1st ICSE Workshop on Software Engineering for High Performance Computing in Science (SE4HPCS 2015), Florence, Italy, May 19 2015. © IEEE 2015

Supporting Scientists in Re-engineering Sequential Programs to Parallel Using Model-driven Engineering

Mohamed Almorsy and John Grundy
Centre for Computing and Engineering Software and Systems
Swinburne University of Technology, Hawthorn, Australia
malmorsy@swin.edu.au, jgrundy@swin.edu.au

Abstract—Developing complex computational-intensive and data-intensive scientific applications requires effective utilization of the computational power of the available computing platforms including grids, clouds, clusters, multi-core and many-core processors, and graphical processing units (GPUs). However, scientists who need to leverage such platforms are usually not parallel or distributed programming experts. Thus, they face numerous challenges when implementing and porting their software-based experimental tools to such platforms. In this paper, we introduce a sequential-to-parallel engineering approach to help scientists in engineering their scientific applications. Our approach is based on capturing sequential program details, planned parallelization aspects, and program deployment details using a set of domain-specific visual languages (DSVLs). Then, using code generation, we generate the corresponding parallel program using necessary parallel and distributed programming models (MPI, OpenCL, or OpenMP). We summarize three case studies (matrix multiplication, N-Body simulation, and digital signal processing) to evaluate our approach.

Keywords—component; Parallel Programming; High-Performance Computing; Domain-specific Visual Languages; Model-driven Engineering

I. INTRODUCTION

The Australian Square Kilometer Array, which will enable astronomers to survey the radio universe with unprecedented speed, is expected to generate terabytes to petabytes of data per day of observations [1]. Processing such big data requires developing large-scale parallel programs that can fulfill the task and produce meaningful outcomes in a reasonable time. Before developing such parallel programs, scientists and HPC experts usually start with a sequential version that solves the problem on a small scale [2]. This is often relatively easy and helps to understand implementation details. However, scaling up such sequential programs to work on big datasets and utilizing the computational power of today's heterogeneous platforms is a very challenging task for scientists because it requires special experience in HPC. On the other hand, the existing parallel programming models and languages, such as MPI, OpenMP, OpenCL, are suitable mainly for expert parallel programmers. Existing automated and user-aided parallelization efforts try to address this gap. However, they are either very low-level, too abstract, or very domain-specific. We categorize these efforts in:

(i) *Compiler-based parallelization* [3-5]: Try to pinpoint parallelizable code sections, usually loop unrolling, in the input program either automatic using static analysis, machine learning and profiling techniques, or explicitly via user specified compiler directives. Kravets et al. [4] introduce Graphite-OpenCL that automatically locates parallelizable loops. Such loops are turned into an OpenCL kernel, and all necessary OpenCL calls for creating and compiling kernels and copying data to/from the device are automatically generated. Similar work was introduced by compilers including Polaris [5], and SUIF [3]. These efforts lack context and program developers' intension information.

(ii) *Abstract modeling or domain-specific languages:* Deliver high level models and/or DSLs usually have implicit mappings to predefined parallel libraries without letting developers specify intended parallelization details [6] such as image processing [7], partial differential equations (PDE) [8] or machine learning [9]. Although, these efforts help in hiding parallelization details from the user, they are limited by the provided functionalities by such DSLs. Moreover, most are text-based with a specific syntax that the developer has to learn to use the language.

(iii) *Portable domain-specific languages* [10-14]: Focus on capturing program parallelization aspects using models, then generating parallel code targeting one or more of the parallel programming models. Jacbo et al. [10] focus on using abstract models to help porting to different computing platforms – e.g. refactoring parallel programs to use MPI instead of OpenMP and switching between OpenCL and CUDA. Han et al. [11] introduce a directive-based approach where developers can annotate their sequential program code. This looks very similar the OpenACC APIs. Dig et al [12] introduce a refactoring tool that help in automating the conversion of sequential programs into parallel program without defining any annotations. The tool is based on locating signatures of possible parallelization aspects (mainly three aspects were covered including the recursion). The approach is based on replacing such matched program constructs with their corresponding java parallel library implementations. Palyart et al. [13] introduce MDE4HPC approach a DSVL to help in specifying and modeling HPC applications. They focus on specification of solution parallelism. However, no code generation included. Jacob et al. also [14] introduce an IDE plugin to help programmers select code blocks and specify the computation device to run on. No support for heterogneous computing platform.

In this paper, we introduce a new approach to help programmers and scientists in effectively parallelizing their sequential programs. Our approach is based on a set of domain-specific visual languages to help in modeling program structure and input/output datasets. Two further DSVLs capture parallelization plans, and platform deployment details. Programmers/scientists do not need to have experience in parallel patterns and heterogeneous computing platforms. Using the parallel program model our toolset generates the necessary code utilizing these patterns and platform configurations. Generated code can be modified further, compiled and run on CPU grid and GPUs.

Our approach saves considerable effort required to migrate parallel programs from one computing platform to another. Thus, programmers and scientists can write a program, model its parallel and deployment aspects, and then get it to run on different computing platforms either single core, multi-core, many-core, or hardware accelerators by primarily updating program deployment details and having target code regenerated. We have evaluated our approach using several different scientific computation case studies including: general-purpose matrix multiplication, N-Body simulation, digital signal processing. Our approach is supported by a web-based tool that provides different features including: scientific DSVL design, code generation, code editing, parallel patterns reuse, reverse engineering, and data visualization.

A. Motivating Example

Multiplication of large matrices is a common problem in scientific computing (e.g. 1 million elements each). Figure **1** shows a sequential program model of the matrix multiplication developed using our parallel program designer tool. It initializes both A and B (init_matrixA, init_matrixB) and then comprises three nested loops (Row, Cols, element – Loop) applying the multiplication operation on rows and columns to calculate the value of $C_{i,j}$. Finally, we print matrix_C. We discuss below how end users (scientists and programmers) using our approach can develop different parallel versions of such a program.

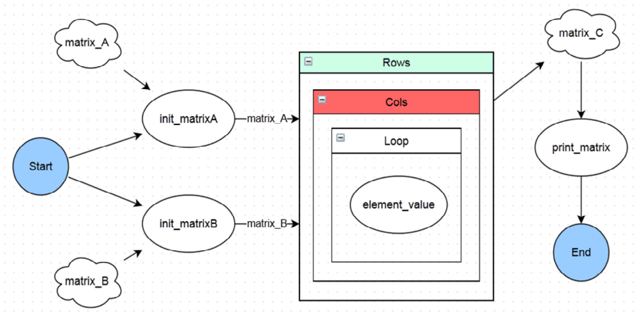


Figure 1. Example sequential matrix multiplication program

II. Parallel Program Designer

Our approach is based on capturing a sequential program definition such as the example in Figure **1**, then developing parallelization plans and deployment details. These details are used to generate the modeled parallel version of the program for further development. Our approach has three key DSVLs, summarized in Figure **2**: (i) SeqDSVL: a Sequential program description language. This includes: sequence, selection, repetition, tasks and data structures. These constructs are available for users to model sequential algorithms as well as developing reusable tasks frequently used in a given domain. (ii) ParaDSVL: Parallelization plan specification language. This includes operations necessary for data and task decomposition such as: split, join, parallel section, loop unrolling and parallel task. As discussed later, these basic constructs are sufficient in capturing parallelization plans used in different parallel patterns. (iii) DepDSVL: Deployment details specification language. This includes computing platform, node specification, and groupings of nodes (in terms of communications). Currently, we capture basic information of the deployment platform and nodes such as number of nodes, grouping of nodes, how many cores per node, number of GPUs per node, etc. In this section, we discuss the parallel and deployment constructs. The sequential program specification is the same as in most of scientific workflows (further details are in [15]). Figure 1 shows an example sequential program model. Figure 2 shows the key concepts of our parallel program designer and key relationships. Below, we discuss these constructs, functionality and attributes.

A. Parallel Constructs

These constructs enable developers and scientists to specify how they plan to transform their sequential program into a parallel version. We focus on visually modeling possible task and data decompositions to achieve intended parallelization regardless of which parallel patterns or parallel programming models required to achieve such parallelization. Our constructs for parallelism are data decomposition (split and join), or task decomposition (parallel section, loop unrolling, parallel task).

- **Data Decomposition:** The first step in data decomposition is to think how input data could be split into smaller chunks where such smaller chunks can be processed faster and in parallel (divide-and-conquer). This implies that at some point we have to merge the outcomes of parallel tasks into one data structure.

Split construct divides an input data item into slices/chunks. The way this is implemented depends on the data structure being used. The user specifies whether a data item is geometrically divisible such as a 1D array, 2D array, image, cubes, hashtables, etc., or recursively divisible such as graphs or trees. For the first type, the user specifies what dimensions to use in splitting the data object – e.g. if we have a matrix, we can split in one dimension e.g. rows or cols, or we could split in two dimension – i.e.

rows and cols (sub-matrices), and so on. Then, the user specifies the size of the slice: could be decided based on the number of processing nodes available (usually called BLOCK policy) or could be repeated slicing (CYCLIC policy) according to a selected slice size (every n elements form one slice). On the other hand, recursive data structures can be split using a link pointer (that points to a next list entry) and the number of computing entities – e.g. in a linked list, we may divide every consecutive N elements into a chunk or according to specific attribute value which is more expensive in terms of computations. The split operation usually does not exist separately. The outcome of the split operation depends on the next node – e.g. it depends on the parallelization technology used in next program node to decide how and where the new slices can be communicated to such nodes. The next node is a parallel section.

Join construct is the inverse of the split operation. It is usually preceded by a task parallelization construct (parallel section, loop unroll, or parallel task) that produces multiple slices of the intended outcome. The objective of the Join operation is to merge these pieces/results (generated by different threads or work items back to the output data structure. If the target data structure is a single-value variable, then the user should state a reduction operation to apply – e.g. if each node calculates the sum of an array chunk, then the Join will compute the sum of sums. In the other case (merge), users specify the target output variable.

- **Task Decomposition:** Program tasks can be parallelized either by decomposing a given task into multiple concurrent subtasks or applying the same task on small chunks of the input data, or both. The latter case usually involves breaking down (unrolling) task loops into subtasks where each subtask does less iterations.

Parallel Section construct helps in grouping a set of operations in one code block (visually, a container) that we want to consider as one unit for parallelization. Scientists can select, according to nature of the available resources, the parallel model to be used in realizing a parallel section – e.g. using multi-core (OpenMP), using GPUs (OpenCL), or distributed nodes (MPI). Data items passed to a parallel section should be broadcasted to the target processing elements on which the parallel section runs. Copying data to/from a parallel section has different scenarios that we discuss below.

In a multi-core model: the parallel section is translated into a code block. The data items declared inside a parallel section or passed directly from outside nodes to an enclosed entity (there is an edge coming from a task to a task enclosed inside the parallel section) are considered as *private* to threads (a parameter in OpenMP directives), whereas parallel section passed in parameters (edges go directly to the parallel section) are considered as *shared* between all threads.

In clustered compute nodes: the parallel section is translated into a code block and the passed in data are distributed (if data are the outcome of a split operation), or broadcasted from the master node to slaves. The parallel section output data are copied from slave nodes back to the master node. This usually followed by a join operation to merge/reduce received data.

We can nest multiple parallel sections, each reflecting a parallelization level and different nested technologies may be used when realizing contained tasks. This is helpful when dealing with heterogeneous computing that requires both distributed (e.g. MPI) and shared memory models (e.g. OpenCL).

Loop Unrolling construct helps in realizing loop parallelism patterns that focuses on unrolling program loop iterations for execution by separate threads (in the case of multiple-cores), work items (in the case of GPUs), slave-based iterations (distributed nodes). Loop unrolling usually requires modifying loop header to run for fewer iterations.

Parallel Task construct is used in two cases: modeling tasks that are already parallelized and do not need to be revised by our code generator – e.g. readymade libraries, user defined tasks; and with tasks to be executed as they are in parallel - i.e. we just need to convert it into e.g. GPU kernel code and use it as it is. If the task needs to be revisited for parallelization, the user replaces it with a parallel section and flush task details and required parallelization.

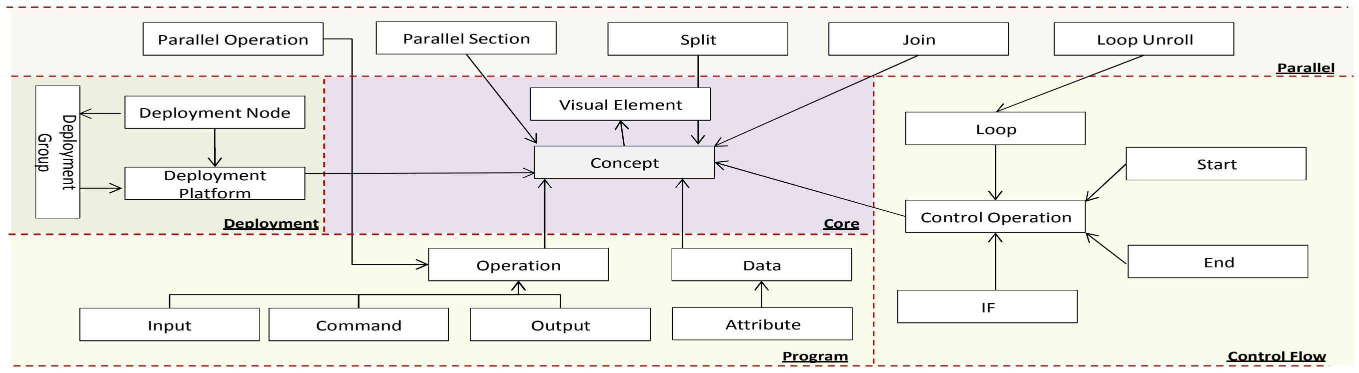


Figure 2. Parallel program designer constructs and their key groups and relationships

B. *Deployment Constructs*

These help developers and scientists to model a program's underlying compute platform. Currently, we are mainly interested in capturing how many nodes to be used in the deployment model, specifications of each node (memory, CPU, storage) and number of accelerators plugged in each node. Each node has a group name attribute that is used when grouping nodes into sub-clusters. This is helpful when mapping specific tasks for execution on a group of nodes – e.g. in the MapReduce model we have Map nodes and Reduce nodes. As a further extension of this we plan to include inter-node communication speed and bandwidth, and accelerator (e.g. GPUs) specification details. This helps in generating efficient code based on nodes capacity and tradeoffs between parallelization and communication overhead.

C. *Model Refinement*

A key problem with existing parallel programming models is the lack of a common model that can work with different, heterogeneous computing platforms. We have found many efforts proposing new common models that try to replace two or more existing parallel models with one platform [16]. In this section, we show that most of the parallel scenarios supported by these parallel programming models can be modeled using our approach.

Message-Passing Interface - (MPI): A set of APIs that facilitate communication between different nodes in a distributed memory based supercomputers or clusters. MPI has APIs for communication (send/receive), synchronization between different nodes (Barriers), combining results from different nodes (gather, reduce), and sharing public information about the cluster (number of processes, current process id). The Split (realized as a loop over number of nodes while calling send API to send data to all slave nodes and another piece of code in the slave node to receive the data slice to work on), Join (realized as send by slave node and loop in the master node to receive data), and program/data flow edges cover the communication (code to run on the master node and code in slave nodes (parallel section)) and reduction operations. The deployment group covers the possible communicators (communication groups). The parallel section helps in consolidating code blocks run in parallel on different nodes. Elements outside the parallel sections represent data and tasks to be executed on the master node. The deployment nodes define the number of nodes to run the MPI program on – i.e. in the MpiRun command params.

OpenMP: A set of compiler directives and APIs that help in parallelizing programs using multi-threading to utilize multi-core shared-memory architectures. OpenMP is based on the fork-join parallel pattern. Parallel Section can be used to group all instructions that we need to execute in parallel on multi-cores processors (using OpenMP). OpenMP usually has to specify the shared data between all threads and the private data for each thread. When multicore mode is selected: any data item declared inside a Parallel Section is declared as a private memory, while data sent to the Parallel Section are considered shared data between threads. This is the same with Parallel Section outgoing edges. Finally, the reduction of data generated by all threads is done using the Join construct. Necessary platform information is delivered as parallel section attributes – e.g. number of threads, current thread id, etc. Our Loop Unrolling construct captures the OpenMP parallel loop directive.

Open Computing Language (OpenCL): A platform to help in heterogeneous computing using CPUs and/or GPUs. OpenCL is based on the Single Program Multiple Data (SPMD) computing model. It provides a set of APIs to help in creating kernels (programs) to be executed by GPU work items (threads). It also supports copying memory between host and accelerator device global memory. Another set of APIs provide current work item Id relative to local work group and global work items. To convert any program into the SPMD model, we have two options: if the tasks will be distributed, then we can leave the task unmodified and pass in a chunk of the data to each parallel thread or process. In the shared memory model we have to change the task to have each instance work on a portion of the *shared* data. Loop unrolling is usually a good candidate source of SPMD parallelism. In this case, we may consider each iteration as a separate thread (work item), so we replace the for loop header with a statement (loop variable = global thread id). Otherwise, we extend the loop header elements (initialization, condition, and update) with a dummy variable that counts how many iterations per thread – e.g. for (initial, *unroll_var = 0*; condition *&& unroll_var < number_of_iterations* ; update , *unroll_var++*). The Parallel section can be used to enclose elements that constitute a new kernel. All arrows directed into the parallel section are considered as kernel parameters and copied from host memory to device global memory. All arrows directed towards one of the internal operations in the parallel section are considered for further local memory copy. Parallel section also delivers global information such as get_global_id and get_local_id, etc. The user needs to specify the number of blocks and number of threads per block. These are parameters in the parallel construct.

Combined MPI, OpenMP, and OpenCL: When targeting heterogeneous platforms we can use nested parallel sections. The outer section could be used for e.g. distributed nodes (MPI), while the inner sections for multi-cores (OpenMP) or GPUs (OpenCL).

III. CASE STUDIES

To evaluate the effectiveness of our approach in parallelizing programs and improving developers' capabilities in handling parallel programs, we have conducted a set of case studies with several scientists. We summarize three of them here: matrix multiplication, N-Body simulation and digital signal processing.

A. *Parallel Matrix Multiplication*

In this section, we show how our approach can help in parallelizing the sequential matrix multiplication program from Figure 1. An initial thought to parallelize this program could be to split one of the matrices while keeping the other matrix as it is, or splitting both A and B. The later will have an impact on the calculations (sum of matrix A rows multiplied by matrix B cols). In either case, we need to take into consideration the deployment details: are we going to run this on a cluster or on a single node? Also we may have other computing devices – e.g. GPUs - that we might use in completing subtasks assigned to each node.

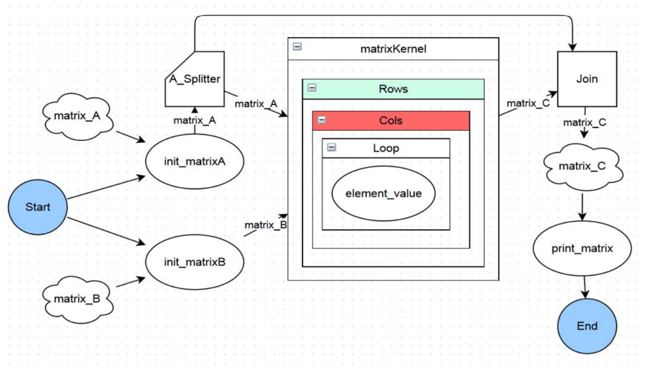


Figure 3. Example parallel matrix multiplication model

```
...
int taskid, ntasks;
MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
MPI_Comm_size(MPI_COMM_WORLD, &ntasks);
...
//@Splitter: A_Split
 int offset0 = 0;
 int aSplitbulkSize = 1*aRows aCols/ ntasks ;
if( taskid == 0 ) {
for (int dest = 1 ; dest <= ntasks ; dest++) {
ierr = MPI_Send( A[offset0], aSplitbulkSize , MPI_DOUBLE, dest
                , 0 , MPI_COMM_WORLD);
offset0 = offset0 + bulkSize;
...
else if( taskid != 0 ){
                ierr  = MPI_Recv( A,  aSplitbulkSize, MPI_DOUBLE, 0 ,
                          0 , MPI_COMM_WORLD , &status); ...
//@Parallel Section: MatrixKernel
if( taskid != 0) { // Calculate C[i][j] }
//@Join: Join
if(taskid !=0) {
    ierr  =  MPI_Send(&C[offset0],  aSplitbulkSize,  MPI_DOUBLE,  0  ,  0  ,
MPI_COMM_WORLD); ... }
else if( taskid == 0 ) {
    for (int i=1; i<=ntasks; i++) {
        ierr = MPI_Recv(C[(i -1) * aSplitbulkSize], i * aSplitbulkSize, MPI_DOUBLE, i ,
0 , MPI_COMM_WORLD , &status); } ...
```

Figure 4. A snippet of generated parallel matrix multiplication code modeled in Figure 3

Figure 3 shows a scenario where we decided to split matrix A and distribute the slices to different cluster nodes and broadcast matrix B to all nodes. This should be done on the master node (tasks outside the parallel section "matrixkernel"). The parallel section defines that the enclosed tasks (loops & calculation) are to be executed in parallel on slave/worker nodes. The outcome of the matrix multiplication by each node is then sent back to the master node where we have a Join operation to merge results together into matrix_C. We have configured the *splitter* construct properties to split matrix A using *block_policy* according to number of nodes and then distribute each chunk to slave nodes. Another splitter could be added if we want to split matrix B as well, but will impact the calculations operation. Figure 4 shows a snippet of the generated parallel code for matrix multiplication. We show parts of the generated code reflecting the split, join and parallel section realization. The rest should be the same as in the sequential version. Using split and join with a cluster computing model is realized using the MPI master-slave pattern (one of the well-known parallel design patterns).

```
//@Parallel Section: GPUKernel
cl_program program;
cl_kernel kernel;
const char *KernelSource = "\
__kernel void ParallelSection( __global double* A, __global double* B) {\
int i = get_global_id (0); {\
for(j = 0;j < aCols ; j++ ) {\
calculate_matrix_element();}\
} }";
program = clCreateProgramWithSource(context, 1, (const char **) &
KernelSource, NULL, &err);
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
kernel = clCreateKernel(program, "ParallelSection", &err);
cl_mem memobjA = clCreateBuffer(context, CL_MEM_READ_WRITE, *, NULL,
&ierr);
ierr = clEnqueueWriteBuffer(command_queue, memobjA, CL_TRUE, 0, *, A, 0,
NULL, NULL); ...
```

Figure 5. A code snippet generated for GPU kernel

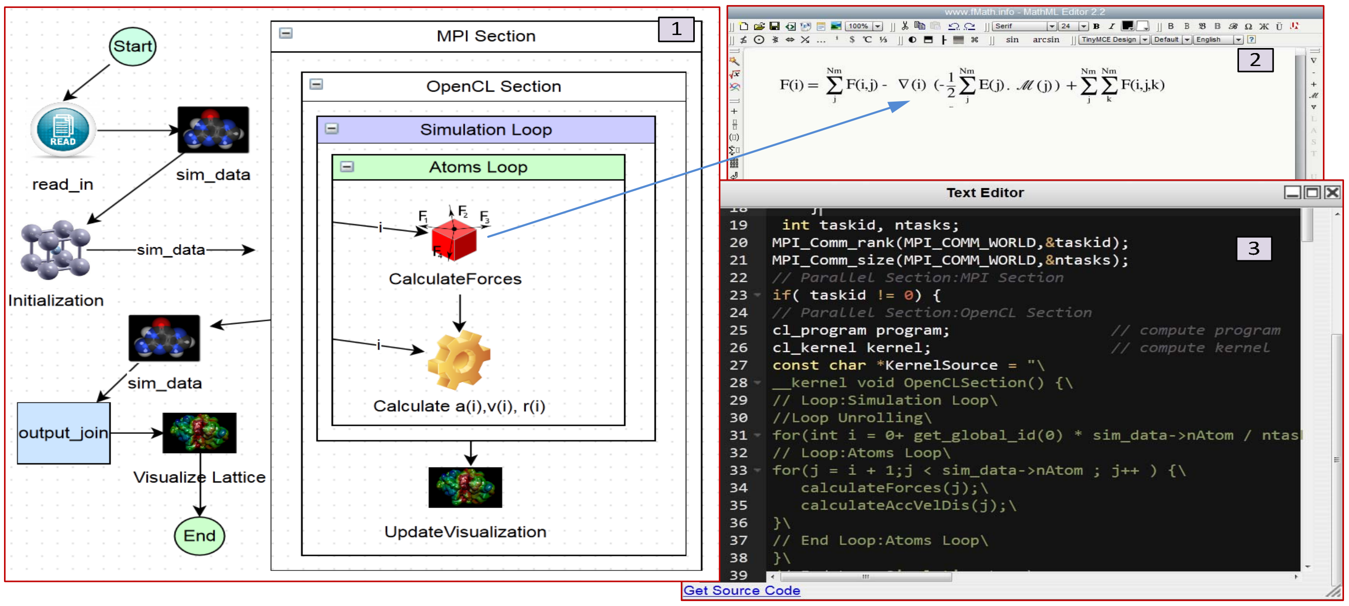
Now, assume that each computing node has a GPU device so that we can use it to accelerate the computations assigned to nodes. In this case we need to add another parallel section inside "matrixkernel" and select "Run on GPU" property to encapsulate the computational tasks into *kernel* to be computed by GPU work groups and work items. The number of work items, work groups, and the dimensions of work items in work groups depend on the size and dimensions of the data being processed. Moreover, the data to be processed by the GPU device will be copied from the host node memory to GPU global memory. The same will be done after work items finish.

Table 1. Performance results (in sec) of the generated MPI code with different matrix size and number of nodes

Matrix Size	**1 node**	**3 nodes**	**5 nodes**	**7 nodes**
100 X 100	0.04	0.01	0.0017	0.0014
300 X 300	0.16	0.04	0.025	0.017
500 X 500	1	0.19	0.11	0.06

Table 2. Performance results (in sec) of the generated MPI+OpenMP code with different matrix size, nodes, cores

Matrix Size	**1 node**	**3 nodes**	**5 nodes**	**7 nodes**
100 X 100	0.04	0.006	0.008	0.005
300 X 300	0.16	0.03	0.016	0.01
500 X 500	1	0.14	0.09	0.04



$$F(i) = \sum_{j}^{Nm} F(i,j) - \nabla(i)\ (-\frac{1}{2}\sum_{j}^{Nm} E(j).\ \mathscr{M}(j)) + \sum_{j}^{Nm}\sum_{k}^{Nm} F(i,j,k)$$

```
 int taskid, ntasks;
MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
MPI_Comm_size(MPI_COMM_WORLD,&ntasks);
// Parallel Section:MPI Section
if( taskid != 0) {
// Parallel Section:OpenCL Section
cl_program program;                      // compute program
cl_kernel kernel;                        // compute kernel
const char *KernelSource = "\
__kernel void OpenCLSection() {\
// Loop:Simulation Loop\
//Loop Unrolling\
for(int i = 0+ get_global_id(0) * sim_data->nAtom / ntas
// Loop:Atoms Loop\
for(j = i + 1;j < sim_data->nAtom ; j++ ) {\
   calculateForces(j);\
   calculateAccVelDis(j);\
}\
// End Loop:Atoms Loop\
}\
```

Figure 6. N-Body simulation models in our web-based IDE tool, Horus HPC

Tables 1, 2 shows the performance improvement (in secs) of the generated code using MPI and MPI+OpenMP against the sequential version of the matrix multiplication program. The time reported excludes matrices initialization tasks, because it is common in all scenarios. It is worth mentioning that the performance of the generated code depends heavily on the user-modelled transformation. We plan to build new plugins that can generate recommendations regarding the parallelization plan and number of nodes, blocks and threads to use.

B. *N-Body Case Study*

In this case study we asked a new PhD student who is working on molecular simulation to use our tool to transform her sequential program into MPI & OpenCL version. She did not have any previous experience in parallel programing. We conducted a 1-hour training to explain how the approach works. *First*, we asked her to model her sequential program using our sequential DSVL developed earlier (introduced in [15]). Then, we asked her to use the parallel construct to model how she wanted to parallelize her program. Figure 6-1 shows a snapshot of the parallel model. Figure 6-3 shows a part of the parallel program, using MPI and OpenCL, generated by our tool based on parallel model Figure 6-1. This took around four hours. On the other hand, we asked a HPC programmer to parallelize the same N-Body sequential program using, which was written by someone else, using MPI and OpenCL. The programmer reported that the initial working parallel version that is the same as our parallel model took around three weeks full-time. It took the same time to get an optimised version.

C. *DSPSR Case Study*

The third case study is reengineering the detection module in an existing digital signal processing for pulsar astronomy tool developed internally at Swinburne by astrophysics team (http://dspsr.sourceforge.net/). The core code of the detection module, a part of it is shown in Figure 7, has mainly two nested loops: one on input signal channels (nchan=512) and the second nested loop is on input signal periods (ndat=1024) for every pairs of signals. The intended solution was to unroll both loops on 8 nodes (each work on 512/8=64 iterations). Each node has a GPU (should work on the inner loop, 1024 iterations). The solution was using two nested parallel section constructs: one for node distribution (MPI), and the other for GPU processing. The key problem we faced in this parallelization task was the dependency between iterations to advance array pointers. The scientist made a simple modification to link array indexes to loop variables instead of using pointers. Then we unrolled the inner loop in an OpenCL GPU kernel.

```
for (unsigned ichan=0; ichan<nchan; ichan++) {
   const float* p = input_base + ndat*ndim*npol*ichan;
   const float* q = p + ndat*ndim;
   r[0] = output_base + ndat*ndim*npol*ichan;
   r[1] = r[0] + 1;…
   for (j=0; j<ndat; j++)  {
      p_r = *p; p++;
      p_i = *p; p++;
      pp = p_r * p_r + p_i * p_i;
      qq = q_r * q_r + q_i * q_i;

      *S0 = pp + qq;  S0 += span;
      *S1 = pp - qq;  S1 += span;
      …
```

Figure 7. exerpt from the detection module to be reengineered

IV. Implementation

Figure 8 shows a high-level architecture of our toolset, Horus HPC, with its two main components: the model designers and code generators. Our parallel program designer is implemented using our Horus framework as a web application based on HTML5 and JavaScript [15]. It is built on an existing open source web modeling tool (https://www.draw.io/). The web-based platform allows developers and scientists to use, collaborate, and share models and solutions. We extended this tool with a DSVL designer to help in developing domain-specific visual languages (DSVLs). This helps scientists and developers in creating operations and data structures that are common to their domains without a need to redefine such operations or data structure when modeling new problems. Once a DSVL is defined, users can register it to be used from within the tool itself.

The parallel program designer (Figure 8-1) is implemented as a DSVL with all constructs reflected in the meta-model discussed above. The code generator (Figure 8-2) is implemented in JavaScript to complement the parallel DSVLs provided through our web-modeling tool. It simply parses the graph nodes (vertices and edges). For simple nodes in the program (those with no parallel impact, e.g. data, if, or task nodes), the core code generator outputs the corresponding realization code, which is currently in C language. For parallel operations – e.g. split, join, and parallel sections, etc. the code generator consults the corresponding parallel technology code generator – e.g. in a split node, if the distributed flag is set, then the MPI code generator is triggered, a sample MPI code generator is shown in Figure 9. This code snippet shows parts of the MPI initialization code, and split construct code generator. Otherwise, if the shared flag is set in a parallel section, the OpenCL generator is triggered. The resultant parallel program is then compiled and deployed on the target computing platform (Figure 8-3). To support reverse (and round-trip) engineering we use code annotations (The same annotation used in the generated code in Figure 4 and Figure 5): where scientists can add annotations in code to guide the model reconstruction process.

Our code generation component takes program models, as a set of nodes and edges developed with the parallel program designer discussed above, as input and generates a corresponding program with the necessary parallelization aspects as defined by users. The code generator has a sequential code generator as its core. This core code generator is responsible for generating a complete sequential version of the user specified model (without any parallelization). This is helpful when the user would like to build his initial program from the existing (predefined, or other users' defined) building blocks. For parallelism, we have separate generators for different parallel programming model – i.e. we have MPI, OpenMP, OpenACC, and OpenCL code generators. This permits further improvements of such code generators separately without modifying the whole code generation component.

The core code generator will issue calls to those specialized code generators whenever it finds a parallel construct used in the input model. According to the parallel model used in these constructs, the core code generator calls the corresponding generator – e.g. if a parallel construct is used with distributed model, the MPI code generator is called. The outcome of this code generation is weaved within the sequential program as needed. This approach improves extensibility in order to support further possible programming models and extension of individual code generators. Each code generator processes each construct in the designer meta-model taking into consideration that the meta-model may be extended in future (more details in the implementation section).

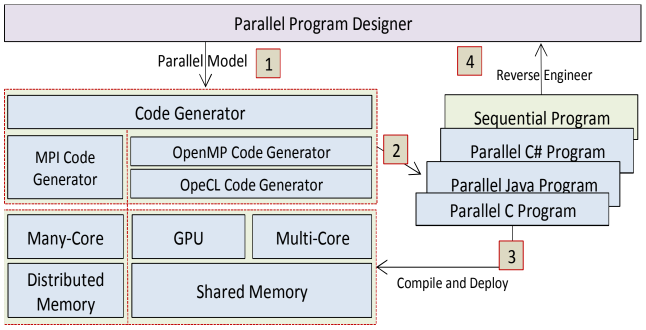


Figure 8. A high-level architecture of our approach

```
…
MPICodeGenerator.prototype.Start = function(currentNode, graph, output, flags) {
…
 output.append(" int taskid, ntasks;");
output.append("MPI_Comm_rank(MPI_COMM_WORLD,&taskid);");
output.append("MPI_Comm_size(MPI_COMM_WORLD,&ntasks);");
}
MPICodeGenerator.prototype.Split =  function(currentNode, graph, output, flags) {
 …
  output.append(" int bulkSize = " + copySize + ";");
  output.append("if( taskid==0 ) {\n");
  output.append("\tfor (int dest = 1 ; dest <= ntasks ; dest++) { \n");
  var line = 'ierr = MPI_Send( ' + propertyName + copyIndex + ', bulkSize , '
                                  + dataType + ', dest , 0 , MPI_COMM_WORLD); ';
       …
```
Figure 9. Code snippet of the MPI Code Generator

First, the code generator checks the specified deployment details: Is it a single node? a cluster? Do nodes have accelerators(GPUs)? Do we have multi-core CPUs? The answers to these questions are used in initializing the program runtime environment including MPI and/or OpenCL initialization as described in the fourth step. *Second*, we generate declarations for all data nodes to avoid "undeclared identifier" compilation errors that may arise when generating tasks that depend on data not declared in the program yet. This usually happens when a node has multiple edges and the edge order does not reflect the real dependency of the next nodes. *Third*, the core code generator builds a dependency list of the model nodes. This list is used to decide the order of graph nodes and edges to traverse for code generation. The code generation starts from the graph *start* node and keeps generating code for the rest of the graph elements until all nodes have been *realized*. *Fourth*, the code generator issues a call to technology-specific code generator according to the node currently being processed and technology reflected on the node, e.g. if the node is a start node, necessary code generators are called to perform necessary initialization tasks. If the node is a split with distributed memory model, then the MPI code generator is called. The same applies on Join, Parallel section, and Loop graph nodes. Each function in the code generator has a predefined signature as follows (currentNode, programGraph, outputCode, flags). Figure **9** shows a code snippet from MPI code generator.

V. Threats to Validity

The number of experiments we have conducted so far is limited with regard to usability, expressiveness, performance and extensibility aspects. We plan to conduct more detailed performance and usability evaluations.

Our code generation approach has some limitations, which we are currently working to address: (i) Memory Management: currently, we do not address optimized memory access when generating kernel code (memory coalescing techniques). (ii) Support for custom data type data communicated between nodes when using MPI. However, this should be easy to support. The same with OpenCL data types such as float2, float4, etc. Currently, this must be specified by developers. (iii) Recursive functions. This is a limitation of the underlying technology – e.g. CUDA does not support recursive kernels. (iv) Data dependency between parallelized iterations is not automatically resolved. It is still the developer/scientist responsibility to handle data dependencies. (v) We do not make use of the parallel programming platform special libraries such as CUDA cuFFT and cuBLAS-XT. This is a design limitation of our current approach implementation. We do not modify the core code of the sequential program. We do extend such code with parallel constructs. Users can add *parallel task* model element that refers to CUDA libraries. We plan to extend our parallel code generators with a collection of libraries and when to use them. Thus, the code generator can look for matches to replace with parallel library APIs. (vi) Deep modifications of the enclosed tasks to use threadId or work item Id is not supported. The readability of the generated code could facilitate fine-tuning activities.

VI. Summary

We described a novel model-driven approach to help in transforming sequential programs to parallel versions through visualizing parallelization aspects and patterns. Our approach enables utilizing the underlying computing platforms without deep experience in parallel programming models. We capture sequential program details including data structures and tasks. Scientists and developers then extend this model with parallelization specifications and specify the deployment details of the program. These three aspects are used in generating a parallel version of the input sequential program. This approach facilitates updating any or all of these three aspects without modifying the other aspects in the model. Code generation supports different possible deployment models and programming models including distributed nodes using MPI, multi-cores using OpenMP, and GPU accelerator devices using the OpenCL and OpenACC programming model. We have applied our approach to several problem domains. Due to space limitations only three examples discussed in this paper including matrix multiplication, N-Body simulation and Digital Signal Processing.

VII. Acknowledgment

This research is supported by the Australian Research Council under Discovery Project DP120102653.

References

[1] S. Johnston, M. Bailes, N. Bartel, C. Baugh, M. Bietenholz, C. Blake*, et al.*, "Science with the Australian square kilometre array pathfinder," *Publications of the Astronomical Society of Australia,* vol. 24, pp. 174-188, 2007.

[2] T. G. Mattson, B. A. Sanders, and B. L. Massingill, *Patterns for parallel programming*: Pearson Education, 2004.

[3] M. W. Hall, J. M. Anderson, S. P. Amarasinghe*, et al.*, "Maximizing Multiprocessor Performance with the SUIF Compiler," *Computer,* vol. 29, pp. 84-89, 1996.

[4] A. Kravets, A. Monakov, and A. Belevantsev, "GRAPHITE-OpenCL: Generate OpenCL Code from Parallel Loops," presented at the GCC Summit 2010, 2010.

[5] W. Blume, R. Eigenmann, J. Hoeflinger, D. Padua, P. Petersen, L. Rauchwerger*, et al.*, "Automatic Detection of Parallelism: A grand challenge for high performance computing," *IEEE Parallel & Distributed Technology: Systems & Applications,* vol.2, p. 37, 1994.

[6] A. K. Sujeeth, T. Rompf, K. J. Brown, H. Lee, H. Chafi, V. Popic, *et al.*, "Composition and Reuse with Compiled Domain-Specific Languages," in Proc. European Conference on Object-Oriented Programming, Montpellier, France, 2013.

[7] J. Ragan-Kelley, C. Barnes, *et al.*, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in Proc. 34th ACM Conf. on Programming language design and implementation, Seattle, USA, 2013.

[8] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos*, et al.*, "Liszt: a domain specific language for building portable mesh-based PDE solvers," presented at the Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, Seattle, Washington, 2011.

[9] A. K. Sujeeth, H. Lee, K. J. Brown, T. Rompf, H. Chafi, M. Wu*, et al.*, "OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning," presented at the Proceedings of the 28th International Conference on Machine Learning, 2011.

[10] F. Jacob, R. Arora, P. Bangalore, M. Mernik, and J. Gray, "Raising the level of abstraction of GPU-programming," in *Proc. 16th Int. Conf. on Parallel and Distributed Processing,* pp. 339-345, 2010.

[11] T. D. Han and T. S. Abdelrahman, "*hi*CUDA: a high-level directive-based language for GPU programming," in Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, Washington, D.C., 2009.

[12] D. Dig, J. Marrero, and M. D. Ernst, "Refactoring sequential Java code for concurrency via concurrent libraries," in Proceedings of the 31st International Conference on Software Engineering, 2009.

[13] M. Palyart, D. Lugato, et al, "HPCML: A Modeling Language Dedicated to High-Performance Scientific Computing," in Proc. of 1st Int. Workshop on Model-Driven Engineering for High Performance and CLoud computing, Innsbruck, Austria 2012.

[14] F. Jacob, D. Whittaker, S. Thapaliya, P. Bangalore, M. Mernik, and J. Gray, "CUDACL: A tool for CUDA and OpenCL programmers," in Proceedings of *2010 International Conference on High Performance Computing (HiPC)*, 2010, pp. 1-11.

[15] M. Almorsy, J. Grundy, et al. , "A Suite of Domain-Specific Visual Languages For Scientific Software Application Modelling," in the proceedings of 2013 IEEE Symposium on Visual Languages and Human-Centric Computing, San Jose, CA, USA, 2013.

[16] J. Diaz, C. Munoz-Caro, and A. Nino, "A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era," *IEEE Transactions on Parallel and Distributed Systems,* vol. 23, pp. 1369-1386, 2012.