Towards Elastic Resource Management

Isaías A. Comprés Ureña and Michael Gerndt

Abstract A new paradigm for HPC Resource Management, called Elastic Computing, is under development at the Invasive Computing Transregional Collaborative Research Center. An extension to MPI for programming elastic applications and a resource manager were implemented. The resource manager is an extension of the SLURM batch scheduler. Resource elasticity allows the resource manager to dictate changes in the resource allocations of running applications based on scheduler decisions. These resource allocation changes are decided by the scheduler based on performance feedback from the applications. The collection of performance feedback from running applications poses unique challenges for the runtime system. In this document, our current performance feedback system is presented.

1 Introduction

Distributed computing systems are expected to deliver performance that is commensurate to their available hardware resources. This is achieved by the optimization of system-wide performance metrics. The optimization of these performance metrics is a task usually delegated to schedulers. In the case of distributed systems, schedulers take as input the jobs to be performed and the set of available compute resources. They produce as output the job startup order and the resources where they will be executed. These orders are referred to as schedules. These schedules affect the per-

Support for this work was provided by the Transregional Collaborative Research Centre 89: Invasive Computing (InvasIC) [29].

I. A. Comprés Ureña (✉) · M. Gerndt
Technical University of Munich (TUM), Rheinstrasse 5,
80803 München, Germany
e-mail: isaias.compres@tum.de

C. Niethammer et al. (eds.), *Tools for High Performance Computing 2017*,
https://doi.org/10.1007/978-3-030-11987-4_7

formance of individual applications and whole systems, and therefore determine the quality of schedulers.

The terms resource manager and scheduler are sometimes used interchangeably. In reality, these are different software components that are often bundled together due to their equal importance. Distributed systems need both a resource manager and a scheduler in order to share their resources with multiple users in a fair and efficient manner.

This document begins with an introduction to the general multiprocessor scheduling, the batch scheduling and the runtime scheduling problems, to help illustrate the need for performance feedback on resource-elastic systems. The additional features of the resource manager that provide performance feedback to the scheduler are described afterwards.

2 Theoretical Background on Multiprocessor Scheduling

The general multiprocessor scheduling problem is stated in an abstract manner in this section. The problem statement for batch scheduling with static resource allocations is presented after that, together with a short discussion on the taxonomy of schedulers and how it is classified. This problem statement is then extended to fit the more specific elastic scheduling problem addressed in this work. New requirements are identified from the new problem statement.

2.1 Problem Statement

Multiprocessor scheduling is an optimization problem that can be stated verbally as follows: given a set of tasks to be completed and a set of resources that can complete them by some means, find an assignment of tasks to resources that optimizes a set of objective functions. The tasks are bounded in time and may require collectively more resources than are available simultaneously; therefore, the assignment of tasks to resources may also require an order. Different orders can produce different outputs of the objective functions.

We can define the problem of scheduling more rigorously. Let T be a set of tasks t_i where the subscript $i \in \mathbb{N}$ identifies each task uniquely; this set may or may not be finite. Similarly, let R be a set of m resources r_j where the subscript $\{j \in \mathbb{N} \mid j < m\}$ identifies each resource uniquely. One or more resources in R can perform the tasks in T in some manner. If $\tau(t_i) \in \mathbb{R}$ is the maximum execution time and $\rho(t_i) \in \mathbb{N}$ the number of resources required to perform a task t_i, then we can define multiprocessor scheduling as the following optimization problem:

$$\begin{aligned} &\text{given inputs} && T = \{t_i \mid \tau(t_i) < \infty \wedge \rho(t_i) \leq m\}, \\ &&& R = \{r_j \mid j < m\} \\ &\text{compute a} && S = \{t_i \mapsto \varrho_i\} \\ &\text{that optimizes} && \sum_{k=0}^{w} O_k \end{aligned} \tag{1}$$

The result of this optimization is a schedule S. The schedule is a set of mappings from individual tasks t_i into specific subsets of resources ϱ_i of size $\rho(t_i)$, where $\varrho_i \subset R$. Tasks where $\rho(t_i) > m$ are impossible to schedule and therefore not considered.

Objective functions typically produce single scalar values in $\mathbb{R}$ within the range $[0, \infty)$. By optimizing (either minimizing or maximizing) the sum of the output of each O_k objective function (e.g. idle node time), where $\{k \in \mathbb{N} \mid k < w\}$, the quality of the produced schedule can be improved. Different objective functions can evaluate the quality of full schedules S or individual mappings $t_i \mapsto \varrho_i$. This allows schedulers to optimize based on system-wide metrics, performance metrics of individual applications, or both.

The sum of all required resources of the tasks in T may exceed the total number of resources m in R:

$$\sum \rho(t_i) > m \tag{2}$$

In such a condition all tasks cannot be started simultaneously at the earliest time of the schedule $\{\delta_0 \in \mathbb{R} \mid \delta_0 > 0\}$. Because of this, both a starting time and duration need to be added as part of each mapping in the schedule when resource sharing is not allowed. Each mapping then becomes a reservation of resources with a starting time $\delta_i \geq \delta_0$ and the duration of its task $\tau(t_i)$, in addition to its set of unique resources ϱ_i. A schedule S then becomes:

$$S = \{t_i \mapsto \langle \varrho_i, \delta_i, \tau(t_i) \rangle\} \tag{3}$$

This modification to S can be inserted in the initial optimization problem definition (Eq. 1) to indicate that schedules need to be produced with these additional timing specifications.

2.2 Computational Complexity

The theoretical complexity of the multiprocessor scheduling problem can be determined with the aid of complexity theory. The goal is to determine the asymptotic complexity of the optimization problem based on its inputs. A bound to the number of steps of possible algorithms, based on the number of steps required to reach a solution, should be determined. Thankfully, this topic has been of great interest to researchers and results from previous analyses [10, 12, 17, 19, 21] are available.

The multiprocessor scheduling problem belongs to a family of problems that have no known solutions of polynomial or lower complexity [6, 7, 18, 31]. It is for this reason that current schedulers rely on approximation algorithms that are based on heuristics. These algorithms settle for solutions that are feasible but not necessarily optimal; the assumption is that in most cases adequate heuristics guide the approximations so that produced schedules approach optimal results, based on a set of objective functions.

2.3 *Resource-Static Scheduling in Distributed Memory HPC Systems*

A scheduling problem for specific compute systems, in a more concrete way, can be classified by several characteristics related to its set of tasks, its set of resources and its method used to generate the output schedule. There have been several efforts to create a taxonomy of scheduling problems [5, 13, 21, 22, 25]. The scheduling problem in distributed HPC systems is clearly defined [8, 9, 16, 23, 26] for current resource-static execution models. Current solutions consist generally of First-Come First-Serve (FCFS) batch scheduling with static allocations and backfilling.

Current supercomputing systems are usually shared among several researchers across multiple institutions. Individual tasks are submitted to these systems by its users, in the form of batch job definitions. The arrival rate of these job definitions can be modeled with the aid of queueing theory. Batch job definitions include their number of resources required, their priority and their maximum execution time, among several other aspects that may not be as important to schedulers. Batch job definitions are entered in a queue. This queue represents the input task set T of the optimization problem 1.

The resources of current supercomputing systems tend to be similar. In most systems, the hardware on each node is identical. There may be cases where the nodes have heterogeneity internally (e.g., in the form of accelerators). A node is abstracted as a single resource in most cases. This means that in spite of the growing amount of parallelism internally at each node, schedulers operate on full nodes, instead of subsets of cores or even accelerators where available.

The operation of schedulers is currently divided in two steps: batch scheduling and backfilling. The batch scheduling step scans a window of the job queue and attempts to start as many jobs as possible based on their priority. When a job cannot be started immediately, it may instead get a resource reservation in the future. Once this first step is done, the scheduler proceeds to the backfilling step: it attempts to start jobs that fit in the gaps of remaining idle resources. Jobs that are started during this second step should not delay the start of higher priority jobs that have reservations.

The general strategy is illustrated in Fig. 1. It presents a scenario with four nodes, a job queue of six jobs with a priority based order. In the illustration, a schedule is computed where job 4 receives a reservation later than jobs 5 and 6 due to the

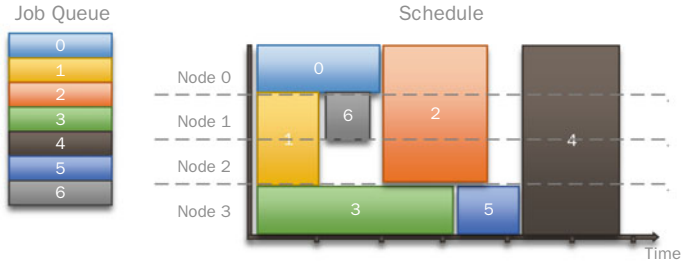

Fig. 1 Possible schedule of a set of static jobs ordered by priority in a queue

availability of resources. In the same schedule, job 6 is scheduled early to minimize idle nodes through a backfilling operation.

In summary, static batch scheduling with backfilling on current distributed systems has the following task set, resource set and algorithm properties:

- Task set:
  - Set properties:
    Multiple users submit tasks
    Tasks submitted randomly
    Unbounded task capacity
    Best effort First-In, First-Out (FIFO)
    Tasks are removed on completion
  - Task properties:
    Set of one or more tasks as jobs
    Jobs are time bounded
    Jobs and tasks are not periodic
    Fixed number of resources specified
    Jobs receive exclusive access to resources
    No Service Level Agreements (SLAs)
- Resource Set
  - Symmetric Multiprocessing (SMP) nodes as resources
  - Nodes have identical hardware (homogeneous)
  - Nodes may have attached accelerators
  - No quality of service (QoS) support
  - Resources are finite and cannot be scaled on demand
  - Resources are located in a single building
  - Power and energy scaling features available
  - No job or task migration support
  - No fault tolerance support

- Algorithm
  - Nodes as the units of resources
  - Job level scheduling (no task level scheduling)
  - Objective functions for mainly system-wide performance metrics
  - Two step *resource-static* scheduling
    Batch scheduling with priority based FIFO
    Backfilling to minimize idle nodes
  - Scheduling without performance guarantees
  - Scheduling without reactive adjustments
  - Jobs cannot be preempted

2.4 Modified Scheduling Problem for Resource-Elastic Execution

The scheduling problem described so far applies to cases where only static allocations are possible. Static allocations mean that the resource reservation of a job stays constant throughout its execution. The scheduling problem needs to be updated if the resource allocation of a job can change during the runtime of its tasks; resources may increase (expansion), decrease (reduction) or the unique nodes allocated to a job may change while their total stays the same (migration).

The current scheduling problem, solved with batch scheduling and backfilling, needs to be modified to include the added flexibility of resource-elastic execution. Only the properties of the jobs in the task set need to be modified:

- Jobs have a range of feasible resource counts.
- Jobs have a time bound that is a function of its resources.

This modified scheduling problem remains very similar to the preexisting one due to only these two differences. All other mentioned properties in the previous section remain. Jobs still retain exclusive access to the resources on its resource allocations, although some resources may be added or removed from this allocation at runtime. Due to this, the time required for the job to complete becomes dependent of the number of resources in time. In general, jobs will still provide a worst case time bound as part of its description.

Although similar to the preexisting scheduling problem, these two differences in the properties of jobs add new requirements to the algorithm of a potential scheduler. In addition to the previous batch and backfilling steps, a scheduler for HPC systems with resource-elastic execution capabilities must also:

1. Continuously monitor the performance of the tasks of running jobs.
2. Adjust the resource allocations of jobs based on their observed performance.

In the proposed design, the first activity is delegated to the resource management infrastructure, while the second activity is delegated to to a scheduler that is under

development. Most of the traditional batch-scheduling activities are still handled by a more traditional scheduler. In the remainder of this document, the way the first activity is carried out by the infrastructure will be described. The scheduler that is currently under development will not be covered in this document.

3 Performance Monitoring Infrastructure

The performance of individual jobs is monitored by the infrastructure. The infrastructure is composed of the MPI library and the resource manager components. Performance data is captured and a performance model is built. The performance model is then used to drive scheduling decisions.

The collection of data is performed in a hierarchical manner. At the lower level, each MPI library linked into each application process detects the structure of the computation in the local process and collects performance data. This structure is then reduced to a node-local representation by the `SLURMD` daemon at each node. Finally, the scheduler performs a final reduction to create the individual performance model of the distributed application. The set of models of all running applications are used to drive scheduling decisions.

3.1 Process-Local Pattern Detection and Performance Measurements

At the process-local view, the MPI library linked to the process performs pattern detection and performance metrics evaluations. The pattern of computation is detected before any performance metric is determined, since these metrics will be attached to specific control flow locations only after they are detected. Process local operations are kept to a minimum once a pattern is detected.

3.1.1 Pattern Detection

Since the pattern detection is intended to occur during the actual production run of applications, the minimization of its performance impact is of great importance. Because of this, the structure of computation is detected based on markers introduced by the compilation wrappers provided by the MPI library (`mpicc` and `mpifc` in this case). There have been previous works that rely on backtracing the sequence of calls in a program to determine unique locations during execution. These are then used as identifiers for pattern detection [1–4, 11, 15], such as loops, in MPI applications. The introduction of these markers at compilation time eliminates the overhead related to backtracing, although the technique is limited to binaries generated within a single software project.

The markers are inserted by splitting the compilation of objects into the emission of assembler and the final assembly step. Thankfully, most modern compilers have support for these operations. In the current implementation, the compiler wrapper works with Intel and GNU compilers. Versions 10.0 and later of the Intel compilers were tested, while versions 4.9 and later were tested for the GNU compilers. Other compilers were not tested, since those are the ones available in the SuperMUC system where this work was evaluated.

The current wrapper based technique relies on the way these compilers generate library calls in the emitted assembler. The actual API names of library calls are preserved, when linking C based libraries. Fortunately, MPI is a pure C based library and its calls can be easily identified with text processing in the intermediate assembler of each target object of the compilation. Additionally, since the MPI standard requires that any operation with the `MPI_` prefix be provided only by the MPI library in compliant programs, it is guaranteed that only MPI operations will be intercepted. Additionally, the `PMPI_` pattern can be selected to preserve support for any PMPI based profilers and tools.

Once the MPI calls are identified in the assembler code, a unique ID is computed and inserted before the MPI call through an operation available in the MPI library. This operation is currently called `MPIX_T_set_call_site_identifier`, and as its prefix `MPIX_T` suggests, it is a non-standard addition to the MPI tools interface. This tooling call sets the identifier for the device layer of the layered software architecture inherited from MPICH. This operation sets an integer identifier that is later read by the library at each individual MPI operation. This identifier establishes the uniqueness of the call site without the need of backtracing.

The MPI library relies on these markers to detect the structure of the computation at runtime. There have been several algorithms developed to detect patterns in sequences [14, 20, 24, 28, 30]. A pattern detection algorithm, that was originally designed to analyze programs from decompilation, fits well this pattern detection use case [32]; this algorithm is also used in other recent related works [2].

The pattern detection algorithm was implemented within the MPI library. In the current implementation, the algorithm provides the following output information to the runtime system, based on the current partial sequence of call site identifiers provided to it as input:

1. The detected Control Flow Graph (CFG).
2. Each node of the CFG is annotated with its number of revisits.
3. Nodes that are the heads of unique loops are marked.
4. Nodes that are the tails of unique loops are marked.
5. Nodes that are reentry points from nested loops are marked.

The CFG update routine is called at relevant MPI operations with their unique identifiers and types. There are different operation types for point-to-point, one-sided, collectives, MPI-IO, etc. The MPI library has an operation that serializes its local CFG to a shared memory segment, where it can then be read directly by the local daemon. Unique blocks of shared memory are dedicated to each MPI rank in the node.

An example can be used to better explain the algorithm's behavior. Listing 1 shows the log output of a single MPI process given the sequence of identifiers:

```
2 0 6 3 1 6 3 1 6 3 1 9 7 9 7 3 1 6 3 1 6 3 1 9 7 9 7
```

The detector can produce a text representation of its current CFG, in tabular form, as logging output. Listing 2 shows the detected CFG that matches the previous sequence. Each output row represents a node in the CFG. The first column is the address in the local memory of the process. The second columns is the identifier number. After that, the loop head flag (H), the loop body flag (B), the reentry counter (Re) and the revisit counter (Rv) are provided. The final two columns provide the tail data of loop heads, and the head data of loop body nodes. As seen in Listing 1, there is also a time differential (TD) computed at each step. In the current implementation, the time resolution of this differential is in nanoseconds. The time of creation is set each time a new node is added to the CFG. Total differential times from head nodes are accumulated on node revisits. The average distance in time from the head node of a loop to any node in the body can therefore be computed by dividing the accumulated differential by its total number of revisits.

```
 0: root id: 2
 1: id: 0; detected: 0; -> NOT in a loop; (TD: 4638)
 2: id: 6; detected: 0; -> NOT in a loop; (TD: 10243)
 3: id: 3; detected: 0; -> NOT in a loop; (TD: 14440)
 4: id: 1; detected: 0; -> NOT in a loop; (TD: 17938)
 5: id: 6; detected: 1; -> head: 6; (TD: 22178)
 6: id: 3; detected: 1; -> head: 6; (TD: 26174)
 7: id: 1; detected: 1; -> head: 6; (TD: 30090)
 8: id: 6; detected: 1; -> head: 6; (TD: 33756)
 9: id: 3; detected: 1; -> head: 6; (TD: 37407)
10: id: 1; detected: 1; -> head: 6; (TD: 41180)
11: id: 9; detected: 0; -> NOT in a loop; (TD: 44758)
12: id: 7; detected: 0; -> NOT in a loop; (TD: 48493)
13: id: 9; detected: 1; -> head: 9; (TD: 52336)
14: id: 7; detected: 1; -> head: 9; (TD: 56155)
15: id: 3; detected: 1; -> body re-entry; head: 6; (TD: 60054)
16: id: 1; detected: 1; -> head: 6; (TD: 63853)
17: id: 6; detected: 1; -> head: 6; (TD: 67418)
18: id: 3; detected: 1; -> head: 6; (TD: 70916)
19: id: 1; detected: 1; -> head: 6; (TD: 74361)
20: id: 6; detected: 1; -> head: 6; (TD: 77798)
21: id: 3; detected: 1; -> head: 6; (TD: 81239)
22: id: 1; detected: 1; -> head: 6; (TD: 84788)
23: id: 9; detected: 1; -> head re-entry; head: 9; (TD: 88710)
24: id: 7; detected: 1; -> head: 9; (TD: 92452)
25: id: 9; detected: 1; -> head: 9; (TD: 96131)
26: id: 7; detected: 1; -> head: 9; (TD: 99669)
```

Listing 7.1 Step by step updates based on the specified ID sequence

Figure 2 presents a graphical depiction of the text based CFG output. Reverse arrows on the left side of the figure represent loops, while the reverse arrow on the right represents a reentry. The time taken at each MPI block is represented as its vertical length. The time of the compute blocks can be computed by subtracting the MPI times from the differential from preceding MPI operations. Their time is also represented by their vertical length in the figure. In summary, all necessary data is included so that such a graph can be computed by the local daemon from the serialized CFG data.

```
Current detected Control Flow Graph (CFG):
0x03; id: 2; H: 0; B: 0; Re: 0; Rv: 0; tail:      ; head:
0x2b; id: 0; H: 0; B: 0; Re: 0; Rv: 0; tail:      ; head:
0x31; id: 6; H: 1; B: 0; Re: 0; Rv: 4; tail: 0x3d; head:
0x37; id: 3; H: 0; B: 1; Re: 1; Rv: 5; tail: 0x55; head: 0x31
0x3d; id: 1; H: 0; B: 1; Re: 0; Rv: 5; tail:      ; head: 0x31
0x4f; id: 9; H: 1; B: 0; Re: 0; Rv: 3; tail: 0x55; head:
0x55; id: 7; H: 0; B: 1; Re: 0; Rv: 3; tail:      ; head: 0x4f
```

Listing 7.2 Example CFG detected based on the specified ID sequence

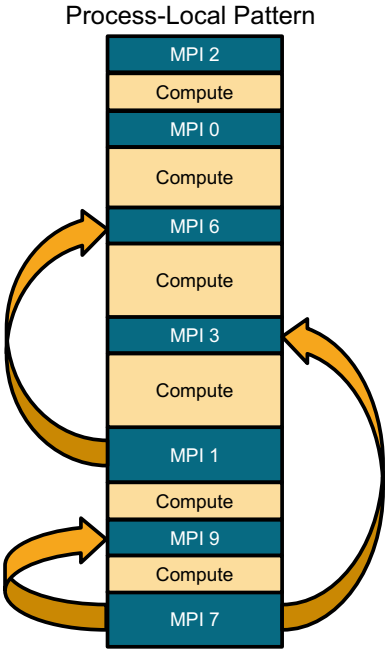

Fig. 2 Process-local Control Flow Graph (CFG) representation

3.1.2 Performance Measurements

The MPI library starts to record performance data once the heads and tails of one or more loops are detected. Currently two performance metrics are recorded:

1. Total Loop Time (TLT)
2. Total MPI Time (TMT)

The TLT metric is the total time spent on the detected loop. The TLT metric can be computed at each loop, including nested loops. The TLT metric is computed from two real numbers. The first one is its creation time. This time is set for each node in the CFG structure regardless of its type. The second one is the last visit time. The MPI library does not perform any more operations for this metric. Instead, the data is provided as it is to the local daemon once requested. The daemon is expected to perform the subtraction of these values for the total accumulated time, and to divide this value by the number of visits (revisits plus one) to get the average.

The second metric is the Total MPI Time (TMT). The TLT is inclusive of this time. This time is the difference between the entry and the exit times of each MPI call. In contrast to the TLT, these times are not stored in the CFG nodes where they are computed; instead, this metric is aggregated in the loop head of the node. There is no recursive search for the loop head in nested loops. The average can be computed by dividing the aggregated times by the total number of visits to their loop heads.

3.2 Node-Local Reductions and Performance Data Updates

Once a loop is detected, the library switches to a mode of CFG verification and performance data collection. As mentioned before, each process serializes its CFG data on its own shared memory segment. Each process notifies its local daemon on the following events:

- Loop detected
- Unexpected Loop exit
- Unexpected loop reentry

These events occur in the sequence presented in Listing 1: a loop detection occurs in steps 5 and 13, in step 11 an unexpected loop exit occurs, and in step 15 an unexpected loop reentry is encountered. All of these create changes in the CFG and therefore need to be communicated to the local daemon. These events tend to be more common during the initialization of MPI applications, and settle after a while. Expected loop reentries in the body or loop heads do not generate any events, since they do not trigger changes in the CFG. The library instead continues updating performance data without notifying its local daemon, if there are no changes to the CFG.

The number of notifications to the local daemon is limited by the sampling timer that currently defaults to one minute. This minimizes synchronization overheads, especially during the initialization of an application. If one or more loop detection or break events occur between timers, the local daemon is notified only once.

Performance data is updated separately from the CFG. These are updated periodically on each expiration of the sampling timer. These are only produced at the next loop head reentry, and not in any arbitrary MPI operation. Each metric specifies the identifier of its loop head, since more than one loop may be detected.

The local daemons do not read the performance data periodically. Instead, the latest data is read on demand when requests from the scheduler are received. These requests also have a field that optionally specifies a new value for the sampling timer. This enables the scheduler to adjust the frequency of data collection per application, based on previous performance data and trends.

3.2.1 Node-Local CFG Reduction

The daemon of a node keeps track of the notifications generated by each of its MPI processes. When any of its local processes have notified that their CFGs have been updated, it proceeds to read them and to perform a CFG reduction operation. The reduction operation depends on the order and type of the operations in it.

The following rules are followed on the collection of CFGs to produce a reduction:

1. Nodes outside of loops are ignored.
2. Consecutive point-to-point or one-sided operations are collapsed.
3. Identical loops are combined into one with a process range.

The reduced CFG is then stored in the memory of the local daemon. It is populated with performance data before it is sent to the scheduler on each request. If a request is received from the scheduler, but the CFG data is still unavailable, the response to the request has a field to indicates this.

An example set of four CFGs is presented in Fig. 3. All processes contain the loop from 6 to 1, but miss the nested loop with head 9 and tail 7. Rule 1 ignores the nodes 2 and 0. MPI operations with identifier 6 and 3 are of the type point-to-point. This means that they will be collapsed according to rule 2. All other operations are in loops. Finally, given rule 3, the loop from 6 to 1 will be clustered for ranks 0 through 3, while the loop from 9 to 7 will be separated for only rank 0. The information on its reentry is preserved. This indicates that it is nested within the common loop, but only at rank 0. The result is presented in Fig. 4.

The three rules in the reduction algorithm can be justified. The first rule is justified by the fact that code that occurs outside of loops is not relevant to elastic execution. The second rule comes from the observation that MPI applications that use multiple point-to-point and one-sided operations match logically across ranks. For example, it is common to observe branching based on the rank number of the local process in an MPI program to determine is the process will perform a send or a receive. These

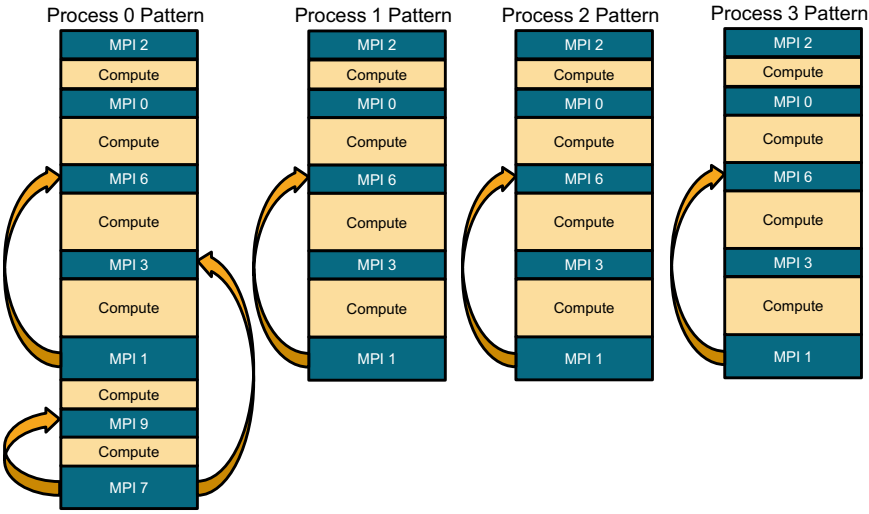

Fig. 3 Set of four CFGs at a node before reduction

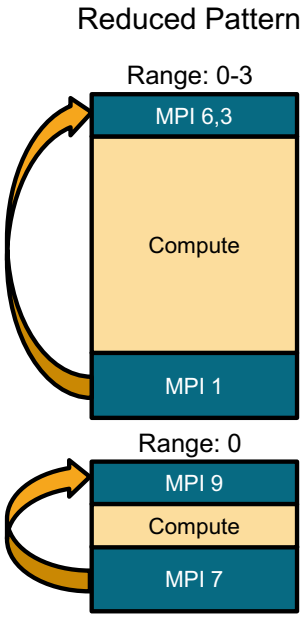

Fig. 4 Reduced CFG from Fig. 3

sends and receives can be matched as a single block of communication in a distributed view of the program, greatly simplifying the loop matching algorithm. This approach does not cover all possible patterns of point-to-point communication, and needs to be updated as the prototype matures. The final rule produces the reduction based on similarity. It is essentially a form of compression.

3.2.2 Node-Local Performance Data Reductions

The sum of all the TLT and TMT metrics of each process in a loop are added to the data of the reduced loop head nodes. In contrast, the mode (the value that occurs the most) of the loop revisit counts are set. It is expected that with enough revisits a small difference in the number of measurements will not affect the mean of the metrics significantly.

3.3 Distributed Reductions and Performance Models

The scheduler generates requests for performance data that reach all the daemons of an application. The requests and responses are routed through the `SRUN` binary of the application, over the Tree Based Overlay Network (TBON) that it creates with the nodes of its application. In the response to these request, each daemon sends the reduced CFGs with the TLT and TMT metrics attached to each loop head. The final distributed view of the CFG of the application is then generated from these at the scheduler.

Matching loops are reduced by combining all of their TLT and TMT metrics. The union of the process sets is set as the final range. The final distributed representation of the earlier example is presented in Fig. 5.

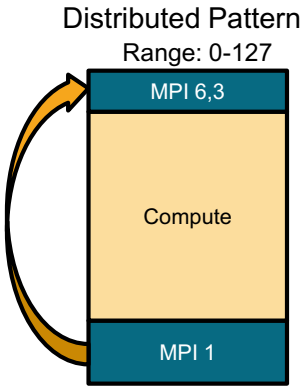

Fig. 5 Final reduced CFG at the scheduler from Fig. 4

Finally, the average loop time and MPI time metrics are computed based on the number of iterations of the loop heads and the TLT and TMT metrics provided. Additional memory is dedicated to store the mean, variance, minimum and maximum values of these final metrics. Finally, a vector of their recent values is stored, to detect performance trends.

3.3.1 SPMD-Phase Performance Model

Currently only one type of performance model has been implemented: the SPMD-Phase model. When the system detects one or more distributed loops, it creates an SPMD-Phase performance model instance for the application. Applications that do not fit this model (i.e., that have no distributed loop) are currently ignored. SPMD-Phase models consist of a set of distributed loops and their performance metadata. In general, models are used by the scheduling heuristic to try to ensure that application phases remain within their efficient range of resources.

4 Elastic Performance Feedback Overheads

A selection of resource manager operations is evaluated in this section. This selection contains all operations that impact the performance of MPI operations during normal computations. The operations that were not included are very numerous, but are either performed locally by one of the resource manager components, or do not impact the performance of preexisting MPI processes thanks to the latency hiding features of the design.

The evaluation has been performed in the SuperMUC [27] petascale system. This supercomputer is managed by the Leibniz Supercomputing Center (LRZ) and is located in Garching, Germany. The resources of this HPC system are managed by an IBM Load Leveler resource manager.

There were some challenges encountered when testing the custom resource manager and communication library. As may be expected, it is not possible to replace the preexisting resource manager. The new resource manager and MPI library were setup dynamically within a job. In that sense, the SLURM resource manager was nested inside of a Load Leveler job.

The SuperMUC system has multiple types of nodes divided in two sets: Phase 1 with Sandy Bridge CPUs, and Phase 2 with Haswell CPUs. Phase 1 nodes are based on a dual socket board with two Sandy Bridge-EP, Xeon E5-2680 CPUs. Each of these CPUs has 8 physical cores each, for a total of 16 per node, running at 2.7 GHz. Phase 2 nodes are also based on a dual socket board but with two Haswell-EP, Xeon E5-2697 CPUs. These have a higher CPU count of 14 physical cores each, for a total of 28 per node, running at lower 2.6 GHz.

All components (SLURM, MPICH and test applications) have been compiled with the GCC version 6 module provided in the SuperMUC system. The SuperMUC interconnect is based on Mellanox Infiniband network interfaces.

4.1 Tree Based Overlay Network (TBON) Latency

Resource adaptation instructions are set by the scheduler for each application when necessary. These adaptation instructions are probed by MPI applications periodically at locations where they can perform a redistribution of their domain. The communication between `SRUN` and the `SLURMD` daemons that manage the execution of an MPI application is important for the probe operation when the adaptation flag is set to true. The algorithm for probing has two sides: the side at each MPI process and the side at each `SLURMD` daemon. When the adaptation flag is set to true, multiple synchronization operations between the `SRUN` program and each daemon take place. These synchronization operations are performed over the Tree Based Overlay Network (TBON) that connects `SRUN` to each `SLURMD` daemon. Because of this, the latency of messages over the TBON can impact the overhead of MPI processes when they are required to adapt.

Figure 6 presents the latency of a single message and its confirmation from each participating node. In the figure, its scalability based on process count is presented. This means that the results for the Sandy Bridge and Haswell nodes will differ mainly due to the different core counts in the nodes. In the case of Haswell, only 20 nodes are needed to run 512 processes, while 32 nodes are needed in the Sandy Bridge nodes. As expected of a TBON network, the latency of messages scales logarithmically.

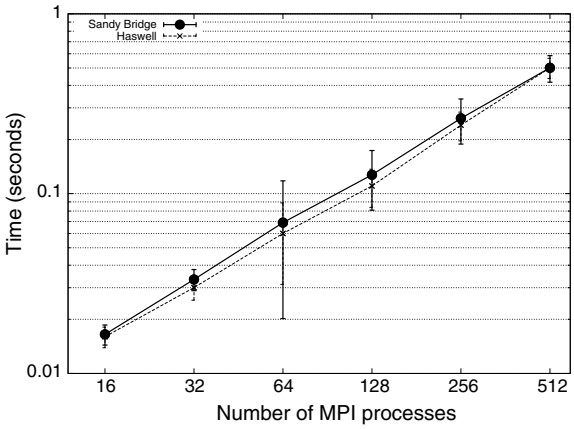

Fig. 6 Latency of TBON messages from `SRUN` to daemons

4.2 Control Flow Graph (CFG) Detection Overhead

In this section, the overhead of the set of operations that perform Control Flow Graph (CFG) detection is measured. Some of these operations impact the performance of MPI processes directly, while some can have a small impact since they are performed in the core where the `SLURMD` daemon of the node runs. These operations are: insertion, reduction, packing, unpacking and collapse.

The reduction, packing, unpacking and collapse operations are not as significant to the performance of MPI application processes due to their infrequent executions, as mentioned. That leaves the insertion operation as the only one that can impact the performance of application processes. The measurements are presented based on their scalability with respect to the size of the CFG graph, the total number of processes at each node, and finally the number of iterations of the loop in the application.

4.2.1 Scaling with Control Flow Graph (CFG) Size

It is important to understand how the detection overheads scale with increased CFG complexity. Figure 7 presents the scalability of all of the operations for CFG sizes between 8 and 1024 entries. Results for Phase 1 and Phase 2 nodes are included side by side for comparison. The sizes of CFGs are typically less than 100 entries, so the wide range of up to 1024 entries is pessimistic.

As mentioned before, the insertion latency is the most significant overhead. Unfortunately, the insertion latency scales exponentially with the number of entries in the CFG. Fortunately, although with bad scalability, the actual cost of the operation is small. A typical MPI operation runs for multiple milliseconds, while the insertion overhead is of around 700 nanoseconds for a 8 entry CFG, up to 10 μs for the extreme case of 1024 CFG entries. For the typical case of 128 CFG entries, the overhead of insertion is less than 2 μs.

The CFG reduction operation scales exponentially with the number of entries in the CFG. The overhead of 5 μs for 8 entries up to about 500 μs in the extreme 1024 entry case are acceptable, given the infrequency of this operation. The packing, unpacking and collapse operations scale exponentially, but their actual costs is much lower than the reduction operation, since these are performed in parallel with the participation of each MPI process. Their maximum cost of 100 μs at the extreme case of 1024 entries is also acceptable given the infrequency of these operations.

4.2.2 Scaling with Process Counts

In addition to scaling with the size of the CFG, it is also important to evaluate how the overheads scale with increasing numbers of processes at each node. These are intra-node operations, so only process counts that are expected to be possible, without oversubscription, in near future HPC nodes are considered: from 2 to 128 processes.

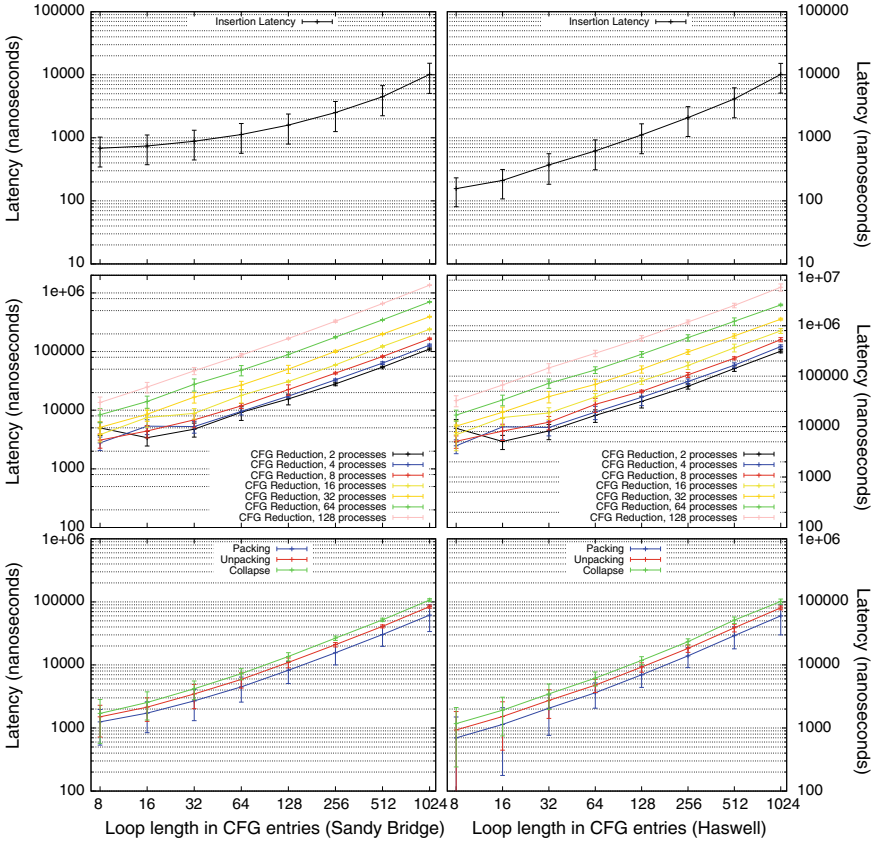

Fig. 7 CFG size performance scaling. Results for SuperMUC Phase 1 (Sandy Bridge, left) and Phase 2 (Haswell, right) are presented

Figure 8 presents scalability data for the detection operations based on process counts. Results for the larger CFG sizes 256, 512 and 1024 are presented for Phase 1 (left) and Phase 2 (right) nodes. As can be seen, the overheads for the insertion, packing, unpacking and collapse operations do not depend on the process counts, while the reduction operation does. Their latencies vary between a few hundred nanoseconds to a few hundred microseconds.

Not scaling with the number of processes is desirable, since it means that an arbitrary number of processes can be added at each node and these overheads will not increase. This is specially important in the case of the insertion latency, since this overhead is added to each MPI operation while the CFG detection mechanism is enabled. Once the CFG logic switches to verification, this overhead is removed. The packing, unpacking and collapse overheads are not as impactful to application performance, as mentioned before, since these occur infrequently.

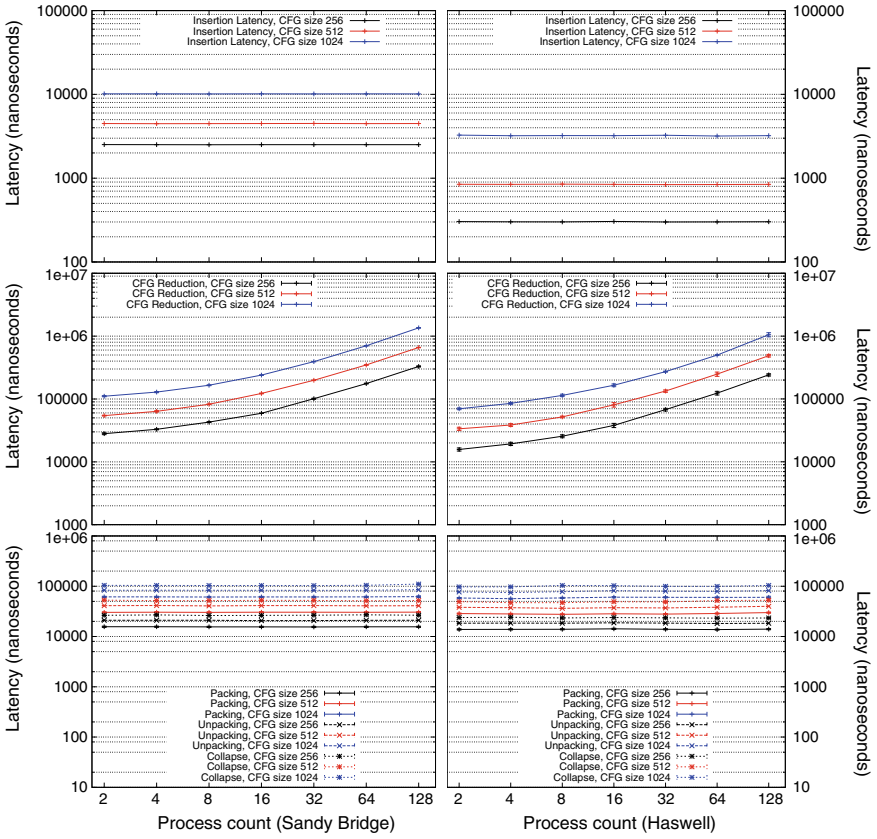

Fig. 8 Process count performance scaling. Results for SuperMUC Phase 1 (Sandy Bridge, left) and Phase 2 (Haswell, right) are presented

The situation for the reduction operation is not so fortunate, where its overhead increases with the number of processes per node of an application. As measured before, the overhead of this operation also increases with larger CFG sizes. Because of this, this operation has the worst scaling properties of the measurement infrastructure. Fortunately, these operations do not occur frequently and the absolute latency numbers it reaches are still not large.

4.3 MPI Performance Impact of the CFG Detection Overhead

Additional measurements were performed to evaluate the impact of these operations in actual MPI operations. MPI operations can run from a few microseconds to multiple seconds, depending on the type of operation, the number of processes and the size of the buffers.

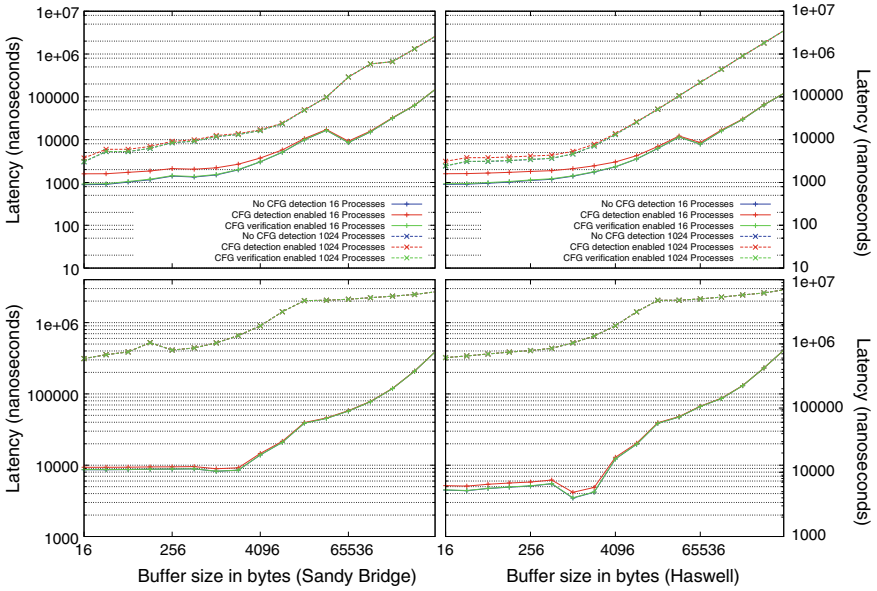

Fig. 9 `MPI_SEND` (top) and `MPI_BCAST` (bottom) performance examples with detection enabled and disabled on a 32 entry CFG loop. Results for SuperMUC Phase 1 (Sandy Bridge, left) and Phase 2 (Haswell, right) are presented

In Fig. 9, results for the `MPI_SEND` and `MPI_BCAST` operations are presented. These two operations were selected since they have the lowest latencies among the set of point-to-point and collective operations, respectively. The figure presents the latency for the `MPI_SEND` operation at the top and the `MPI_BCAST` operation at the bottom. Results for Phase 1 (left) and Phase 2 (right) nodes are presented side by side for comparison. Results for 16 and 1024 processes are presented with buffer sizes from 16 bytes up to a megabyte. The size of the CFG was set to 32 for these tests. Most applications and benchmarks that have been evaluated generate less CFG entries by the time they terminate.

As can be seen in the plots, the performance of `MPI_SEND` is only impacted significantly for message sizes of up to 4096 bytes, but only at lower process counts. For the case of 1024 processes, the overhead of the CFG detection algorithm is insignificant even for very small messages of 16 bytes. Additionally, the overhead of detection is not measurable on verification mode. This means that its overhead will only be observed when the detection algorithm has not encountered a loop, or when it exits a loop and resumes its detection.

A smaller performance impact can be observed for the `MPI_BCAST` operation. As mentioned before, the latency of this operation is the lowest among MPI collectives; therefore, the impact of CFG detection can be expected to be almost negligible when collectives are being used. Although the detection overhead is lower in terms of absolute latency, the percentage impact is higher in the case of Phase 2 nodes.

5 Conclusion

A CFG detection algorithm was implemented without the need of backtracing, in the MPI library. These CFGs are detected at each process and shared with the local resource manager daemons at compute nodes. These are eventually transferred to the scheduler running at a remote node through the TBON of the nodes allocated to each application. The overhead was shown to depend on the length of the CFG of applications. Because most applications produce CFGs that are in the order of hundreds of elements and the detection does not rely on backtracing, the overhead of detection was kept in the order of nanoseconds in most cases. The library switches to a verification only mechanism when a partial CFG remains stable. The overhead of verification cannot be measured even on single byte MPI messages with latencies in the order of microseconds.

A performance model can be produced with the data to drive scheduling decisions. It is expected that the integration of programming models and resource managers will increase in importance as exascale levels of performance are reached in HPC systems. Programming models that support resource-elastic execution and bring computational and energy efficiency benefits, while at the same time allowing for fault-tolerance, are expected to increase in importance in the near future. Performance feedback mechanisms, such as the one presented here, will allow future schedulers to make quality resource-scaling decisions to further improve system-wide efficiency metrics in HPC systems.

References

1. Aguilar, X., Fürlinger, K., Laure, E.: MPI trace compression using event flow graphs. In: Euro-Par 2014 Parallel Processing: 20th International Conference, Porto, Portugal, August 25–29, 2014. Proceedings, pp. 1–12. Springer International Publishing (2014). https://doi.org/10.1007/978-3-319-09873-91
2. Aguilar, X., Fürlinger, K., Laure, E.: Automatic on-line detection of MPI application structure with event flow graphs. In: Euro-Par 2015: Parallel Processing: 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings, pp. 70–81. Springer, Berlin, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48096-06
3. Aguilar, X., Fürlinger, K., Laure, E.: Visual MPI performance analysis using event flow graphs. Proced. Comput. Sci. **51**, 1353 – 1362 (2015). https://doi.org/10.1016/j.procs.2015.05.322. URL http://www.sciencedirect.com/science/article/pii/S1877050915011308
4. Aguilar, X., Fürlinger, K., Laure, E.: Event flow graphs for MPI performance monitoring and analysis. In: Tools for High Performance Computing 2015: Proceedings of the 9th International Workshop on Parallel Tools for High Performance Computing, September 2015, Dresden, Germany, pp. 103–115. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-39589-08
5. Casavant, T.L., Kuhl, J.G.: A taxonomy of scheduling in general-purpose distributed computing systems. IEEE Trans. Softw. Eng. **14**(2), 141–154 (1988). https://doi.org/10.1109/32.4634
6. Coffman, E.G., J., Garey, M.R., Johnson, D.S.: An application of bin-packing to multiprocessor scheduling. SIAM J. Comput. **7**(1), 1–17 (1978). https://doi.org/10.1137/0207001

7. Davis, R.I., Burns, A.: A survey of hard real-time scheduling for multiprocessor systems. ACM Comput. Surv. **43**(4), 35:1–35:44 (2011). https://doi.org/10.1145/1978802.1978814
8. Etsion, Y., Tsafrir, D.: A short survey of commercial cluster batch schedulers. Sch. Comput. Sci. Eng. Hebr. Univ. Jerus. **44221**, 2005–13 (2005)
9. Feitelson, D.G., Rudolph, L., Schwiegelshohn, U.: Parallel job scheduling—a status report. In: Proceedings of the 10th International Conference on Job Scheduling Strategies for Parallel Processing, JSSPP 2004, pp. 1–16. Springer, Berlin, Heidelberg (2005). https://doi.org/10.1007/114075221
10. Fortnow, L.: The status of the P versus NP problem. Commun. ACM **52**(9), 78–86 (2009). https://doi.org/10.1145/1562164.1562186
11. Fürlinger, K., Skinner, D.: Capturing and visualizing event flow graphs of MPI applications. In: Euro-Par 2009—Parallel Processing Workshops: HPPC, HeteroPar, PROPER, ROIA, UNICORE, VHPC, Delft, The Netherlands, August 25–28, 2009, Revised Selected Papers, pp. 218–227. Springer, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14122-526
12. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA (1990)
13. Graham, R., Lawler, E., Lenstra, J., Kan, A.: Optimization and approximation in deterministic sequencing and scheduling: a survey. In: Proceedings of the Advanced Research Institute on Discrete Optimization and Systems Applications, Annals of Discrete Mathematics, vol. 5, pp. 287–326. Elsevier (1979). https://doi.org/10.1016/S0167-5060(08)70356-X
14. Havlak, P.: Nesting of reducible and irreducible loops. ACM Trans. Program. Lang. Syst. **19**(4), 557–567 (1997). https://doi.org/10.1145/262004.262005
15. Ioannou, N., Kauschke, M., Gries, M., Cintra, M.: Phase-based application-driven hierarchical power management on the single-chip cloud computer. In: 2011 International Conference on Parallel Architectures and Compilation Techniques, pp. 131–142 (2011). https://doi.org/10.1109/PACT.2011.19
16. Jackson, D.B., Snell, Q., Clement, M.J.: Core algorithms of the Maui scheduler. In: Revised Papers from the 7th International Workshop on Job Scheduling Strategies for Parallel Processing, JSSPP 2001, pp. 87–102. Springer, London, UK (2001). http://dl.acm.org/citation.cfm?id=646382.689682
17. Karp, R.M.: Reducibility among combinatorial problems. In: Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations, pp. 85–103. Springer US, Boston, MA (1972). https://doi.org/10.1007/978-1-4684-2001-29
18. Khan, A.A., Mccreary, C.L., Jones, M.S.: A comparison of multiprocessor scheduling heuristics. In: Internatonal Conference on Parallel Processing Vol. 2, vol. 2, pp. 243–250 (1994). https://doi.org/10.1109/ICPP.1994.19
19. Lawler, E.L., Lenstra, J.K., Kan, A.H.R., Shmoys, D.B.: Chapter 9 sequencing and scheduling: Algorithms and complexity. In: Logistics of Production and Inventory, Handbooks in Operations Research and Management Science, vol. 4, pp. 445 – 522. Elsevier (1993). https://doi.org/10.1016/S0927-0507(05)80189-6
20. Lee, I., Iliopoulos, C.S., Park, K.: Linear time algorithm for the longest common repeat problem. J. Discret. Algorithms **5**(2), 243–249 (2007). https://doi.org/10.1016/j.jda.2006.03.019. 2004 Symposium on String Processing and Information Retrieval
21. Lenstra, J., Kan, A.R., Brucker, P.: Complexity of machine scheduling problems. In: Studies in Integer Programming, Annals of Discrete Mathematics, vol. 1, pp. 343–362. Elsevier (1977). https://doi.org/10.1016/S0167-5060(08)70743-X
22. Lopes, R.V., Menascé, D.: A taxonomy of job scheduling on distributed computing systems. IEEE Trans. Parallel Distrib. Syst. **27**(12), 3412–3428 (2016). https://doi.org/10.1109/TPDS.2016.2537821
23. Mu'alem, A.W., Feitelson, D.G.: Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. IEEE Trans. Parallel Distrib. Syst. **12**(6), 529–543 (2001). https://doi.org/10.1109/71.932708
24. Ramalingam, G.: Identifying loops in almost linear time. ACM Trans. Program. Lang. Syst. **21**(2), 175–188 (1999). https://doi.org/10.1145/316686.316687

25. Rotithor, H.G.: Taxonomy of dynamic task scheduling schemes in distributed computing systems. IEE Proc. Comput. Digital Techn. **141**(1), 1–10 (1994). https://doi.org/10.1049/ip-cdt:19949630
26. Srinivasan, S., Kettimuthu, R., Subramani, V., Sadayappan, P.: Selective reservation strategies for backfill job scheduling. In: Job Scheduling Strategies for Parallel Processing: 8th International Workshop, JSSPP 2002 Edinburgh, Scotland, UK, July 24, 2002 Revised Papers, pp. 55–71. Springer, Berlin, Heidelberg (2002). https://doi.org/10.1007/3-540-36180-44
27. SuperMUC Petascale System (2017). https://www.lrz.de/services/compute/supermuc/. [Online]
28. Tarjan, R.: Testing flow graph reducibility. In: Proceedings of the Fifth Annual ACM Symposium on Theory of Computing, STOC 1973, pp. 96–107. ACM, New York, NY, USA (1973). https://doi.org/10.1145/800125.804040
29. Transregional Research Center InvasIC (2017). http://www.invasic.de. [Online]
30. Ukkonen, E.: On-line construction of suffix trees. Algorithmica **14**(3), 249–260 (1995). 10.1007/BF01206331
31. Ullman, J.: Np-complete scheduling problems. J. Comput. Syst. Sci. **10**(3), 384–393 (1975). https://doi.org/10.1016/S0022-0000(75)80008-0
32. Wei, T., Mao, J., Zou, W., Chen, Y.: A new algorithm for identifying loops in decompilation. In: Static Analysis: 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007. Proceedings, pp. 170–183. Springer, Berlin, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74061-211