PReMAS: Simulator for resource management

David P. Bunde
Department of Computer Science
Knox College
Galesburg, IL USA
Email: dbunde@knox.edu

Vitus J. Leung
Analytics Department
Sandia National Laboratories
Albuquerque, NM USA
Email: vjleung@sandia.gov

***Abstract*—This paper reports on PReMAS, a simulator used for research on parallel scheduling, processor allocation, and task mapping for systems with rigid batch jobs scheduled with pure space sharing. PReMAS features an object-oriented design that has proven adaptible for a variety of research foci and is easily extendible to incorporate new resource management algorithms. We are making its code public to save others development time and to promote the reproducibility of resource management research. We discuss PReMAS's design and lessons learned in its development to promote greater discussion of the simulators underlying so much resource management research.**

I. Introduction

Simulation is a key technique for research in resource management. It allows researchers to develop algorithms for systems which are either unavailable or not yet created. It also gives researchers the ability to control all aspects of an experimental system and allows detailed "instrumentation" of the system to record its behavior at any desired level of detail and without impacting the system itself.

Roughly speaking, a simulation-based research project in resource management involves two distinct tasks. First, the simulator is developed and results are collected. This requires writing complicated code for a simulator that implements one or more algorithms of interest, followed by running the simulator on appropriate data sets. For the second task, the researcher distills information about the simulator and algorithms into a compact form that can be presented along with the results in a technical paper from which others will learn general lessons.

While this research model works well in most cases, it has two significant disadvantages. The first is that the distilled description of the simulator and algorithm(s) may not be sufficient for other groups to reproduce the results. In practice, the code for any significant program like a simulator includes many details and it is difficult to judge which must be explained for other developers to reproduce its behavior. When writing, there is also tension between the goals of presenting the main ideas of a research project and including implementation details. Paper page limits further complicate authors' efforts to describe their work. All of these factors mean that some technical papers fail to present their work in a fully reproducible way. Without direct access to the simulator itself, the reader's only recourse in these cases is to seek clarification from the author, a less than ideal solution that either leaves questions unanswered or creates work for the author, who may well have moved on to another project.

The other disadvantage of the current research model is that it forces each research group to create an entirely new simulator. While some differences between simulators arise from the different questions being asked by each group, the duplicate effort to reimplement shared features seems tragic given the perpetual time shortage experienced by many researchers.

To promote the reproducibility of our results and also to save time for other researchers, we have decided to release the code for our simulator, which we have named the Parallel Resource Management Algorithm Simulator (PReMAS). It is available at http://faculty.knox.edu/dbunde/PReMAS under a BSD-style license to maximize potential reusability. In addition, this paper contains a discussion of its design and the lessons we learned developing it. Our hope is that our actions and this paper will encourage others to similarly release their code or at least join a more public discussion of simulation design.

PReMAS has been developed to answer a series of research questions in High-Performance Computing (HPC), particularly targeting high-end systems managed by the US Department of Energy. The first systems simulated were part of the CPlant project [35]; a modern successor system is Cielo [28]. The simulator has been used for research on scheduling ([37], [25], [31]), processor allocation ([22], [5], [38], [19]), and task mapping ([6], [23], [3]). The scheduling and processor allocation functionality has been incorporated into the Structural Simulation Toolkit (SST) [33], a simulation framework for the design, evaluation, and optimization of HPC architectures and applications; [32] describes our contributions.

PReMAS performs a relatively high-level simulation. It reports the quality of a processor allocation or task mapping in terms of metrics such as the average pairwise distance between allocated nodes or communicating tasks. These metrics have been shown to correlate with job execution time (e.g. [22], [11]), but the higher-level simulation requires fewer assumptions about the jobs than attempting to directly model the effect of allocation or mapping on job execution time. This is a different approach from the SMPI [8] component of SimGrid [36], which quickly runs a modified version of the actual application and thus requires access to its code.

Central to our ability to use PReMAS on a variety of problems has been its object-oriented design. Scheduling, processor allocation, and task mapping are each performed by different components with interfaces that enforce a separation of roles. This allows new implementations of a component to be swapped in, allowing research on the components separately. This design has also allowed much of the simulator's use and development to be performed by undergraduate research assistants, who can begin focusing on only a single component.

The contributions of this work include the following:

- The code itself.
- Lessons learned trying to design a flexible, extensible simulator.
- Several cases where refactoring simulator code led to a deeper understanding of the (scheduling and processor allocation) algorithms involved.
- Discussion of issues that threaten the reproducibility of simulation results.
- Our strategy for incorporating undergraduates in resource management research using PReMAS.

Our hope is that this paper will encourage others to share their simulators or at least discuss their features in more detail. We believe that the resource management research community would benefit from having a common code base or at least having simulator code readily available to clarify the contents of research publications.

The remainder of this paper is organized as follows. Section II defines the resource management problems we have examined with PReMAS. Section III summarizes the simulator design. Sections IV and V describe the lessons learned while implementing it. Section VI outlines our process for using it in undergraduate research at Knox College. Section VII summarizes related work. Section VIII discusses future plans.

II. Problem definitions

Because definitions can vary within the resource management research community, we now define our setting and the terms relevant to the simulator. Our definitions reflect our focus on systems like those run at Sandia National Labs for the US Department of Energy, but the assumptions we make could be relaxed by other adopters of the simulator.

A. Properties of jobs

A *job* is a program being run, plus the relevant data, which are submitted together without any expectation of interactivity (batch jobs). The simulated HPC system has many nodes, each of which will be assigned to at most one job at a time (pure space sharing). In particular, all the cores of a given node will be assigned to the same job. The compute nodes are diskless and all data is stored separately so that jobs are agnostic about the nodes they run on. The number of nodes needed by a job is declared upon job submission and does not change during execution (rigid jobs). Our typical assumption is that, once started, jobs cannot be interrupted and must run to completion (non-preemptive jobs), but we have incorporated one scheduler that assumes jobs can be stopped and restarted. We assume the system has identical nodes, forbidding nodes with differing configurations.

When users submit a job to the system, they specify the number of nodes it needs and provide an *estimated running time*. This time is an upper bound on the job's duration for use when deciding when to run it. The system will kill a job exceeding its estimated running time to prevent run-away jobs. This behavior motivates users to provide conservative estimates (i.e. greater than the job's *actual running time*), meaning that most jobs "finish early" from the system's perspective.

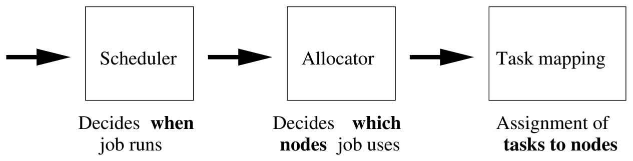
Scheduler
Allocator
Task mapping
Decides **when** job runs
Decides **which nodes** job uses
Assignment of **tasks to nodes**

Fig. 1. Resource management steps between job submission and execution.

Most of our simulations use traces from the Parallel Workloads Archive [13], which provides scheduling information on sequences of jobs run on systems at a variety of HPC centers. For PReMAS, we use a script to remove unused information from a trace, leaving just each job's *arrival time* (when it was submitted to the system), number of nodes used, actual running time, and (optionally) estimated running time. When other information is needed (e.g. a job's desired shape), this is added in a subsequent step.

B. Resource management steps

Jobs submitted to the simulated HPC system go through the series of resource management steps shown in Figure 1. In principle, these steps could be combined, but they are separated for practical reasons.

The first step is *scheduling*, deciding when each job should run. The typical assumption for PReMAS is that jobs, once started, always run until they complete or they reach their estimated running time. With this assumption, a scheduling decision (which job(s) to start, if any) is needed when a new job arrives and when one completes. The simulator also implements one algorithm (timed-run [37]) that will stop and restart jobs from the beginning, which also requires periodic scheduling decisions (whether to stop a currently-running job).

The second step for each job is *processor allocation*, the selection of particular free nodes on which to run a specific job that has been scheduled. There are two kinds of allocations:

- *Contiguous allocation*, in which each job is guaranteed to be allocated to a compact shape, and
- *Non-contiguous allocation*, in which arbitrary sets of nodes can be assigned to a particular job.

The kind of allocation used is system-dependent; contiguous allocation eliminates inter-job contention, but its fragmentation can delay jobs even when enough nodes are available.

For research on non-contiguous allocation, it is important to note that PReMAS does not model changes in running time caused by allocation quality. Instead, it reports the quality of an allocation as the sum of pairwise L_1 distances between allocated nodes; this is the number of hops made by the job's messages, which correlates with job running time (e.g. [22]). This is a conservative measure of allocator performance since improvements that shorten job running times will reduce system utilization, potentially allowing further improvements in allocation quality, but this is a point that needs to be carefully explained when describing results obtained with PReMAS since the reader may otherwise expect results in terms of running time directly.

The third and final step for each job is *task mapping*, the assignment of job tasks to nodes (or cores within each

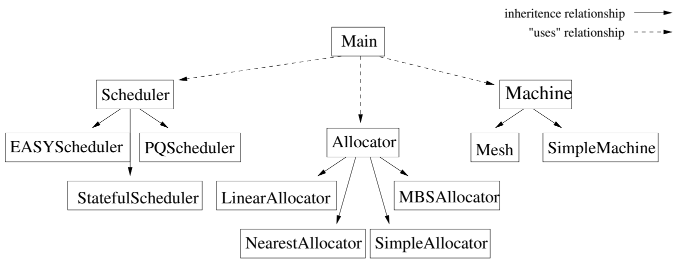

Fig. 2. Simplified class diagram for PReMAS. Diagram taken from [32].

node). For MPI programs, task mapping is each MPI rank identifying its role. Unlike scheduling and processor allocation, task mapping is typically done in the application rather than system software. It is also often done implicitly without much consideration, despite several experiments showing that it can significantly impact application running time (e.g. [18], [4]).

III. Overall design of PReMAS

Now we describe the design of our simulator. At its center is a priority queue of events ordered by time. Prototypical events are the arrival of a new job or the completion of a currently-running job. The job arrival events are added before the simulation begins. Whenever a job is started, a completion event for that job is added to the queue. The simulator's main loop simply removes and processes the next event from the priority queue. This part is standard for event-based simulators.

The special part of PReMAS that allows it to serve diverse roles is the careful way in which responsibility for decisions and system state is split between components. Each component is a Java object and we use inheritence from an abstract base class to allow each of them a variety of implementations. The main system state is stored in `Scheduler`, `Allocator`, and `Machine` objects; we return to task mapping in Section III-B below. Data and control flow each obey the ordered structure suggested in the previous section, from `Scheduler` to `Allocator` to `Machine`. Figure 2, copied from [32], gives an overview of the structure.

`Scheduler` keeps track of waiting jobs and any plans it has about when they will run. These data structures are updated when the simulator's main loop calls its `jobArrives` and `jobFinishes` methods. After these, the main loop also calls `tryToStart`, which allows the `Scheduler` to start a job. (If a job is started, `tryToStart` is called again so multiple jobs can start at the same time.) In general, `Scheduler` uses `Allocator` to decide if a job can start.

`Allocator` is responsible for deciding whether a job can be started and deciding which nodes it will be allocated. `Allocator`'s most important method is `allocate`, which takes the job in question. This method returns `null` if that job cannot be allocated. Otherwise, it returns an `AllocInfo` object as a `Machine`-specific representation of the selected allocation. Importantly, `allocate` does not change the system state so that `Scheduler` can call it multiple times to see which jobs can start. When `Scheduler` has decided on a particular job, the corresponding `AllocInfo` is returned to the main loop, which updates the system state.

`Machine` keeps track of the free nodes. Its `SimpleMachine` subclass just records the number of free nodes, treating the machine as a "bag of nodes", while the `Mesh` subclass keeps track of the identities of free nodes in the three-dimensional mesh. Other topologies could be easily added, but `Mesh` is the only one implemented thus far since it matches our target machines.

All of these parts interact with `Statistics`, a class that generates log files appropriate to the type of research being performed. It provides common headers with information about the simulation setup and ensures that the same output format is used by each subclass implementing a component. The jobs themselves are represented by `Job` objects. The `Main` class provides overall management of the simulation, including reading in the input trace, creating all the objects, and running the main simulation loop.

A. *Use cases*

The following are specific ways the simulator is used, with the relevant subclasses:

- For much scheduling research, the job allocations are ignored; non-contiguous allocation is implicitly assumed and all allocations are considered equally good. In this case, the only machine state needed is the number of free nodes. This is supported by the `SimpleMachine` subclass of `Machine`. It requires no allocation decisions other than a sanity check that enough nodes are available, functionality provided by the `SimpleAllocator` subclass of `Allocator`.
- Most of our research on processor allocation with the simulator is non-contiguous allocation on meshes. In this case, we use the `Mesh` subclass of `Machine`. Numerous schedulers are supported, but we typically use `EASYScheduler`, which implements EASY [24], an algorithm used on actual HPC systems (e.g. [2], [24]) and commonly considered in scheduling research. It allows jobs *backfill* (i.e. to start ahead of earlier-queued jobs) when doing so does not delay the job at the head of the queue.
- The situation gets much more complicated with contiguous allocation. In this case, jobs have hyper-rectangular shapes and may be unable to run even if enough nodes are available due to fragmentation. A subclass `ContiguousJob` of `Job` stores the shape information. `Scheduler` uses `Allocator` to determine whether a particular job can run. Since not all schedulers can work in this setting, contiguous allocators extend the `ContiguousAllocator` subclass of `Allocator` and runtime type checking is used to ensure selection of a valid combination of scheduling and allocation algorithms.
  Simple schedulers such as FCFS (priority-based scheduling from `PQScheduler` with job priorities from `FIFOComparator`) work for contiguous allocation without changes. We also provide a special version of EASY [24] for contiguous allocations. This is necessary because EASY gives the first queued job a guaranteed starting time; thus, EASY is not allowed to backfill without checking

with the allocator that the backfilled job will not interfere with the guaranteed job. This requires additional functionality from the allocator, represented by the `PairTestableAllocator` interface. Again, runtime type checking ensures that valid pairs of algorithms are used. The same type checking also automatically selects a version of the scheduler appropriate for contiguous allocation when one is needed.

Figure 3 lists the schedulers we implemented. Figures 4 and 5 do the same for non-contiguous and contiguous allocation algorithms respectively. Note that Figure 4 lists algorithm families due to the refactoring described in Section IV.

B. Task mapping

The main way to use PReMAS for task mapping is as a post-processing step. First, use PReMAS to generate a log of the nodes allocated to each job. This log has enough information to run the task mapping algorithms and compare the resulting mappings based on distance-based metrics. Performing task mapping outside the main simulation avoids recomputing the schedule and all the job allocations for each task mapping algorithm; recall that we do not yet model the effect of mapping quality on job running time. That said, we plan on adding this effect so we have also provided the ability to compute the mapping for each job as it is allocated.

C. Extensibility

Particular care was taken to make PReMAS easily extensible. This is the main reason for the use of an object-oriented design methodology and the many abstract classes. In addition, we used the reflection features of Java, which allow a program to refer to its own classes. In particular, each simulator component (e.g. scheduler or allocator) is created by a `Factory` object containing a map from names suitable for the program's command line to the corresponding subclass. These subclasses are registered in a static method of `Main`. For example, to populate the factory responsible for creating the `Machine` object (called `machineFactory`), the following code registers the two subclasses of `Machine`:

```
machineFactory.registerClass("simple",
                    SimpleMachine.class);
machineFactory.registerClass("mesh",
                    Mesh.class);
```

With this, the factory can process command line arguments to create the desired type of `Machine` using the static method `Make` supplied by each subclass. The factories also generate usage information via each class's static `getParamHelp` method. This has the effect of locating the help and creation information about each class within that class and requiring only a single new registration line to add another subclass.

IV. Lessons from algorithm refactoring

Now we discuss the lessons learned developing PReMAS. As part of trying to improve its software design, we tried to avoid duplicate code by factoring out common functionality into subroutines. Modest examples of this were the development of a single routine to process all the command line options that configure a simulator run (e.g. the scheduler, allocator, and machine) and centralizing the code to generate output files (so all have a common header describing the simulation that created them). Much more notable, however, are several times when refactoring to unify duplicate code led us to think about the algorithms differently.

A. Compression order in Conservative-style backfilling

The greatest intellectual benefit of refactoring we saw began with the implementation of scheduling with Conservative backfilling (Conservative) [30]. The idea behind Conservative is to mostly maintain a FCFS/FIFO job order except to allow *backfilling*, in which a job is started before other jobs that arrived earlier, when doing so does not delay any previously-arrived job. In order to do this, Conservative keeps a plan of when it expects to run each job. The position of each job within the plan is called its *reservation*; no matter what happens, each job is guaranteed to start no later than its reservation. When a new job arrives, Conservative adds it to the current plan at the earliest possible time (which becomes its reservation); since previously-arrived jobs are already in the plan, placing the new job will not delay any of them.

Our research focused on what happens when a job finishes early (due to a job's actual time being less than its estimated time), leaving a hole in the plan. If the schedule is rebuilt from scratch, it is possible for a job that was previously backfilled to violate its reservation. To avoid this, a special operation called *compression* is performed. In compression, each job is removed from the plan one at a time and rescheduled into the rest of the plan. Since each job can be returned to its previous place, compression never moves any job later in the plan. The exact procedure for compression is left slightly vague in the original paper [30], but the simplest implementation is to remove and reschedule jobs in the order they occur in the current plan. This is what our simulator does and also what the behavior of the (non-public) simulator described in the original paper [15]; for clarity, we use "Conservative" to mean the algorithm with this version of compression.

When we implemented Conservative, however, we noticed the potential for other compression procedures and parameterized them using a comparator. When a job finishes early, the jobs are rescheduled in the order given by the comparator. We called this algorithm Prioritized Compression (PC) and also considered a version called Delayed Compression (DC) that postpones compression unless the job being rescheduled can start immediately or a new job arrives. We first used PC and DC with the Shortest Job First order to improve job average waiting time and with the Widest Job First order to benefit jobs using a larger fraction of the system [25]. Later, we used the First-Come First-Served order to improve previously-proposed [34] measures of job-level fairness [31]. Other orders such as user-provided priority could also be explored.

B. Center-based allocation algorithms

A second example of refactoring benefiting the research occurred with what we now call "center-based allocation algorithms". We now present the algorithms and then discuss how the commonalities were identified and factored out.

Priority-queue	Run jobs in order indicated by a comparator. Includes FCFS, Shortest Job First, Widest Job First, etc. (`PQScheduler`)
EASY [24]	Backfilling with guarantee for first queued job. (`EASYScheduler`, `EASYContigScheduler`)
Conservative [30]	Backfilling with a guarantee for every job. See Section IV-A. (`StatefulScheduler`)
PC [25]	Variation of Conservative that uses a priority function to order rescheduling operations when a job finishes early. See Section IV-A. (`StatefulScheduler`)
DC [25]	Variation of PC that delays rescheduling operations if possible to increase the impact of the priority function. See Section IV-A. (`StatefulScheduler`)
Aggressive	Backfilling without limits. Note that some use this name for EASY. (`AggressiveScheduler`)
Scan [20]	Several variations on the idea of grouping jobs by size and rotating thru the sizes. (`ScanScheduler`)
Timed-Run [37]	Give each arriving job a quick run to look for errors before the main scheduling decision. Assumes jobs can be restarted. (`TimedRunScheduler`)

Fig. 3. Schedulers implemented in PReMAS, with providing class(es) in parentheses.

Center-based [5], [21]	Build candidate allocation centered on free nodes and possibly others. Use a scoring function to rate each candidate allocation and pick the best one. See Section IV-B. (`NearestAllocator`, `CenterGenerator`, `PointCollector`, `Scorer`)
Linear [22], [26]	Impose linear order on nodes and sort free list. Either select a prefix of it or treat groups of consecutive nodes as bins and apply a bin-packing heuristic to select from one of them. See Section IV-C. (`LinearAllocator`, `SortedFreeListAllocator`, `BestFitAllocator`, `FirstFitAllocator`)
Buddy [26], [38]	Organize entire system into hierarchical decomposition and use this for buddy system similar to memory allocation. Allocate multiple pieces to jobs as necessary. Includes several different algorithms based on the hierarchy used. (`MBSAllocator`, `OctetMBSAllocator`, `GranularMBSAllocator`)
Random	Allocate random free nodes. (`RandomAllocator`)

Fig. 4. Families of non-contiguous processor allocation algorithms implemented in PReMAS, with providing class(es) in parentheses.

First Fit [40]	Look for place to fit job and take first one in which it fits. (`FirstFitContigousAllocator`)
Best Fit [40]	Look at all places to fit job and take smallest one in which it fits. (`BestFitContiguousAllocator`)
MPL [1]	Prioritize places along mesh boundary. (`MPLAllocator`)

Fig. 5. Contiguous processor allocation algorithms implemented in PReMAS, with providing class(es) in parentheses.

One of these algorithms is MC1x1 [5], which is based on an algorithm MC [29] designed for a slightly different setting. In that setting, jobs have not just a number of desired nodes, but also a specific submesh shape, presumably related to their communication pattern. MC considers an allocation built using *shells* centered on each of the free nodes in the system. Shell 0 contains nodes in the job's desired shape around the center. Shell 1 is one larger in each direction and subsequent shells are formed using the same procedure. Figure 6 shows a series of shells for a 3×1 job centered on the node marked with X. Based on these shells, MC chooses a candidate allocation corresponding to the center by selecting the necessary numbers of nodes from the lowest-numbered shells. The *score* of a candidate allocation is the sum of its nodes' shell numbers. MC takes the allocation with the lowest score. MC1x1 adapts MC to our setting, where jobs do not have a desired shape, by setting shell 0 to a 1×1 submesh.

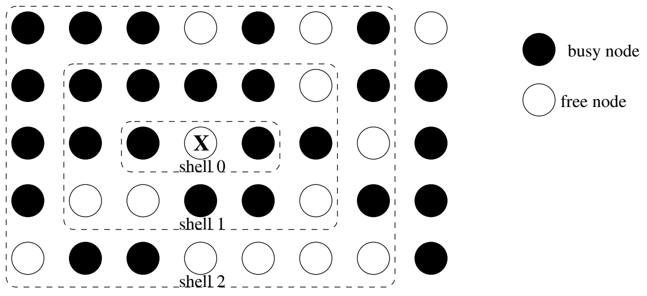

Fig. 6. Illustration of MC. Shells for a 3×1 job centered on the node X.

A second center-based allocation algorithm is Gen-Alg [21]. Gen-Alg was designed explicitly to minimize the sum of the pairwise L_1 distances between selected nodes. (Recall that the L_1 distance between two points, also called the *Manhattan distance*, is the sum of the differences in each coordinate.) As with MC1x1, Gen-Alg considers candidate allocations centered on free nodes and selects the best one, but it does each of these in a different way. To build a candidate allocation, it uses shells based on the L_1 distance from the center, which grow as shown in Figure 7. To select between these candidate allocations, it uses the sum of pairwise L_1 distances between the nodes in each allocation. Gen-Alg always gives an allocation whose sum of pairwise L_1 distances is no worse than $2 - 2/k$ times the best possible [21], where k is the allocation size.

Our final center-based allocation algorithm is MM [5]. It is the same as Gen-Alg except for considering more possible centers. In addition to the locations of free nodes, MM uses a candidate center at every position sharing x, y, and z coordinates with (possibly distinct) free nodes. Figure 8 illustrates this in two dimensions. MM always gives an allocation whose sum of pairwise L_1 distances is no worse than $2 - 1/(2d)$ times the best possible, where d is the system dimensionality [5]; this guarantee is $7/4$ in 2D and $11/6$ in 3D. The guarantee is stronger than Gen-Alg's for large jobs.

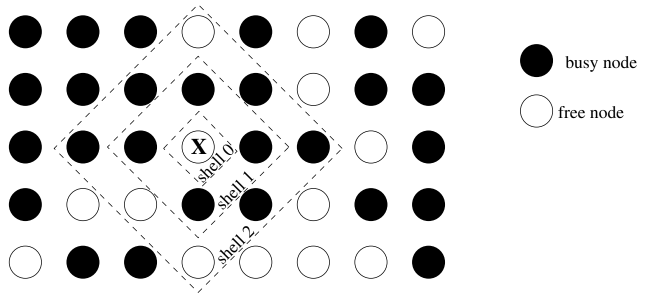

Fig. 7. Illustration of Gen-Alg. Shells for a job centered on the node X.

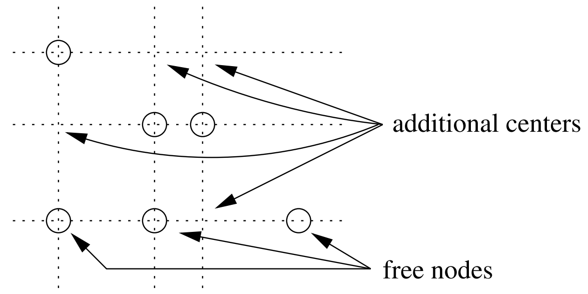

Fig. 8. Additional centers used by MM in 2D. In addition to free nodes (denoted with circles), MM also tries all positions sharing x and y coordinates with free nodes, i.e. every intersection of axis-parallel lines thru the free nodes.

While implementing these algorithms, we realized that they share a common framework. They each identify candidate centers, collect the nearest points according to some metric around each into a candidate allocation, and then score the candidate allocations. Thus, we implemented all these algorithms with a single class (`NearestAllocator`) whose constructor takes a specification for these decisions in the form of members of subclasses of the abstract classes `CenterGenerator`, `PointCollector`, and `Scorer`. Figure 9 specifies each of the algorithms above in this framework. Other center-based algorithms could easily be created by combining our subclasses in a different way or defining new ones. We have also found the common conceptual framework valuable in its own right as a way of thinking about different algorithms.

C. Linear allocation algorithms

Our final example of refactoring helping us think about algorithm design is in another class of processor allocation algorithms, which we call "linear allocation algorithms".

The common feature of these algorithms is that they assign a numerical order to the nodes and then use this order when

Algorithm	Candidate centers	Distance	Evaluation
MC1x1	free nodes	L_∞	L_∞ distance from center
Gen-Alg	free nodes	L_1	pairwise L_1 distance
MM	points sharing x and y coords with free nodes	L_1	pairwise L_1 distance

Fig. 9. Parameters that create known center-based algorithms

allocating nodes. The simplest such strategy is to use a sorted free list, allocating the lowest-numbered free nodes to a new job. This was initially used on CPlant systems [35], with the order determined by the physical location of hardware. Lo et al. [26] considered various logical orders (e.g. row-major based on the network connections) and Leung et al. [22] introduced the idea of ordering nodes based on their position along a space-filling curve. Leung et al. [22] also introduced the idea of grouping consecutive free nodes together and using bin-packing heuristics to select them. For example, the First Fit heuristic selects the earliest group that is large enough for the entire job and the Best Fit heuristic selects the smallest group that is large enough.

In PReMAS, an abstract class (`LinearAllocator`) provides the functionality common to all linear allocation algorithms, such as reading a file to define the node order. Then subclasses (`SortedFreeListAllocator`, `BestFitAllocator`, and `FirstFitAllocator`) implement particular selection heuristics. The division of linear algorithms into ordering and selection heuristic guided our research, leading to experiments to isolate the effect of the order [38].

V. Issues with reproducibility

In addition to these major lessons learned, implementing the simulator and using it over a period of time also revealed a number of smaller issues concerning the reproducibility of resource management research. Most of these are minor ambiguities with commonly-used data sets or algorithm descriptions. We do not believe that any are individually important, but they and issues like them threaten our ability to reproduce the work of others. This is an important capability if our community is to have a true academic conversation in which researchers validate and improve upon each other's work. Promoting this capability is one reason for having a community code base and thus why we are distributing PReMAS.

The first issue concerns the traces in the Parallel Workloads Archive [13]. Many of the traces have different versions, including some that are "cleaned" to remove jobs felt to be unrepresentative. Each job line also includes a job status field, which marks some lines as reporting scheduler actions (i.e. suspending a job) rather than new jobs. Together, the different traces and the status values can create a multitude of choices for each trace. We followed the advice of the Archive by using cleaned versions and ignoring jobs with status 2, 3, and 4 (suspended jobs). We also ignored jobs with running time or status listed as -1 ("unknown") and jobs that were canceled (status 5) before being started. We have also tried to consistently state the trace version and our criteria for removing jobs when presenting our work. This is more clear than many papers using these traces, but even so it is hard to know if others would generate the same job sets from our description. Reproducibility is better served by sharing the code.

The next issue has to do with ties in event times. Since all trace times are in seconds, several job arrivals can happen "simultaneously", as can job arrivals and completions. Since jobs are scheduled as they arrive and the number of free nodes affects what happens, different orders can easily yield different schedules. As an aid to reproducibility, we systematically broke ties so that job completions happen before arrivals and broke ties regarding job order using their order in the trace. Since job completions can themselves trigger the scheduler, there is the added choice of whether to schedule new jobs after every job completion or after all simultaneous completions; we opted for the latter. We have never discussed these decisions in prior publications, nor have we seen anyone else do so.

A special kind of tie happens with 0-length jobs, which can have a tie between their start and completion times. These jobs appear in the traces when a job finishes in the same second it started. They caused significant complexity in our implementation of Conservative, which must keep a plan of when each job is expected to run. Even though they run instantly, 0-length jobs can still block other jobs from starting if they (briefly) use nodes needed by the other job. In order to deal with these jobs, we had to treat planned starts and completions of 0-length jobs separately from the planned starts and completions of other jobs. Other options would be to remove the jobs or (better) to give them short but non-zero duration. This issue does not affect a large number of jobs, but the variety of choices makes it a threat to reproducibility. In addition, the complexity of this code increases the value of having a shared code base since Conservative proved so challenging to implement and it is a core algorithm used in many scheduling studies.

We also found ties to be meaningful for center-based processor allocation algorithms (described in Section IV-B). Recall that these algorithms build candidate allocations using free nodes in shells of nodes at increasing distances from a candidate center. If the algorithm only needs some of the free nodes in the outermost used shell, then it must select between them. We encountered this when implementing MC1x1, the first implementation of which made some clearly inferior choices when selecting nodes. Specifically, for jobs requesting 2 nodes, it sometimes chose a pair that were diagonal from each other (i.e. at distance 2) rather than an adjacent pair. We fixed this, but the description of MC1x1 leaves considerable leeway as to how nodes should be selected from the outermost shell.

Ties in center-based processor allocation became an issue again when that part of the code was ported into C++ for inclusion into SST. The output of the C++ and Java versions were compared as a debugging check and were found to differ. This occurred because the candidate allocations were assembled by sorting the free nodes and each language provided a different library sorting routine. This experience is both an argument for completely resolving all ties and a warning about the difficulty of maintaining equivalent programs in different languages.

Ties can also occur in many other settings. For example, when selecting a job based on its width for width-based scheduling or backfilling, which job is selected in a tie? When rescheduling jobs in Conservative's compression operation, which job goes first if several jobs have a common planned starting time? In each of these cases, we tried to break the ties in a consistent and reasonable way, using a numbering of jobs and events when necessary, but our orderings are not necessarily the same as others would choose. The issue reminds us of the assumption of general position in computational geometry, a topic which generated significant discussion in that area (e.g. [7], [12]). It may be worthwhile for our community to consider a systematic approach to tie breaking as well.

VI. Using the simulator with undergraduates at Knox College

As mentioned previously, an important aspect of our simulator is that it has mostly been used and developed by undergraduate students. Since our hope is that others can use it similarly, we now discuss this aspect of our experience.

The majority of students using this simulator were summer researchers. These students worked 40 hours a week for 10 weeks. The group of students (1–4 depending on the year) met daily with one of us (Bunde) as a group and also individually with him as needed. The first couple of meetings focused on discussing a series of papers related to the research topic. Later meetings increasingly shift to discussing their progress and developing future plans. The students are given some discretion if a particular research question excites them, but we provide specific problems and most students work on these.

The simulator has also been used by some students in coursework, particularly in a senior seminar course. This course is mainly a research seminar, with most of the term spent discussing research papers, and only a small hands-on component using the simulator. Some also used it in their term project. These projects are smaller in scope than the summer projects, however, and none have been publishable to date.

The best approach we have found for introducing students to the simulator is to have them run some of its existing functionality and then give them a specific task involving a small part of it, generally implementing a specific algorithm from the literature. Initially it is important to provide close guidance while the students learn about the simulator's overall structure. Because they have no prior research experience and only limited familarity with the literature, it is particularly difficult for the students to interpret their results and go from output graphs to research "stories". This is the main activity that occupies later daily meetings as we and the students try to understand their output and revise algorithms to improve the results. Due to both their limited background and the limited time, we have also found it necessary to do most of the writing about the project.

VII. Related work

Now we briefly summarize other research related to PReMAS. Simulation is a huge area, with entire conferences dedicated to it (e.g. the Winter Simulation Conference, PMBS, and SIMUTools). Despite this, we continue to observe that most research groups use their own home-grown simulators.

There are some other publicly-available simulators for resource management problems. GridSim [17] and CloudSim [9] are freely available, but focus on different kinds of distributed systems than PReMAS. SimGrid [36] provides functionality to simulate a wide variety of systems. Its SMPI component [8] allows detailed simulation of individual MPI jobs, including network effects. This could be used to investigate the effects of processor allocation and task mapping decisions at the job level; PReMAS uses a simpler model for each job, but does a system-level simulation to see the effect of allocation policy (for example) over an entire trace.

Another option is ProcSimity [27], which was designed for research in processor allocation. It both supports distance-based metrics such as we use and has the ability to model message flits moving through the network. Despite these appealing features, ProcSimity lacks the modularity and easy extensibility of PReMAS. A bigger issue is its handling of time; despite the varying timescales of events (job arrivals vs flits traversing a switch), only a single unit of time is used. In practice with Parallel Workload traces, this means that all time units are in seconds and the network is highly overloaded.

Eventually, the Structural Simulation Toolkit (SST) [33], to which some of our code has been ported, may also be a competitor for PReMAS. It is freely-available and aims to allow system simulation at a variety of levels. Currently, however, much of the functionality is for simulations at much lower levels (e.g. architecture) and the code ported from PReMAS provides much of the high-level functionality. A more general objection is that the size and complexity of SST makes it challenging to use; compiling the SST code requires installing several other software packages.

There are also specific other simulators of interest for large systems, such as BigSim [39] and MARS [10], but neither of these is publicly available.

In addition to specific other simulators, Frachtenberg and Feitelson [16] give a related discussion about best practices for scheduling simulation.

VIII. Discussion

We hope that others find this discussion and our simulator code useful, but it will certainly not be the last word. Realizing a shared code base requires a community discussion and effort. We are also quite aware that using and adapting code written by others can be challenging. This is particularly true when the code involved was written as a research prototype without plans for long-term maintenance; our own code certainly suffers from this, though we are working to improve the situation. In particular, we hope to use the code as an example in software development classes, both as an example of the issue of software maintenance and as the subject of assignments on design and testing.

It is also clear that others will want to create forks of this code rather than using and developing for it directly. One reason for this is our choice of programming language: Java is convenient for our simulator since it is the main programming language for most of the student researchers who participate in our project, but we know that others will prefer other languages. (We have already seen this in the integration of some of our code into SST, which required a port into C++.) Forking the code base potentially limits its usefulness by increasing the difficulty of comparing results or incorporating the improvements of others. One way that even a split code base could provide many of the advantages we see is for the community to standardize the interfaces between components. (By "interface", we mean the functions and their arguments rather than anything language specific.) Standardization at this level (and code being generally available) would greatly facilitate translating algorithms between simulators even with variations in the rest of the code base.

Beyond these general issues, there are many possible improvements to the simulator itself. Consider the following:

- Visualization— A natural extension is to provide tools for visualization. We have a tool for small-scale visualizations of allocations, but the tool cannot handle large traces. Expanding this and/or adding tools to visualize schedules and task mappings could make it easier to compare algorithms with more than high-level numeric measures such as average response time and average pairwise distance.
- Workload models— So far, the simulator has been run entirely using traces. An alternative is to incorporate some of the many workload models (see e.g. [14]), either within the simulator or as a preprocessor to generate a trace.
- Modeling messages— Research in task mapping and, to a lesser degree, processor allocation would benefit from a more detailed simulation that would better model the congestion on particular links. The obvious way to do this is to model individual messages, but one representation at an intermediate level of detail is to record communication by marking the links used with the aggregate amount of traffic.
- Performance improvements— Some runs, particularly those including processor allocation on large systems, can take days or weeks to execute. Likely some of the algorithm implementations can be improved by reconsidering them in the light of larger systems. This is likely important as HPC systems keep becoming larger. Parallelization of the simulator itself is also a natural approach.

There are also smaller projects, such as incorporating additional algorithms from the literature. We plan on pursuing these improvements, but we also encourage others to use our code and share the enhancements made in the course of their work.

Acknowledgment

We thank the anonymous reviewers for their helpful suggestions. We also thank all of the students who have contributed to PReMAS over the years and Jaime Spacco for his help preparing it for public release. D.P. Bunde was partially supported by contract 899808 from Sandia National Laboratories. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

References

[1] Ismail Ababneh. On submesh allocation for 2d mesh multicomputers using the free-list approach: Global placement schemes. *Performance Evaluation*, 66(2):105–120, 2009.

[2] J.A. Ang, R.A. Ballance, L.A. Fisk, J.R. Johnston, and K.T. Pedretti. Red storm capability computing queuing policy. In *Cray Users Group*, 2005.

[3] E. Balzuweit, D.P. Bunde, V.J. Leung, A. Finley, and A.C.S. Lee. Local search to improve task mapping. In *Proc. 7th Intern. Workshop Parallel Programming Models and Systems Software for High-End Computing (P2S2)*, 2014.

[4] R.F. Barrett, C.T. Vaughan, S.D. Hammond, and D. Roweth. Reducing the bulk in the bulk synchronous parallel mode. *Parallel processing letters*, to appear.

[5] M.A. Bender, D.P. Bunde, E.D. Demaine, S.P. Fekete, V.J. Leung, H. Meijer, and C.A. Phillips. Communication-aware processor allocation for supercomputers: Finding point sets of small average distance. *Algorithmica*, 50(2):279–298, 2008.

[6] D.P. Bunde, J. Ebbers, S.P. Feer, V.J. Leung, N.W. Price, Z.D. Rhodes, and M. Swank. Task mapping for non-contiguous allocations. Poster presented at *Conf. High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

[7] C. Burnikel, K. Mehlhorn, and S. Schirra. On degeneracy in geometric computations. In *Proc. 5th Annual ACM-SIAM Symp. Discrete Algorithms (SODA)*, pages 16–23, 1994.

[8] P.-N. Clauss, M. Stillwell, S. Genaud, F. Suter, H. Casanova, and M. Quinson. Single node on-line simulation of MPI applications with SMPI. In *Proc. 25th IEEE Intern. Parallel and Distributed Processing Symp. (IPDPS)*, pages 664–675, 2011.

[9] Cloudsim: A framework for modeling and simulation of cloud computing infrastructures and services. http://www.cloudbus.org/cloudsim/.

[10] W.E. Denzel, J. Li, P. Walker, and Y. Jin. A framework for end-to-end simulation of high-performance computing systems. In *Proc. 1st Intern. Conf. Simulation tools and techniques for communications, networks and systems & workshops (SIMUTools)*, page 21, 2008.

[11] M. Deveci, S. Rajamanickam, V.J. Leung, K.T. Pedretti, S.L. Olivier, D.P. Bunde, Ü.V. Çatalyürek, and K.D. Devine. Exploiting geometric partitioning in task mapping for parallel computers. In *Proc. 28th IEEE Intern. Parallel and Distributed Processing Symp. (IPDPS)*, 2014.

[12] I.Z. Emiris and J.F. Canny. A general approach to removing degeneracies. *SIAM J. Comput.*, 24(3):650–664, 1995.

[13] D. Feitelson. The parallel workloads archive. http://www.cs.huji.ac.il/labs/parallel/workload/index.html.

[14] D. Feitelson. The parallel workloads archive: Models. http://www.cs.huji.ac.il/labs/parallel/workload/models.html.

[15] D.G. Feitelson. Personal communication by email, 2013.

[16] E. Frachtenberg and D.G. Feitelson. Pitfalls in parallel job scheduling evaluation. In *Proc. 11th Workshop Job Scheduling Strategies for Parallel Processing (JSSPP)*, volume 3834 of *LNCS*, pages 257–282, 2005.

[17] Gridsim: A grid simulation toolkit for resource modelling and application scheduling for parallel and distributed computing. http://www.buyya.com/gridsim/.

[18] F. Gygi, Erik W. Draeger, M. Schulz, B.R. de Supinski, J.A. Gunnels, V. Austel, J.C. Sexton, F. Franchetti, S. Kral, C.W. Ueberhuber, and J. Lorenz. Large-scale electronic structure calculations of high-Z metals on the BlueGene/L platform. In *Proc. ACM/IEEE Conf. High Performance Networking and Computing (SC)*, 2006.

[19] C.R. Johnson, D.P. Bunde, and V.J. Leung. A tie-breaking strategy for processor allocation in meshes. In *Proc. 6th Intern. Workshop Scheduling and Resource Management for Parallel and Distributed Systems*, 2010.

[20] P. Krueger, T.-H. Lai, and V.A. Dixit-Radiya. Job scheduling is more important than processor allocation for hypercube computers. *IEEE Trans. Parallel and Distributed Systems*, 5(5):488–497, 1994.

[21] S.O. Krumke, M.V. Marathe, H. Noltemeier, V. Radhakrishnan, S.S. Ravi, and D.J. Rosenkrantz. Compact location problems. *Theoretical Computer Science*, 181(2):379–404, 1997.

[22] V. Leung, E. Arkin, M. Bender, D. Bunde, J. Johnston, A. Lal, J. Mitchell, C. Phillips, and S. Seiden. Processor allocation on Cplant: Achieving general processor locality using one-dimensional allocation strategies. In *Proc. 4th IEEE Intern. Conf. on Cluster Computing*, pages 296–304, 2002.

[23] V.J. Leung, D.P. Bunde, J. Ebbers, S.P. Feer, N.W. Price, Z.D. Rhodes, and M. Swank. Task mapping stencil computations for non-contiguous allocations. In *Proc. 19th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP)*, pages 377–378, 2014.

[24] D. Lifka. The ANL/IBM SP scheduling system. In *Proc. 1st Workshop Job Scheduling Strategies for Parallel Processing (JSSPP)*, volume 949 of *LNCS*, pages 295–303, 1995.

[25] A.M. Lindsay, M. Galloway-Carson, C.R. Johnson, D.P. Bunde, and V.J. Leung. Backfilling with guarantees made as jobs arrive. *Concurrency and Computation: Practice and Experience*, 25(4):513–523, 2013.

[26] V. Lo, K. Windisch, W. Liu, and B. Nitzberg. Non-contiguous processor allocation algorithms for mesh-connected multicomputers. *IEEE Trans. Parallel and Distributed Systems*, 8(7):712–726, 1997.

[27] V.M. Lo. Procsimity. http://www.cs.uoregon.edu/research/DistributedComputing/ProcSimity.html.

[28] Los Alamos National Laboratory. High-performance computing: Cielo supercomputer. http://www.lanl.gov/orgs/hps/cielo/index.html.

[29] J. Mache, V. Lo, and K. Windisch. Minimizing message-passing contention in fragmentation-free processor allocation. In *Proc. 10th IASTED Intern. Conf. Parallel and Distributed Computing and Systems*, pages 120–124, 1997.

[30] A.W. Mu'alem and D.G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Trans. Parallel and Distributed Syst.*, 12(6):529–543, 2001.

[31] A. Rajbhandary, D.P. Bunde, and V.J. Leung. Variations of conservative backfilling to improve fairness. In *Proc. 17th Workshop Job Scheduling Strategies for Parallel Processing (JSSPP)*, 2013.

[32] A. Rodrigues, K. Bergman, D. Bunde, E. Cooper-Balis, K. Ferreira, K.S. Hemmert, B. Barrett, C. Versaggi, R. Hendry, B. Jacob, H. Kim, V. Leung, M. Levenhagen, M. Rasquinha, R. Riesen, P. Rosenfeld, M. del Carmen Ruiz Varela, , and S. Yalamanchili. Improvements to the structural simulation toolkit. In *Proc. 5th Intern. ICST Conf. Simulation Tools and Techniques (SIMUTools)*, 2012.

[33] A.F. Rodrigues, K.S. Hemmert, B.W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, and B. Jacob. The structural simulation toolkit. *SIGMETRICS Perform. Eval. Rev.*, 38:37–42, 2011.

[34] G. Sabin and P. Sadayappan. Unfairness metrics for space-sharing parallel job schedulers. In *Proc. 11th Workshop Job Scheduling Strategies for Parallel Processing (JSSPP)*, volume 3834 of *LNCS*, pages 238–256, 2005.

[35] Sandia National Laboratories. The Computational Plant Project. http://www.cs.sandia.gov/cplant.

[36] Simgrid — Versatile simulation of distributed systems. http://simgrid.gforge.inria.fr/.

[37] O. Thebe, D.P. Bunde, and V.J. Leung. Scheduling restartable jobs with short test runs. In *Proc. 14th Workshop Job Scheduling Strategies for Parallel Processing (JSSPP)*, volume 5798 of *LNCS*, pages 116–137, 2009.

[38] P. Walker, D.P. Bunde, and V.J. Leung. Faster high-quality processor allocation. In *Proc. 11th LCI Intern. Conf. High-Performance Clustered Computing*, 2010.

[39] G. Zheng, T. Wilmarth, P. Jagadishprasad, and L. Kale. Simulation-based performance prediction for large parallel machines. *International J. Parallel Programming*, pages 183–207, 2005.

[40] Y. Zhu. Efficient processor allocation strategies for mesh-connected parallel computers. *J. Parallel and Distributed Computing*, 16:328–337, 1992.