The MAP$_3$S Static-and-Regular Mesh Simulation and Wavefront Parallel-Programming Patterns

Robert Niewiadomski, José Nelson Amaral, Duane Szafron
{*niewiado, amaral, duane*}*@cs.ualberta.ca*
Department of Computing Science, University of Alberta
Edmonton, AB, Canada

***Abstract*— This paper presents the Simulation and Wavefront parallel-programming patterns of the MAP$_3$S pattern-based parallel programming system for distributed-memory environments. Both patterns target iterative computations on static-and-regular meshes. In addition to providing performance-oriented features, such as asynchronous communication and distribution of the computational workload that is tailored to fit the computation, the patterns also provide usability-oriented features, such as direct mesh-access, mesh memory-footprint distribution, and a versatile data-dependency specification scripting-language. Parallel programs developed using MAP$_3$S achieve significant performance gains and capability enhancements on both low-end and high-end interconnect-equipped distributed-memory systems.**

I. Introduction

Parallel computing is an important tool in many areas of scientific research. Perhaps nowhere is this more evident than in the study of physical phenomena and the design of complex engineered systems. Examples include aeronautics, astronomy, biology and meteorology. In general, parallel computing facilitates the simulation of physical phenomena whose investigation by other means is infeasible or too expensive.

The processing and memory demands of parallel scientific computations continue to increase. Consequently, scientists and engineers have turned to distributed-memory systems for their parallel computations, because such systems offer large and scalable aggregate-processing and -memory capacities, while being affordable. Unfortunately, the complexity of distributed-memory systems makes them difficult to program. In practice, developing performance-efficient parallel programs for distributed-memory systems is a daunting task that requires large amounts of time, effort and expertise.

A pattern-based parallel programming system can reduce the programming complexity of distributed-memory systems. A parallel-programming pattern encapsulates the communication and synchronization of a parallel program. The observation that several seemingly different scientific applications share a common computation-communication-synchronization pattern is the key insight that motivates the development of pattern-based parallel programming. This programming model decomposes the complexity of parallel application development across individuals or groups who play three roles: engine designer, pattern designer, and application developer. The amount of required technical expertise in programming distributed-memory systems decreases across the three roles, while the amount of problem domain knowledge increases, as illustrated in Figure1. Application developers leverage the parallel/distributed expertise of the developers whose role is earlier in the chain so that they can focus on the application-domain specific nature of tasks rather than the details of how to implement the necessary parallel/distributed computations. Since this process is designed to aid application developers, they are referred to as "users".[1] A user selects an appropriate parallel-computation pattern to generate an application program and adapts it by writing the actual sequential computation that takes place on each processing node. The coordination of activities between processing nodes — communication and synchronization — is handled by the engine code according to the specifications recorded by the pattern designer.

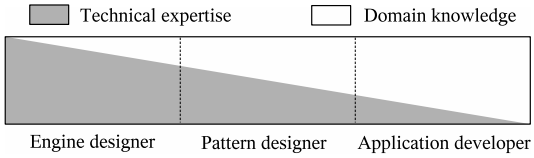


Fig. 1. A conceptual breakdown of the relative amounts of technical expertise and problem-domain knowledge for each of the three roles in the development process using a pattern-based parallel-programming system.

This paper presents two new parallel-programming patterns in the MPI/C Advanced Pattern-based Parallel Programming System (MAP$_3$S): the Simulation pattern and the Wavefront pattern. Both patterns target iterative computations on static-and-regular meshes. The mesh is static in the sense that its dimensions are fixed, and regular in the sense that it has an element for each element in the volume of the mesh. The Simulation pattern targets an iterative computation where the current iteration uses the values computed in the previous iteration as input. Examples of such iterative computations are found among cellular-automata algorithms such as Conway's Game of Life. In the Wavefront pattern, a function is computed at each element in a mesh. However, the computation of the value of an element uses the values of certain other mesh elements. Therefore, the computation is done in regions, where the computation for a region uses the values at other regions as input. Examples of Wavefront computations are found in dynamic-programming algorithms such as the Matrix Chain Multiplication algorithm. MAP$_3$S provides advanced

[1]The term "end-user" refers to the user of the application being developed.

performance-oriented features, such as asynchronous communication, and tailored distribution of the computational workload. MAP$_3$S also features powerful usability-oriented features, such as direct mesh-access, automatic distribution of the mesh memory-footprint, and a scripting language for the specification of data-dependencies. The main contributions of this paper are:

- A demonstration that generative pattern systems can be implemented effectively using the combination of MPI and C.
- A simple, yet very expressive, data-dependency-specification language that frees the pattern user from being limited to using pre-defined data-dependencies.
- Two very reasonable patterns and the possibility of reusing the engine code for the creation of more patterns. Moreover, the development of the engine code is independent of the patterns: the engine can be swapped for a better engine without changing the patterns.
- An experimental evaluation demonstrating that MAP$_3$S is successful not only at producing performance-gains, but also at producing capability-gains by way distributing the memory footprint of the computation, thereby allowing for the handling of computations that are limited by space as much as by time.

The remainder of this paper is organized as follows. Section II, presents a pattern-user oriented overview of the patterns. Section III presents a pattern-developer oriented overview of the patterns. Section IV presents an experimental evaluation of the patterns. Section V presents a summary of related work. Section VI presents the conclusions of this paper.

II. The Pattern-User Perspective

This section describes the use, structure, and main features of the Wavefront and Simulation patterns.

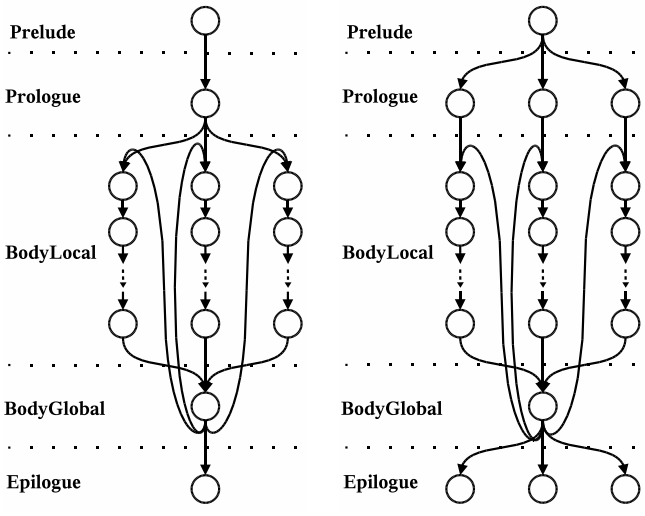
A. Using the patterns

The use of a pattern involves the execution of a *pattern-instance generator*. The generator reads a *pattern-instance specification-file* with parameters such as the identity of the pattern, the number of dimensions of the mesh, and the data dependencies governing the computation.

An instance of a pattern is a collection of source-code files organized into the categories of *user-side code* and *engine-side code*. The engine-side code carries out the parallel computation defined in the user-side code. The pattern-instance generator generates the engine-side code and a template code for the user-side. This template code contains function and data-structure declarations. The user defines the parallel computation by providing definitions for the functions and data-structures whose declarations are in the user-side code templates.

B. The Simulation pattern user-side code

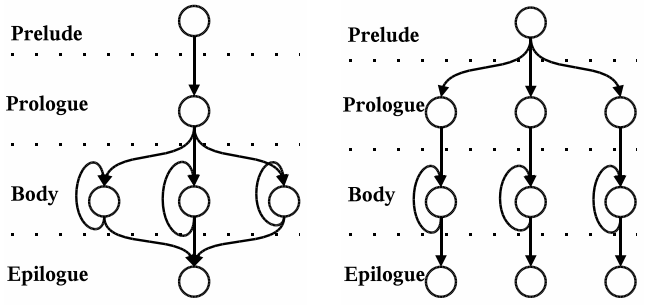
The user-side code of an instance of the Simulation pattern is based on five phases of computation: *prelude*, *prologue*, *body-local*, *body-global* and *epilogue*. Each phase has a corresponding function in the user-side code.



(a) Sequential prologue and sequential epilogue.

(b) Parallel prologue and parallel epilogue.

Fig. 2. The Simulation pattern on a distributed-memory system consisting of three nodes.



(a) Sequential prologue and sequential epilogue.

(b) Parallel prologue and parallel epilogue.

Fig. 3. The Wavefront pattern on a distributed-memory system consisting of three nodes.

The prelude executes on a single node to determine the size of the mesh along each dimension. The prologue computes the first instance of the mesh. As shown in Figure 2, the prologue may be executed on a single node or on all nodes in parallel. The body-local, executed in parallel, computes the next instance of the mesh based on portions of the previous instance of the mesh. The body-global is executed on a single node to compute the termination condition. The epilogue uses the last instance of the mesh to produce the application's output. The epilogue can be executed on a single node or in parallel. Figure 4 shows the template declarations of user-side functions corresponding to the five phases of computation. The parameters in these declarations highlight the ability to forward information from one phase to another.

The mesh is logically partitioned into *mesh blocks*. A mesh block is an n-dimensional sub-mesh, where n is the number of dimensions of the mesh. The size of a mesh-block along each dimension is uniform across all dimensions. Mesh blocks located at the boundaries of the mesh may be smaller due to

```
Action_t Prelude(i32_t argumentCount, i8_t **argument, MeshDescriptor_t *meshDescriptor, PreludeData_t *preludeData);

Action_t Prologue(MeshElement_t **firstMesh, MeshDescriptor_t meshDescriptor, PreludeData_t *preludeData,
                  PrologueData_t *prologueData);

Action_t BodyLocal(MeshElement_t **currentMesh, MeshElement_t **previousMesh, MeshDescriptor_t meshDescriptor,
                   MeshBlockDescriptor_t meshBlockDescriptor, PreludeData_t *preludeData, PrologueData_t *prologueData,
                   BodyGlobalData_t *bodyGlobalData, BodyLocalData_t *bodyLocalData, i32_t iteration);

Action_t BodyGlobal(MeshDescriptor_t meshDescriptor, i32_t meshBlockCount, MeshBlockDescriptor_t *meshBlockDescriptor,
                    BodyLocalData_t *bodyLocalData, PreludeData_t *preludeData, PrologueData_t *prologueData,
                    BodyGlobalData_t *bodyGlobalData, i32_t iteration);

Action_t Epilogue(MeshElement_t **lastMesh, MeshDescriptor_t meshDescriptor, PreludeData_t *preludeData,
                  PrologueData_t *prologueData, BodyGlobalData_t *bodyGlobalData, i32_t lastIteration);
```

Fig. 4. User-side functions for the Simulation pattern using a two-dimensional mesh, sequential prologue, sequential epilogue and dense mesh-storage. Use of a parallel prologue requires the arguments `i32_t MeshBlockCount` and `MeshBlockDescriptor_t *meshBlockDescriptor` in the `Prologue` function. Use of a parallel `Epilogue` also requires these arguments. Use of sparse mesh-storage requires the argument `MeshElement_t **` to change to `MeshElement_t ****`.

truncation. Each node is assigned one or more mesh blocks. The degree of parallelism achieved is limited by the data dependencies that govern the computation.

C. The Wavefront pattern user-side code

The single *body* phase in the Wavefront pattern (see Figure 3) executes on all nodes in parallel to compute values for different regions of the mesh. Each node has access to the regions of the mesh that it uses. Figure 5 shows the declarations of the user-side functions for this pattern. Information is passed from one phase to another via parameters. The computation is carried out at the granularity of a mesh block.

D. Data dependencies

The mesh computations are governed by a set of data dependencies. For simplicity, in the discussion that follows, "element" refers to the value of a mesh element. The computation of an element cannot proceed until all the elements that it uses are available. In the Simulation pattern, successive instances of the mesh are computed. The computation of the next instance does not proceed before the current instance is finished. The computation of elements in each instance can use any elements from the previous instance, but none from the current one. Therefore, the order in which elements are computed during a particular iteration does not matter. However, in the Wavefront pattern, only a single instance of the mesh is computed. Elements in the single instance of the mesh may depend on other elements in this instance. Therefore data dependencies impose a partial-order constraint on the computation of the elements.

In MAP_3S the pattern user specifies the set of data-dependencies in a pattern-instance specification file. Such a specification should be straightforward to read/write and sufficiently expressive to permit the specification of both conventional and non-conventional data-dependencies. To meet this criterion, MAP_3S implements a language based on *conditional shape-lists* for data-dependency specification.

A conditional shape-list consists of two parts: the *condition* and the *shape list*. The condition is a C language expression that evaluates to true or false. The shape list is a list of shapes defined by *shape descriptors*. A shape descriptor for an n-dimensional mesh consists of n pairs of values, each specifying a range in dimension i, $0 \leq i < n$. For instance, if the mesh has two dimensions then each shape descriptor is of the form $([x_0, x_1], [y_0, y_1])$. Such a descriptor defines a rectangular area of the mesh limited by the specified range-boundaries. Each boundary is specified by a C expression that evaluates to an integer. The C expressions of conditions and shape descriptors can use integer variables called *mesh-element-position variables*, which describe the position of a mesh element. For instance, if the mesh has two dimensions, then the C expressions can use the mesh-element-position variables `x` and `y`, for the x-axis and y-axis positions of the mesh element. The C expressions can also use integer variables describing the sizes of the mesh along each of its dimensions. For a two-dimensional mesh, the variables `maxX` and `maxY` can be used.

```
{y > x , {([0, x - 1], [y, y]), ([x, x], [0, x - 1]),
          ([x, x], [x, x])}};
{y <= x, {([0, y - 1], [y, y]), ([x, x], [0, y - 1])}};
```

Fig. 6. Data dependencies of the Lower/Upper Matrix Decomposition problem as specified using conditional shape-lists.

A conditional shape-list defines the set of data dependencies for the computation as follows. The set of mesh elements that are used in the computation of a mesh element p is obtained by replacing the coordinates of p for the mesh-element-position variables in the C expressions of the conditional shape-list. Whenever a condition evaluates to true, the computation of p depends on all of the elements specified by its shape descriptors.

Conditional shape-lists can express non-trivial sets of data dependencies. For instance, the Lower/Upper Matrix Decomposition problem is solved using a dynamic-programming algorithm that involves a square two-dimensional mesh and the following set of data dependencies. Given two mesh elements $e = (x, y)$ and $e' = (x', y')$, if e is at or above the diagonal of the mesh, then the computation of e uses: every element e' where $0 \leq x' \leq y - 1$ and $y' = y$, and every element e' where $x' = x$ and $0 \leq y' \leq y - 1$. If e is below the diagonal of the mesh, e depends on: every element e where $x' = x$ and $y' = x$, every element e' where $0 \leq x' \leq x-1$ and $y' = y$, and every element e' where $x' = x$ and $0 \leq y' \leq x - 1$. Figure 6 shows how this set of data-dependencies can be expressed using conditional shape-lists.

```
Action_t Prelude(i32_t argumentCount, i8_t **argument, MeshDescriptor_t *meshDescriptor, PreludeData_t *preludeData);

Action_t Prologue(MeshElement_t **mesh, MeshDescriptor_t meshDescriptor, PreludeData_t *preludeData,
                  PrologueData_t *prologueData);

Action_t Body(MeshElement_t **mesh, MeshDescriptor_t meshDescriptor, MeshBlockDescriptor_t meshBlockDescriptor,
              PreludeData_t *preludeData, PrologueData_t *prologueData, BodyData_t *bodyLocalData);

Action_t Epilogue(MeshElement_t **mesh, MeshDescriptor_t meshDescriptor, PreludeData_t *preludeData,
                  PrologueData_t *prologueData);
```

Fig. 5. User-side functions for the Wavefront pattern using a dense two-dimensional mesh, sequential prologue, sequential epilogue and dense mesh-storage. Use of a parallel `Prologue` requires the arguments `i32_t MeshBlockCount` and `MeshBlockDescriptor_t *meshBlockDescriptor`. Use of a parallel `Epilogue` also requires these arguments. Use of sparse mesh-storage requires the argument `MeshElement_t **` to change to `MeshElement_t ****`

MAP_3S uses element-wise data-dependencies to compute data dependencies at the mesh-block granularity. The computation of a mesh block does not proceed until the computation of all the blocks that it uses is finished. The conversion of element-wise dependencies to block-level dependencies may generate a cycle in the block-level data-dependency graph which would make it impossible for a schedule to be generated. The engine-side code detects cyclic dependencies and generates an error message. Despite the potential for cyclic dependencies, the mesh computations that are targeted by the Wavefront pattern are unlikely to feature cyclic dependencies.

E. Mesh representation, access and memory-footprint distribution

MAP_3S offers two mesh-representations: *dense mesh-storage* and *sparse mesh-storage*. The dense mesh-storage resperentation is a conventional mesh-representation. The sparse mesh-storage representation introduces a level of abstraction that partitions the mesh into sub-meshes corresponding to mesh blocks. The indirection required to access a sparse mesh-storage representation involves the use of simple bit operation, namely bit-shifts and bit-masks. For convenience, the user has access to a macro function to access sparse meshes that hides the bit operations.[2] The motivation behind offering the two mesh-representations is that the dense mesh-storage representation is familiar to most users, while the sparse mesh-storage representation affords advanced users with the opportunity to potentially improve data-reference locality and to benefit from the distribution of the mesh memory-footprint among the nodes of the system.

MAP_3S provides the user-side code with *direct mesh-access*. This differs from the approach commonly employed in other pattern-based parallel-programming systems, whereby the user-side code uses *indirect mesh-access*. With indirect mesh-access, the user-side code accesses a dedicated data-structure representing the sub-mesh and a dedicated data-structure representing other regions of the mesh that are required to carry out the computation. The difficulty with indirect mesh-access is in the expression of the computation for each element of the mesh. If the user starts with a sequential version of the mesh computation, the conversion of the code to access indirect mesh-access data-structures may be non-trivial and error-prone, thus hindering productivity. The more complex data-structures may also reduce the success of compiler optimizations and prevent vectorization. In contrast, with direct mesh-access each mesh element can be directly referenced by the user-side code.

Direct mesh-access has the following semantics. When computing elements of a mesh block, the user-side code can read any element from the current mesh-block and from mesh blocks that contain elements that it uses, as specified by the pattern data-dependencies. It can only write to elements of the current mesh-block. When the computation of a mesh block is scheduled, all elements that the mesh block needs are guaranteed to be up to date.

In the mesh computations that are targeted by MAP_3S the memory footprint of the computation can be as much a limiting factor as the execution time. This is illustrated by the Needleman-Wunsch dynamic-programming algorithm for computing an optimal alignment of genetic sequences. A sequential implementation of this algorithm finishes in a few seconds on a contemporary workstation even when the mesh occupies the entire memory. A similar scenario plays out in cellular-automata algorithms where there is an inverse relationship between the granularity of the simulation and the size of the mesh. Moreover, the accuracy of simulation results is often a function of granularity and, consequently, mesh size. Thus, MAP_3S implements automatic distribution of the mesh memory-footprint when using sparse mesh-storage.

Distribution of the mesh memory-footprint allows the solution of problems that are too large to fit in the memory of a single workstation. The performance yielded by existing distributed-shared memory software packages, *i.e.* to distribute the mesh memory-footprint, suffers because of large communication-latencies. In contrast, MAP_3S leverages data-dependency information to produce an execution schedule for the user-side code that effectively hides large communication-latencies. The engine-side code constructs an asynchronous-communication pipeline that is aware of data-dependency relationships. This pipeline is not only proficient at hiding communication latency in the presence of irregular data dependencies but also strives to minimize the size of communication buffers.

III. The Pattern-Developer Perspective

This section describes the execution of the engine-side code, its distribution of the computational workload, and the computation and communication schedule.

[2] In the future, MAP_3S is likely to provide support for the C++ programming language, in which case the usability of sparse mesh-storage would improve with operator overloading.

A. The execution of the engine-side code

The execution of the engine-side code consists of two phases: preprocessing and processing. During preprocessing, the engine-side code:

1) obtains the size of the mesh along each of its dimensions by executing `Prelude`;
2) processes the conditional shape-lists to determine data-dependency relationships;
3) maps mesh blocks to nodes, based on the data-dependency relationships;
4) creates a computation and communication schedule to drive the execution of either `BodyLocal` and `BodyGlobal` for the Simulation pattern, or `Body` for the Wavefront pattern; and
5) allocates two instances of the mesh for the Simulation pattern, or a single instance of the mesh for the Wavefront pattern.

During processing, the engine-side code executes `Prologue` and then it follows the computation and communication schedule to execute either `BodyLocal` and `BodyGlobal` for the Simulation pattern, or `Body` for the Wavefront pattern, and, finally, it executes `Epilogue`.

B. Distribution of the computational workload

The engine-side code uses the data-dependency relationships to assign each mesh block to a node. First each mesh block b is assigned a label. In this assignment, b is said to *use* a mesh block b' if and only if one or more elements in b uses one or more elements in b'. The assignment proceeds as follows. If b does not use itself or any other mesh block and is not used by itself or any other mesh blocks, then b is a *dead mesh-block*. Otherwise, b is an *alive mesh-block*. An alive mesh-block that does not use itself or any other mesh block is an *alive/non-computed mesh-block*. Otherwise, it is an *alive/computed mesh-block*.

In the Simulation pattern, the engine-side code evenly partitions all alive/non-computed mesh-blocks according to their positions in the mesh to obtain n sets of mesh blocks, where n is the number of nodes. The engine-side code then assigns all mesh blocks in the i-th set of mesh blocks to the i-th node. This partitioning and assignment is repeated for all alive/computed mesh-blocks. A similar approach is used in the Wavefront pattern.

The labeling of mesh blocks saves time and space. Time is saved since no computations are performed on dead mesh-blocks and only a single computation is performed on each alive non-computed mesh-block (during the prologue). Space is saved because dead mesh-blocks do not need to be stored. For example, the engine-side code is more efficient on the Matrix Chain Multiplication problem, which is solved using a dynamic-programming algorithm, because mesh elements below the diagonal are not computed; mesh elements on the diagonal are only initialized; and mesh elements above the diagonal are computed. In expressing the data-dependencies for this algorithm, the user can force mesh blocks below the diagonal to be ignored completely. The user simply creates conditionals that evaluate to false for mesh elements below the diagonal.

C. The computation and communication schedule

The schedule is constructed using a *mesh-block data-dependence graph*.[3] In this directed acyclic graph, each vertex represents an alive/computed mesh-block and each arc represents a data dependency, where the mesh block at the head of the arc uses the mesh block at the tail. The graph can be organized into distinct levels, where the topmost level contains every vertex with no incoming arcs. Note that there must be at least one such vertex, since the graph is finite and has no cycles. The next level consists of every vertex that has incoming arcs only from vertices in the top level. There must be at least one node in each level because otherwise the graph would have cycles. This level-assignment algorithm continues until there are no more vertices to be placed in a level. The algorithm must end since the graph is finite.

In the Simulation pattern, the schedule is built around a while loop whose iteration count is infinite. In each iteration of the while loop, the computation portion of the schedule drives the engine-side code to execute `BodyLocal` on each alive/computed mesh-block assigned to the local node, and then calls `BodyGlobal` if the local node is the root machine. Each iteration of the while loop performs communication operations involving mesh-block data that is produced and consumed by `BodyLocal`. These communication operations comprise the asynchronous-communication pipeline of the Simulation pattern. In each iteration i, the engine-side code completes sending operations and receiving operations that it initiated in iteration $i-1$, and initiates sending operations and receiving operations that it will complete in iteration $i+1$. In addition, each iteration of the while loop performs communication operations involving control data as well as auxiliary data produced and consumed by `BodyLocal` and `BodyGlobal`.

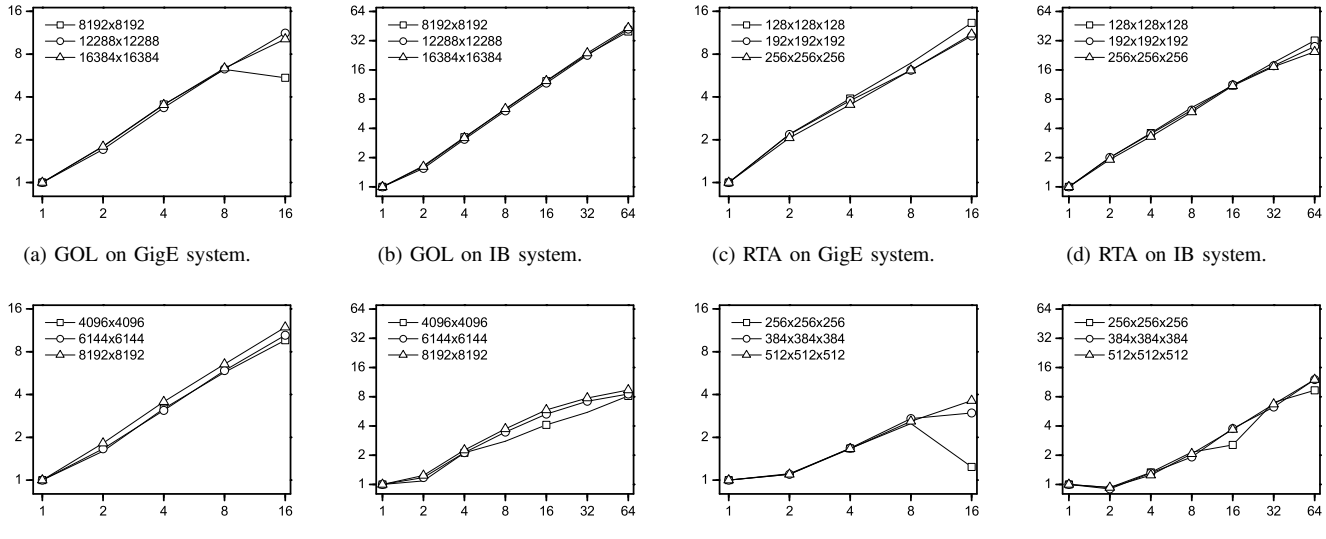
In the Wavefront pattern, the schedule is built around a for loop whose iteration count is equal to the number of levels of the mesh-block data-dependence graph. Iteration i of the for loop executes `Body` on each alive/computed mesh-block that is at level i in the data-dependence graph and is assigned to the local node. The communication schedule in the Wavefront pattern uses a pipeline similar to that of the Simulation pattern.

IV. Experimental Evaluation

This section assesses both the *performance* of applications generated by MAP_3S and MAP_3S' *capability* in solving larger instances of problems than sequential implementations. The main findings of this experimental evaluation are:

- With dense mesh-storage, on problems with sufficiently large granularity of computation, MAP_3S delivers speedups in the range of ≈ 10 to ≈ 12 on a 16-node

[3]Because the computation of mesh elements in the Simulation pattern is not constrained by data dependencies, the graph is only used to create a communication schedule for the Simulation pattern.



(a) GOL on GigE system. (b) GOL on IB system. (c) RTA on GigE system. (d) RTA on IB system.

(e) LUMD on GigE system. (f) LUMD on IB system. (g) MSA on GigE system. (h) MSA on IB system.

Fig. 7. Performance on three instances of each problem on GigE and IB. Speedup over sequential implementation (y-axis) is plotted against the number of nodes (x-axis).

system, and in the range of ≈ 10 to ≈ 43 on a 64-node system.

- With sparse mesh-storage, MAP$_3$S delivers speedups that are smaller than those obtained with dense-mesh storage, but does so while consuming between $\approx 20\%$ to $\approx 50\%$ less memory on a per node basis, which allows for the solving of problems that are too large to be solved using the sequential implementation.

A. Software and hardware

Implementations for four applications were developed using MAP$_3$S: Room-Temperature Annealing (RTA) and Game-of-Life (GOL) using the Simulation pattern; and Lower/Upper Matrix Decomposition (LUMD) and the Multiple-Sequence Alignment (MSA) using the Wavefront pattern. RTA features floating-point aithmetic and has immediate-neighbor data dependencies on a three-dimensional non-toroidal mesh. Both GOL and MSA feature integer arithmetic and immediate-neighbor data dependencies. GOL uses a two-dimensional fully-toroidal mesh and MSA uses a three-dimensional non-toroidal mesh. LUMD features floating-point arithmetic and has non-trivial data-dependencies (see Figure 6) on a two-dimensional non-toroidal mesh.

The sequential implementation of each application is straightforward and representative of, what we believe to be, average programming-skill. These sequential implementations were the basis for the user-side code of each corresponding MAP$_3$S implementation.

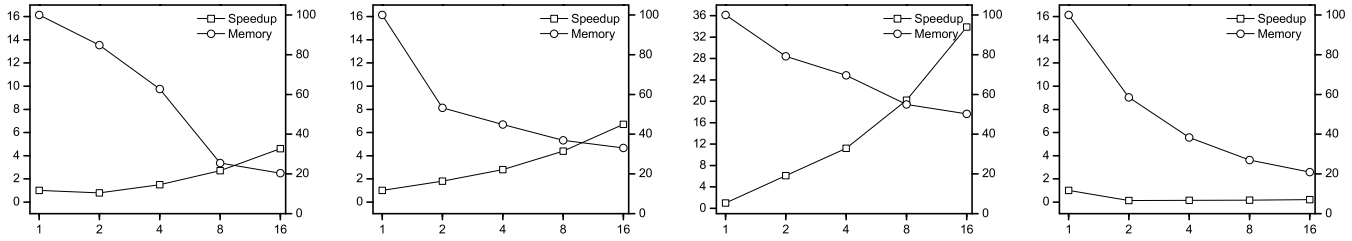
Two systems are used for performance evaluation: GigE and IB. GigE has 16 dual AMD Opteron 248 nodes with 5 GB of RAM and a single-switch Gigabit Ethernet interconnect. IB has 128 dual AMD Opteron 250 nodes with 2 GB of RAM and a 4X SDR Infiniband interconnect with multiple-switches in a fat-tree configuration. GCC at level `-O3` is used on both systems. GigE uses MPICH2 (MPI-2 standard) and IB uses the HP implementation of the MPI-1 standard. GigE was used in exclusive mode (no other activities in the machine). IB has a shared-access policy. The experiments had dedicated access up to 64 of the 128 nodes. Nonetheless, there was interconnect contention from other activities during the parallel runs.

B. Performance

Figure 7 shows the speedups of parallel implementations using a parallel prologue, a parallel epilogue and dense mesh-storage, with 4096-element mesh blocks. Each curve represents a different mesh size. With the exception of MSA, MAP$_3$S achieved consistent performance gains on both systems. In MSA the granularity of the computation is too small: a sequential run of the largest MSA instance runs in less than 5 seconds.

The use of 4096-element mesh-blocks was effective for all applications other than LUMD. In all applications the amount of computation and communication on an element varies depending on the element's location in the mesh. However, the magnitude of this variance is much larger in LUMD. The solution is to decrease the mesh-block capacity for LUMD [4], [1]. The use of a 1024-element mesh-block had no effect on performance on the GigE system, but did increase performance on the IB system, where it brought up the 32 node speedup to ≈ 20. A 256-element mesh-block decreased performance on the GigE system but increased performance on the IB system, where it brought up the 32 node speedup to ≈ 15.

All four applications are communication intensive, thus coping with communication latency is important. Except for LUMD, MAP$_3$S achieves a balanced distribution of the computational workload. Under these circumstances, discrepancies between the wall-clock and process-clock times reliably assesses successful latency hiding. In the experiments ran on



(a) GOL: 16,384 x 16,384. (b) RTA: 256 x 256 x 256. (c) LUMD: 8,192 x 8,192. (d) MSA: 512 x 512 x 512.

Fig. 8. Performance and capability on the largest instance of each problem on GigE. Speedup over sequential implementation (left y-axis) and percentage of sequential implementation memory-consumption (right y-axis) is plotted against the number of nodes (x-axis).

the IB system, wall-clock time is, on average, less than $\approx 1\%$ larger than process-clock time. For the GigE system wall-clock time is, on average, $\approx 28\%$ larger than the process-clock time. Thus, it appears that MAP_3S is efficiently handling latency on the IB system but not on the GigE system. It appears that the culprit is the communication in the body-global phase — a single gather and a single scatter. Removing this communication, for investigation, in GOL and RTA renders the discrepancies virtually non-existent.

The parallel implementations of GOL and RTA incur an overhead for packing/unpacking mesh data. A sending node examines a bit vector to identify the mesh data to be sent and copies this data into a communication buffer. Conversely, the receiving node unpacks the mesh data by examining a bit vector to identify the received mesh data and copy the data from a communication buffer to the mesh. In the future better algorithms may reduce this communication overhead.

C. Capability

Two set of experiments study the capability of MAP_3S to solve large problem instances: *large sequential* and *large distributed*. The large sequential experiments solve large problem-instances that can be solved sequentially. The corresponding parallel versions use parallel prologue, parallel epilogue and the sparse mesh-representation. These experiments assess the effectiveness of MAP_3S at distributing the memory footprint of the comutation, and the performance overhead incurred by distribution. The large parallel experiments solve large problem-instances that cannot be solved sequentially because of the limited memory capacity of a single node. These experiments assess MAP_3S capability to distribute the memory footprint of the computation.

Both sets of experiments measure the maximum memory-consumption on any node. They do so not as the maximum memory-footprint of the program on any node, but as the maximum system-memory-footprint of any node, which is the sum of all memory footprints of all running programs on a node, including the operating system. This approach provides a more accurate comparison because in executing a parallel program, the operating system requires communication buffers and control data-structures that are not required when running a sequential program.

Figure 8 presents the speedups and maximum memory-consumption for the large sequential experiments. The maximum memory-consumption is reported as a percentage of the maximum memory-consumption of the sequential implementation. MAP_3S effectively distributes the memory footprint for all problems. After certain number of nodes are used, the memory usage in each node stops decreasing. This phenomenon is especially pronounced on LUMD because of the large number of data dependencies — especially for elements at the lower-right hand side of the mesh. Moreover, the distribution of the memory footprint significantly improves performance on the LUMD problem while significantly degrading performance on the other problems. This performance improvement on the LUMD problem appears to be due to better data-reference locality. Indeed when we re-wrote the sequential version of LUMD to use the sparse mesh-representation in combination with mesh-block granularity processing, the performance of the sequential version improved by a factor of ≈ 4.

The distribution of the mesh memory-footprint in the large distributed experiments (see Figure 9) allows MAP_3S to solve instances of the GOL, RTA and MSA with a 32 GB mesh memory-footprint, and an instance of LUMD with a 12 GB mesh memory-footprint. The GOL and MSA problems were most amenable to mesh memory-footprint distribution, followed by the RTA problem, and followed far behind by the LUMD problem.

V. Related Work

MAP_3S is a derivative of another pattern-based parallel-programming system called the Correct Object-Oriented Pattern-Based Parallel Programming System (CO_2P_3S) [5].[4] Like CO_2P_3S, MAP_3S expresses parallelism using a template-oriented approach [8]. The development of MAP_3S was spurred by the desire to address the principal limitations of CO_2P_3S. The limitations in question are performance and scalability. In particular, CO_2P_3S makes use of Java and targets shared-memory systems. Although a later extension of CO_2P_3S provides support for distributed-memory systems, the resulting performance and scalability is limited [9]. For this reason, MAP_3S makes use of C, in combination with MPI, and targets distributed-memory systems. In addition to striving to address the performance and scalability limitations of CO_2P_3S,

[4]For an extensive listing of CO_2P_3S publications see *http://www.cs.ualberta.ca/ systems/cops/publications.html.*

Problem Instance	Global Mesh Memory-Footprint (GB)	Maximum Local Memory-Consumption (GB)	Process-Clock Execution-Time (Hours)	Wall-Clock Execution-Time (Hours)
GOL 131,072 x 131,072	32.0	3.0	19.2	19.9
RTA 1,024 x 1,024 x 1,024	32.0	4.4	12.9	17.7
LUMD 40,132 x 40,132	12.0	3.0	16.9	19.4
MSA 2,048 x 2,048 x 2,048	32.0	3.0	0.08	0.16

Fig. 9. Pushing the limits of mesh memory-footprint distribution on the GigE system using 16 nodes.

MAP_3S strives to be more versatile with respect to supporting a wider range of parallel computations without requiring the involvement of the pattern-based parallel-programming system developer. The data-dependency specification language is one important example.

A preliminary investigation established that the combination of C and MPI enables MAP_3S to support parallel-programming patterns akin to those supported by CO_2P_3S [6]. This investigation was limited in scope. At that point, the engine was primitive and performance was evaluated only on a shared-memory system. Although the investigation addressed issues concerning programmability, it did not deal with the performance and capability issues.

It is difficult to assess the performance and capability of MAP_3S due to a lack of published comparable results. For example, CO_2P_3S results are not comparable since the performance and capability of CO_2P_3S was handicapped by its use of Java, rather than C. However, we can point to the work by Liu and Schmidt [4]. They describe performance gains for hand-crafted solutions to problems that could be solved using the Wavefront pattern, running on a Myrinet-based interconnect. MAP_3S generated code exhibits very similar performance gains to their hand-coded implementations.

A distinguishing usability-oriented feature of MAP_3S is its use of a versatile scripting-language for data-dependency set specification. This approach to data-dependency set specification is more practical than the approach used in CO_2P_3S, where the user had at their disposal several data-dependency sets in a database. If the desired data-dependency set was not in the database then the user could potentially derive it by extending a data-dependency set in the database. Because the degree to which the user could extend a data-dependency set is limited, the user may ultimately have to write code to add a new dependency or contact the CO_2P_3S development team in order to requisition the new data-dependency. A similar set of complications arises in less advanced bit-oriented methods, such as in the case of htalib [2].

Cole points out that despite years of research, pattern-based parallel programming systems continue to fail to become mainstream parallel-programming tools [3]. To that extent, Cole outlines a four-point manifesto that developers of pattern-based parallel programming systems should strive for in order to attain such mainstream status. Because each of the four points in Cole's manifesto is general, whether or not MAP_3S is successful in fulfilling the points is subjective. However, we believe that MAP_3S, in its current state, fulfills the principles of each point to some extent, especially in the case of the point concerning 'payback'.

An alternative to using a pattern-based parallel-programming system is to utilize a parallel programming language, such as ZPL, or a parallel programming library, such as htalib [2], [7]. Doing so typically involves leveraging data-level parallelism in array operations, whether through the use of language constructs, or relaying on automatic parallelization. In general, when the computation is well handled and recognized by the underlying system the resulting performance can be equivalent to that of a hand-coded effort.

VI. Conclusions

This paper describes an important step in the development of MAP_3S. The Simulation and Wavefront patterns are very widely used for the implementation of parallel applications. The demonstration that a generative pattern systems using MPI/C can produce significant performance- and capability-gains in distributed-memory environments is very encouraging. We suspect that the levels of performance and capability reported in this paper are on par with very good hand-crafted solutions for each problem. A usage study will be necessary both to test this hypothesis and to learn about the easy-of-use of MAP_3S by pattern designers and by pattern users that are unfamiliar with pattern-based parallel programming.

References

[1] John Anvik, Steve MacDonald, Duane Szafron, Jonathan Schaeffer, Steven Bromling, and Kai Tan. Generating parallel programs from the wavefront design pattern. In *6th International Parallel and Distributed Processing Symposium (IPDPS 2002)*, 2002. On CD.

[2] Ganesh Bikshandi, Jia Guo a nd Christoph von Praun, Gabriel Tanase, Basilio B. Fraguela, María Jesús Garzarán, David A. Padua, and Lawrence Rauchwerger. Design and Use of htalib - A Library for Hierarchically Tiled Arrays. In *Languages and Compilers for Parallel Computing, 19th International Workshop, LCPC 2006*, pages 17–32, 2006.

[3] Murray Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004.

[4] Weiguo Liu and Bertil Schmidt. Parallel design pattern for computational biology and scientific computing applications. In *2003 IEEE International Conference on Cluster Computing (CLUSTER 2003)*, pages 456–459, 2003.

[5] Steve MacDonald. *From patterns to frameworks to parallel programs*. PhD thesis, University of Alberta, Edmonton, AB, Canada, 2002, Department of Computing Science, 2002.

[6] Paras Mehta, José Nelson Amaral, and Duane Szafron. Is MPI Suitable for a Generative Design-Pattern System? *Parallel Computing*, 32(7-8):616–626, 2006.

[7] Lawrence Snyder. The design and development of ZPL. In *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference*, pages 1–37, 2007.

[8] Duane Szafron and Jonathan Schaeffer. An Experiment to Measure the Usability of Parallel Programming Systems. *Concurrency Practice and Experience*, 8(2):147–166, 1996.

[9] Kai Tan, Duane Szafron, Jonathan Schaeffer, John Anvik, and Steve MacDonald. Using generative design patterns to generate parallel code for a distributed memory environment. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, (PPOPP 2003)*, pages 203–215, 2003.