Pattern

4.1 Description

A distributed queue system is a pulling model of task parallelism that allows processes to dynamically balance loads in a computation. Given that a problem can be broken down into independently computed tasks (*i.e.,* each task can be computed as an autonomous unit, and any tasks it is dependent on have already been completed before it has been created.) , in a task parallel system, each process independently consumes these tasks. In a distributed queue system, each process maintains its own work queue which holds these tasks, and steals work from other processes when there are no tasks in its own queue. For systems that start with few tasks but quickly generate a large number of tasks, a distributed queue is a good way to balance workloads.

4.2 Details

4.2.1 Tasks

The code for this pattern deals with task management by making use of two user-created structures: a `Task` and a `Result`. In general, a `Task` can be represented as a C structure containing a function pointer with functional arguments. That is, a `Task` is something to do, and the arguments with which to do it. The `Result` is a structure that contains the pertinent results of processing a `Task`. This structure is returned to the process that generated the initial `Task` in order to make use of the computed information.

For processing, the `Tasks` are stored in a `WorkList`. Each process maintains its own `WorkList` and processes work from that list. If there is no work, then the process will attempt to take work from other processes.

Care must be taken by the user in designing a `Task` and `Result` for two reasons. First, the size of the structures must have a guaranteed maximum at compile time, because they are passed in MPI messages. Second, using pointers to reference data from structures is not possible, because pointers reference only local memory space. Once a pointer is sent to another process, its reference is irrelevant; it no longer applies. That said, a pointer could be passed in a `Task` if only to be subsequently returned in the result, so that updating results can be handled properly.

4.2.2 Threads

To allow processes to perform task computation while simultaneously being able to respond to requests, two pthreads are run on each process: one computation thread and one communication thread.

The computation thread checks the `Worklist` for work and consumes tasks if they are available. If there are no tasks to process, a message is sent requesting work from other threads. The process to request from is chosen at random, but this algorithm can be changed to better suit the topography of the processes. Finally, once a task is processed, the computation thread will return the result to the parent of the processed task for an update. If the parent is not local to the process (*i.e.,* not in the memory space of the updating process), the result will be sent to the process that generated the task.

The communication thread instead blocks on a message receive and, upon receipt of any messages, deals with those incoming messages:

- **Request for work**: if the process has extra work, it will send work to the requesting process
- **Reply to request**: if another process has sent work, the work is added to the `WorkList` and the computation thread will be woken up, otherwise another request will be sent
- **Returning result**: the parent task will be updated
- **Termination signal**: if all work is complete, the termination token will be passed on, and the process will wait on a broadcast of the termination signal.

4.2.3 Termination

Termination is handled with a token-ring algorithm. When the root process has no work remaining, it will send a termination token to a process. The process will determine if it is done, in which case, it will pass a true token to the next process. If there are tasks remaining, it will pass a false token. In this manner, the token is passed to each process in the program execution environment. Once it returns to the root process, the token's value is broadcast to each process. If the token returns as true, the computation is complete, and all processes can terminate.

4.2.4 Framework Hooks

There are two major hooks for the user to fill in this pattern: `processTask` and `updateResult`. `processTask`, as the name suggests, involves performing the task that has just been pulled off of the worklist. `updateResult` allows the user to do what he wishes with the results of a particular task calculation. It will be called on the process that generated the task whose result is being updated.

For each of these hooks, it may be necessary for the user to add tasks back onto the `WorkList` as they are generated. As such, the user must understand how to create and add tasks to the `WorkList`, using functions such as `wl_push_tail` to push work to the tail of the `WorkList`.

4.2.5 Remaining Issues

It appears that pthread creation can only be done once in an execution. Allowing multiple distributed queue systems to be created in a single runtime would be advantageous and should be looked at.

Additionally, framework hooks still require the user to understand the `WorkList` structure of the framework in order to spawn tasks from the processing or updating of previous tasks. Removing this link between application-specific and pattern code would be ideal.

4.3 Example Problem

There is no real example problem implemented in the code right now, but rather a simple algorithm which will produce two tasks for every one processed, until 40 tasks have been created.

5 Search Tree Pattern

5.1 Description

This pattern deals with problems that involve tree traversal, specifically traversal in a divide-and-conquer manner. The framework is built starting from the distributed queue framework, adapted for a divide-and-conquer traversal of a tree.

The execution of search tree pattern program consists of two phases: divide and conquer. The divide stage involves generating more nodes that can be processed in parallel. The conquer stage involves the sequential processing of a node.

This dichotomy implies a division of the search tree into two parts. The initial expansion of the tree - that occurring at higher depths - comprises the divide phase. The expansion of these nodes occurs in parallel, and each node can potentially be expanded by any process. At deeper levels of the tree, expansion is performed sequentially, with the process that expands the root of a given subtree expanding all the nodes in that subtree. The partition of this tree is called the parallel threshold.

5.2 Details

5.2.1 Adaptation of Distributed Queue

There are a number of necessary members of the structure defining a task that are used in the adaptation of the distributed queue.

- **id**: Unique identifier for each node in the tree
- **depth**: Depth of the node in the tree
- **generator**: Process id of the generating process
- **parent_pointer_g**: Pointer to the parent of the node, in the generator's address space
- **children**: Number of children for this task

A `Task` in the distributed queue represents a node in the tree; but only so far as a parallel expansion of the tree. When searches are performed sequentially below that parallel threshold, no new tasks are created.

The `processTask` hook of the distributed queue pattern will determine whether the parallel threshold has been reached by calling the `parallel` hook. If it has not, more division is required and, as such the `divide` hook of the search tree pattern is called. The `divide` hook will then presumably generate more nodes to be processed in parallel. If the parallel threshold has been reached, the `conquer` hook is run. This should sequentially process the node, usually by sequentially searching a deeper subtree, before returning a result that is passed back up the tree via the `updateTask` hook of the distributed queue.

5.2.2 Modifications to the Distributed Queue

Some changes needed to be made to actual pattern code to better suit it to problems fitting this divide-and-conquer model.

First, a mechanism to allow a search to be restarted was added. This is useful in iterative deepening searches, where searches are repeated at increasing depths until a solution is found. At the end of a search, a framework hook (`continuing`) is called which can check if the search is complete and, if more search is required, create a new root node for a new tree to be searched.

Second, the framework code keeps track of how many children each node generates, and will use this information to determine when a node is updated back to the parent node. That is, once all of a node's children have been processed and have updated back, the node will update back to its own parent, thus saving repeated communications that might otherwise occur.

5.2.3 Alpha-Beta Search

Although alpha-beta searches have been implemented using this pattern, they are not strict divide-and-conquer tree searches and, as such, require some modification to the code generated by the pattern. In the future, development of a separate pattern for Alpha-Beta search, possibly taking advantage of some of the unique parallelizable features of the search, may be necessary.

5.3 Example Problem

There are two example problems that have been implemented with this pattern. The first is the Game of Kece, a zero-sum two player game involving the placement of crossing words on a two dimensional board. This game requires an Alpha-Beta search. The second is finding the solution to a 15-puzzle, which requires an iterative deepening A* search; a better fit to the divide-and-conquer pattern.

6 Wavefront Pattern

6.1 Description

Wavefront problems are generally calculations involving two-dimensional data structures with dependencies between the elements that can be mapped using directed acyclic graphs. Problems with these kinds of data dependency relationships cannot be solved by using a mesh pattern implementation. However, the data structure as well as the requirement of communicating fringe rows and columns between processes is still present in this pattern of problems. Therefore, the basic structure of a mesh pattern can then be preserved. The wavefront pattern, then, adapts the mesh pattern by providing a mechanism to express the dependencies between array elements, and a communication pattern that reflects these dependencies at a `ProcessBlock` level.

6.2 Details

6.2.1 Modification of Mesh Pattern

The elements in a wavefront calculation must be processed in an order that respects the data dependencies between elements. Consequently, the `ProcessBlocks` that divide up the array must also be allocated, distributed, and processed in a particular order. Therefore, the choice of a particular order for processing `ProcessBlock`s must be made; a choice that will impact both the distribution of these blocks as well as communication between them. For simplicity, we modify the Mesh Pattern by adding an array that stores this distribution of blocks, `pb_distribution`.

0	2	5	9
1	4	8	12
3	7	11	14
6	10	13	15

Figure 1: Sample distribution of blocks in a wavefront

The communication framework of the mesh is also altered for this pattern. We are no longer communicating a block's complete fringe to neighbouring blocks, but rather sending only the fringe rows and columns that are in directions that correspond to element dependencies. Once an entire block has been processed, relevant neighbours are determined, and the fringes in the necessary direction are sent to that neighbour.

Finally, before a block can be processed, we confirm that all of its dependencies have been filled. Thus, each block keeps track of its dependencies when fringe elements are received, in an array that has been addeded to the `ProcessBlock` structure, `filled_deps`.

6.2.2 Framework Hooks

The hooks from the mesh pattern are, for the most part, retained from the mesh pattern. Pre- and postprocessing hooks, as well as the completion and data output hooks remain the same. There are some differences with the iteration hooks, however. Because there are dependency relationships between elements, not all the element operation special cases are required. For example, in a wavefront with 'north' and 'west' dependencies, only the top edge, left edge and top left corner are special cases. The other edges of the block are general 'interior' cases; therefore there are only four 'types' of elements, and hence only four hooks are required for the computation.

Also, because there are only two dependencies, only two neighbouring elements are required for the computation for each element. Hence, the function headers for the framework hooks are different than those in the mesh pattern, with far fewer `Elems` required to be passed to the function. Function headers are determined by the data dependencies of the wavefront computation.

6.2.3 Distribution Array

The array describing block distribution, `pb_distribution` is of particular note, because defining and referencing it correctly is vital to correct operation of the pattern.

The dimensions of this array match the layout of the `ProcessBlocks` of the complete mesh. Each element in this array represents one of these blocks, and the value of each element indicates its place in the

1	0	0	0	0	0
0	3	1	0	3	0
0	1	1	0	2	0
0	0	0	2	1	0
0	0	2	3	2	2
0	0	0	1	2	3

Figure 2: An example of a skyline matrix

order of computation.

This array is created before the computation begins and is used to guide the distribution of blocks in such a way as to maximize parallelism. One such distribution is indicated in Figure 1. Here is a situation where there are 16 `ProcessBlocks`, arranged in a 4x4 mesh. (Recall that each of these blocks can hold a number of mesh elements.) If we process each block in the order indicated in the figure, note that blocks 6, 7, 8, and 9 can be processed simultaneously, because all of their dependencies have been filled once block 5 is processed.

Thus, ideally, each of blocks 6, 7, 8, and 9 would be distributed to different processes, so that the element values therein could be calculated in parallel. The `pb_distribution` array is maintained to accomplish this distribution, by distributing blocks round-robin to the processes in the order implied by the numbering shown in the figure. Moreover, when communication of fringes is required, this array is again referenced to determine to which process a fringe column or row needs to be sent.

6.3 Skyline Matrix Wavefront

A variation of a wavefront of this nature is a skyline matrix; a matrix in which each column and row has an indentation up to the diagonal with zero or more '0' values but has non-zero values thereafter. Specifically, there are values r_i with $1 < r_i < i$ such that each row i has values of '0' for each element up to element r_i, and non-zero elements thereafter. Similarly, there are values c_i for each column with the same conditions. An example of a skyline matrix is Figure 2.

There is code included for an MPI/C implementation of the Skyline Matrix Solver implemented in CO_2P_3S in [1]. It remains to generalize this pattern to a skyline matrix wavefront pattern, if such a pattern is warranted.

6.4 Example Problem

The example problem used for this pattern is Sequence Alignment; a dynamic programming algorithm used to determine the similarity between two strings of characters.

7 Pipeline Pattern

7.1 Description

The pipeline pattern is adapted from Steve MacDonald's development of a Master-Worker state-based pipeline in [2]. Whereas his treatment focussed on an object-oriented design with a master process serving work to many slave processes, this design takes a different approach, involving functional programming and a peer-peer relationship between processes.

7.2 Details

Once again, a pattern design has been adapted from a previous design; in this case we build a state-based pipeline on top of a distributed queue-based task parallelism system. A `Task` in this pattern consists of input data and an index of the pipeline stage that the data is prepared for. A `Stage` is defined using structures, which store the size of input data, and pointers to packing and unpacking functions, as well as a pointer to the execution function. Information about all stages are stored in a global array, and the index stored in a `Task` indexes that global array.

7.2.1 Modifications

No modifications to the distributed queue were required, as the pattern has been built strictly on top of a peer-peer task distribution system. In fact, the pipeline is independent of the distribution system; so much so that, as long as the programming interface of the distribution remained the same, the implementation of that system (master-worker vs. peer-peer, shared queue vs. distributed queue, etc.) could change without affecting the pattern code.

7.2.2 Hooks

Defining pipeline stages as structures with function pointers to necessary functions makes it possible to separate the user's contribution to the program from the pattern code. Unfortunately, this leads to a conundrum about the framework hook. On the one hand, to give the user more freedom, we could allow the user to write the `processTask` hook of the distributed queue pattern. This, however, does not allow us to take advantage of constructs built into the pipeline pattern above what is provided by the distributed queue. On the other hand, we could remove the hook completely, and simply have the processed task read the stage and pass the data to the indicated stage execution function. This second option limits the freedom of the user in manipulating stage data. At present, to maintain a pattern structure, the latter option is implemented, with the user only writing stage functions and defining the stages for the global array.

7.3 Example Problem

To demonstrate the pipeline pattern, we implement the map overlay problem; the same problem explored in [1]. Given two equal-sized maps containing non-overlapping polygons, the map overlay problems asks what polygons are generating when overlapping the two maps. For simplicity, we assume rectangular map sections which cover the entire extent of the map.

8 Future Work

8.1 C++

An important goal for future work with this project is using C++ rather than C as the basic programming language. During the course of this work, the value of using object-orientation in pattern-based parallel programming has been made clear by the language constructs that are lacking in C. Moreover, C++ is a language used more readily by those in the field, making it more attractive as the basis for a parallel programming system.

8.2 Macros

As was discussed in section 2, the use of macros as framework hooks, while better for performance, are not an ideal design solution to our problems. In expanding on this work, finding a better solution for this design issue will be necessary.

8.3 Performance Parameters

The focus of MAP_3S is not only providing users with a workable framework for parallel programming, but also to give users the power to obtain the best performance from their programs. To do so requires a study of places in the code where performance-affecting choices can be made, and the extrapolation of these choices into parameters the pattern user can choose to adapt the framework to the system and problem being completed.

8.4 Pattern Database

Simply put, the more patterns that are provided by MAP_3S , the more relevant MAP_3S will be as a parallel programming system. To that end, one area of future work would focus on looking at more patterns and, in so doing, expanding the scope of MAP_3S' applicability. Creating an actual database, as well, consisting of a defined data structure to delineate pattern code and parameters, that would be easily referenced by a code generation tool, is also an important goal.

8.5 Code Generation Tool

The most important aspect of this work that remains is an actual code generation tool, equivalent to the CO_2P_3S GUI that is provided with that tool. The user interface for such a tool, described using man pages, has been completed already. Combining these specifications with a well-defined pattern database structure would create a command-line interface tool that could be used to turn MAP_3S from prototyped code into a full-fledged parallel programming system.

References

[1] J. Anvik. Asserting the utility of CO_2P_3S using the Cowichan problems. Master's thesis, University of Alberta, Edmonton, AB, Canada, Dept. of Computing Science 2002.

[2] S. MacDonald. Transforming an object-oriented pipeline to a master-worker: The state-based pipeline. In *Workshop on Patterns in High Performance Computing*, Urbana, Illinois, USA, May 2005.

[3] S. MacDonald, J. Anvik, S. Bromling, J. Schaeffer, D. Szafron, and K. Tan. From patterns to frameworks to parallel programs. *Parallel Computing*, 28(12):1663–1683, 2002.

[4] P. Mehta, J. Amaral, and D. Szafron. Is MPI suitable for a generative design-pattern system? In *Workshop on Patterns in High Performance Computing*, Urbana, Illinois, USA, May 2005.