Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200063

Invasive Computing for Power Corridor Management

Jophin JOHN, Santiago NARVAEZ and Michael GERNDT
Technical University of Munich
Department of Informatics
85748 Garching, Germany
E-mail: {john, narvaez, gerndt}@in.tum.de

Abstract. This paper investigates the use of invasive computing to enforce the power budget in an HPC infrastructure. Invasive MPI along with the Invasive Resource Manager (IRM) provides an infrastructure for developing malleable/invasive applications. In IRM, a power model is used to predict the power consumption of each application. If a violation in power corridor is predicted, IRM reconfigures the node allocation among the applications to keep the whole system back into the power corridor. Since development of invasive applications is a complex task, a new programming model called Elastic Phase Oriented Programming (EPOP) is developed to simplify the invasive programming. This model is also capable of collecting and sharing power usage metrics as well as performance metrics to IRM.

Keywords. dynamic resource management, power corridor enforcement, MPI, High Performance Computing, Slurm Batch Scheduler

1. Introduction

Current contracts between energy companies and compute centers are written in accordance to the so called power corridor. Therefore, the power consumption must be bounded by certain upper and lower limits. If the compute center goes beyond those limits (i.e., if consumes less or more than what it is stipulated in the contract) some fines could be applied by the energy company. The compute center can act as well as a power stabilizer for the grid load [1]. This means that dynamic adaptions of the power corridor might be part of the electricity contract, and could be requested by the electricity company. The compute center will have economic incentives for doing so, decreasing the electricity costs. To enforce the upper limit it is possible to use well-known techniques such as power capping; nevertheless these cannot be used to enforce the lower limit (i.e., to increase the system power consumption). In this work we show how a new paradigm for parallel computing, namely invasive computing, can be used for such case.

Invasive computing is a paradigm introduced by Teich [2]. A program that follows this paradigm, called henceforth ”invasive program”, should be able to request, use and finally free processing, communication, and memory resources in the neighborhood of its computing environment.

An invasive program is by definition malleable. This in turn means that certain optimizations, which would be otherwise hindered, are now possible. A nice example that

comes to hand is MPI, which usually has a very static view of the application. All the tasks that are created at the beginning of the application run when the batch scheduler has distributed the resources. The resources are exclusively reserved for this job and the tasks continue running until the end of it. In contrast, the resources assigned to an invasive program could change at runtime.

Within the Transregio Special Research Centre Invasic (TRR89), TUM investigates invasive resource management for HPC systems. We developed an MPI extension called iMPI [3,4] which allows through three new MPI functions writing invasive MPI applications. In combination with an extension of the Slurm batch scheduler, nodes can be dynamically redistributed among running MPI applications. This enables reducing idle nodes by more flexible scheduling, increased energy efficiency by redistributing nodes according to application efficiency, and supporting novel, dynamic applications, such as a Tsunami simulation that can use the resources more efficiently.

Development of invasive applications presents certain challenges. For example, the developer must take care of defining where an adaptation is possible, handling the newly joining processes, redistribution of data, among others. This results in a complex control flow which makes the development of invasive application difficult.

This paper reports on our two contributions. First contribution is a high level programming model on top of iMPI called EPOP (Elastic Phase Oriented Programming model) that simplifies the programming of iMPI applications by providing explicit control flow between elastic and rigid program phases. Second contribution is a power corridor management infrastructure using extensions we made to iMPI, IRM and EPOP system to collect power measurements, compute a power model and use it to keep the system inside the power corridor by redistributing resources. The presented work is based on an early prototype developed in [5,6] which was consolidated and extended.

This paper is divided into 6 sections. Section 2 will give an overview about the related work. Section 3 acts as an introduction to create invasive applications using our infrastructure. Section 4 explains in detail how the power corridor mangement was implemented followed by section 5 which presents the evaluation of the infrastructure. Finally, section 6 presents the conclusions.

2. Related work

EPOP is a programming model that provides malleability to MPI applications using the invasive infrastructure provided by iMPI and IRM. Charm++ and Adaptive Message Passing Interface (AMPI) [7] also supports the malleability of jobs by checkpoint restart along with the task migration and dynamic load balancing. AMPI abstracts the MPI processes as migratable threads and the runtime system of Charm++ deals with the scheduling and migration of these threads. Standard MPI is extended to support the Charm++ runtime system. AMPI follows a message-driven execution model and there is oversubscription due to the threading. In contrast, EPOP is based on the invasive properties of the iMPI and uses the standard MPI execution model with no oversubscription. EPOP can also provide application specific profiling information like the node level power usage and mpi time to IRM.

There are several techniques employed by supercomputing centers to control the system-wide power consumption. One such notable technique is dynamically shutting

down the jobs when a power budget is reached [8]. There is an approach where scheduler decides the future job allocation based on an application's power efficiency in the past runs. Another technique is the usage of Intelligent energy-aware backfilling algorithms along with stoppage of a node to control the total power usage [9]. In some techniques, idle nodes are selectively powered down to meet the power requirements [10]. Another power management approach is to utilize the power capping mechanisms supported by the hardware as well as forcing a system to operate at specified frequencies. There are plenty of researches [11] focussing on power capping as well as dynamic frequency scaling techniques to bring down the power usage of the system. One such approach is using CPU and memory Dynamic Voltage and Frequency Scaling (DVFS) for system-wide power capping [12]. One important difference between our work and these techniques is that ours use invasive computing for dynamic power corridor management. Also, most of these systems are using a reactive approach, which means that they only act once the system is out of the power corridor. In contrast, we use a proactive approach where resource adaptations are performed based on the power usage predictions. Additionally, our system can also handle dynamic power budget requirements.

3. Programming Invasive MPI Applications

Invasive applications can be developed using an invasive infrastructure, which in this case is constituted by the Invasive Resource Manager (IRM) along with the Invasive MPI (iMPI). IRM provides dynamic resource management and iMPI provides routines to utilize this dynamism.

IRM is an extension of the Simple Linux Utility for Resource Management (Slurm) [13]. IRM decides to expand/shrink an application based on its performance. IRM informs iMPI of the decision. iMPI[4] is an extension to MPICH [14], where the following new operations have been added to bring dynamism:

MPI_Init_adapt(...) signals the resource manager that the application will be adaptive.

MPI_Probe_adapt(...) is used to check whether there are any resource changes.

MPI_Comm_adapt_begin(...) is called to begin the adaptation window.

MPI_Comm_adapt_commit() finalizes resource adaptation.

Pseudocode for creating an iMPI application is shown in Listing 1. In the beginning, `MPI_Init_adapt()` is used to signal IRM that the application is invasive. It is also used to distinguish whether a process was created as part of a resource change or was it created at the start of application (Listing 1, lines 4-7). This is essential since `MPI_Comm_adapt_begin()` should immediately be called by the newly joining process to start the adaptation(Listing 1, lines 8-11). This call will notify IRM that the newly created processes are ready and IRM then notifies the existing processes about resource redistribution.

```
1  ...
2  MPI_Init_adapt(..., mytype)
3  // Initialization block
4  if mytype = starting_process{
5   set phase_index = 0
6  }
7  else{// Newly joining processes
8   MPI_Comm_adapt_begin(...);
9     // Redistribute data
10  MPI_Comm_adapt_commit( );
11 }
12 // Begin elastic block 1
13 if(phase_index == 0){
14  while ( block_condition ){
15   MPI_Probe_adapt(...)
16   if resource_change {
17    MPI_Comm_adapt_begin(...)
18    // Redistribute data
19    MPI_Comm_adapt_commit( )
20   }
21   iteration_number++;
22 // Compute Intensive part
23 }
24  phase_index++;
25 }
26 // End elastic block 1
27 ...
28 // Begin elastic block n
29 if(phase_index == n){
30 ...
31 }
32 // End elastic block n
33 // Finalization block
34 ...
```

Listing 1: Pseudocode of a simple iMPI program

```
1  ...
2  void init_block(...){
3  // Code in initialization block
4  }
5  setInit(init_block);
6
7
8
9  void elastic_block_1(...){
10 /* Compute intensive part of
11  elastic block 1*/
12 }
13 bool block_condition (...){
14 // Looping of elastic_block
15 }
16 void resource_change(...){
17 // Redistribute data
18 }
19 setElastic(elastic_block_1,
20              block_condition,
21                resource_change);
22
23
24 ...
25
26 setElastic(elastic_block_n,...);
27
28 ...
29
30 void finalize_block(...){
31 // Code in finalization block
32 }
33 setRigid(finalize_block,...);
34 ...
```

Listing 2: Pseudocode in Listing 1 as an EPOP program

Meanwhile, the existing processes should frequently check for the resource change using `MPI_Probe_adapt()` during the computation. In case of a resource change, `MPI_Comm_adapt_begin()` is called in order to take part in the adaptation (Listing 1, lines 15-19).

Once all the processes are at the adaptation window, the entry point, required data, etc. can be distributed (Listing 1, line 9 and 18) among new processes. Entry point refers to the application region where the new processes can safely join the existing processes. In the lines 13 and 29 of Listing 1, `phase_index` is used to identify these phases/entry points. As seen in Listing 1, the application is logically divided into different elastic blocks (parts of code where resource redistribution is possible) for creating suitable entry points for the joining processes. `MPI_Comm_adapt_commit()` is then called to finalize

the adaptation. After this point, all the processes continue the computation.

One of the issues with such an invasive application is the multiple control flows. As seen from Listing 1, the pre-existing processes should identify the entry points, probe for resource changes, enter the adaptation window, redistribute the data and entry points, etc while the newly joining processes should immediately enter the adaptation window and wait for the entry points, data, etc. This complicates the invasive application development.

The Elastic Phase Oriented Programming model (EPOP) simplifies this application development process by providing the concept of "Phases" to mark different parts of an application. A simple invasive application can have three logical parts/phases: an initialization part, a compute intensive part that can be benefited from the resource adaptation and a finalization part to write the results. EPOP provides different "Phases" to represent these parts. They are:

Init phase : to represent initialization part of an application.

Elastic phase : to represent compute intensive part of the application that can benefit from resource adaptation.

Rigid phase : to represent parts of an application that does not need resource adaptation.

Branch phase : to switch between different phases.

A simple EPOP version of the application in Listing 1 is shown in Listing 2. The EPOP driver, which is in charge of control flow in EPOP applications, will call iMPI routines in the background (not shown in the listing) to make it invasive.

Elastic block in Listing 1 (lines 13-26) contains a looping construct (line 14) that determines how many times the main compute part (line 22) will be called. It also contains a resource change probing part (line 15), which checks for a resource change and does data distribution, and an entry point transfer in case of a resource change (lines 16-20). These parts can be represented as a collection of simple functions like in Listing 2 (lines 14-26) and can be marked as an elastic phase using `setElastic(...)`. EPOP will probe for resource change and will call the `resource_change` function corresponding to the elastic phase whenever there is a resource redistribution. Whenever the newly joining processes are available, EPOP will bring it into the `resource_change` function of the current elastic phase. As a result, `phase_index` used in lines 13 and 29 of Listing 1 is not needed in EPOP. The phases are executed in the same order as they are declared (In Listing 2, lines 5,19,25 and 33 will declare phases).

EPOP and iMPI only provide methods to simplify the addition/removal of processes into/from an application. In addition EPOP will also bring all the processes to a common entry point specified by the developer. During resource change, a developer is responsible for maintaining the topological properties of the application (for example; create a new topology with a new number of processes) as well as redistributing the data among existing and joining/leaving processes. This design decision was made because each application has its own data distribution, which might be based on number of processes, threads, and other things known by the developer. For iMPI/EPOP application, users can redistribute the data among all processes during the adaptation window and hence after adaptation every process has the required data to do the computation.

4. Power corridor management

4.1. Measurements

Since we were using proactive approach for power management, we needed to predict whether the system will go out of power corridor. This prediction is done using the time series analysis techniques described in Section 4.2 which in turn require previous power measurements. On Intel systems, such as the one used for the testing, energy consumption estimations are done through the Running Average Power Limit (RAPL) sensors [15], and these values can be accessed via the Model Specific Registers (MSR). The power can then be derived by dividing this value by the time between measurements. There are multiple libraries that can be used to access these registers. For this case, we chose to use LIKWID [16].

The infrastructure was deployed on a job allocation in SuperMUC [17]. There was no cluster level measurement infrastructure available to us, and thus using RAPL was the only possible way to obtain power measurements. This also meant that we had to focus on the power consumed on the nodes, omitting the cooling system, networking components, storage system, etc. Nevertheless, measuring only the power consumed by the nodes is still enough to demonstrate the effectiveness of using resource redistribution as a power corridor management technique. Additionally, if a cluster wide tool becomes available, then the input power values can be taken from it instead of RAPL.

IRM communicates to the EPOP driver the frequency and number of measurements to be taken. Next, one rank per node will create a thread in charge of taking power measurements. Once it has accumulated the required number, they are aggregated by the leading node, and then sent to the scheduler. At this point, IRM receives the measurements and stores them. Once it has enough data, the forecast module comes into play. The main purpose of this module is to predict the future maximum and minimum power consumption of the system. They represent the worst case scenarios, i.e., the cases where the system could go out of the power corridor.

4.2. Forecasting

Using time series analysis one can try to find an underlying structure of some data, such as power consumption values. Two things are required: 1. a valid time series to work with. 2. A specific method to analyse the data. For the latter, we have chosen to use three techniques, the AutoRegressive Integrated Moving Average (ARIMA), Seasonal ARIMA with exogenous regressors (SARIMAX) and the Holt-Winters method.

ARIMA is a ”classical model”, in the sense that it has been studied extensively. It is composed of an Integrated component, which is in charge of making data stationary, and an ARMA component, which models this stationary data. The latter can again be subdivided into an AutoRegressive component (AR), which captures the relation between the current value of the time series and some of its past values, and a Moving Average component (MA) that represents the influence of an often unexplained random shock. Using both of them, plus the Integrated component, one can derive the ARIMA model. SARIMAX is an extension of ARIMA that supports time series with a seasonal component. The third method used in this work, called Holt-Winters or Third Exponential Smoothing, assigns exponentially decreasing weights to past observations. It is capable

of model data with both trend and seasonality, distinguishing for the latter the case of additive and multiplicative seasonality.

4.3. Decision Making

Every time the controller predicts that the system might go outside the power corridor, it is necessary to redistribute the nodes to try to prevent it. This problem is expressed more formally in Equation 1, where we have K applications running on a system with N nodes ($K \leq N$). We assume that every idle node consumes p_{idle} power. The idea is to minimize the power consumed $f(k_{idle})$ by the idle nodes, such that both the upper U and lower L boundaries are fulfilled.

$$
\begin{aligned}
&MINIMIZE \\
&\quad f(k_{idle}) = k_{idle} * p_{idle} \\
&SUBJECT\ TO \\
&\quad l \leq \sum_{i=0}^{K-1} k_i * p_{min}^{(i)} + k_{idle} * p_{idle} \\
&\quad u \geq \sum_{i=0}^{K-1} k_i * p_{max}^{(i)} + k_{idle} * p_{idle} \\
&\quad 1 \leq k_i \leq N,\ k_i \in \mathbb{N} \setminus \{0\},\ i = 0, \cdots, K-1 \\
&\quad 0 \leq k_{idle} < N,\ k_{idle} \in \mathbb{N}
\end{aligned}
\tag{1}
$$

The solution to the system is found via Pulp, a Python Integer Programming Solver module [18]. In turn, Pulp acts as an interface to several solvers. In this case we have used Coin-or Branch and Cut (CBC). We tested Pulp with systems with $K = 2, 4, 8$ and 16, and the solving time was always under 0.5 seconds. Considering that a decision has to be made from one schedule pass to the next, Pulp is fast enough. Once a valid node distribution is found, an adaptation occurs (see Section 3).

4.4. Guarantees

In an infrastructure with a size of N Nodes, where K applications are running:

Our system will enforce upper power corridor U, if and only if the power consumption of the system when each application runs in only one node is less than the power corridor upper bound U, as expressed in Equation 2.

$$
U \geq \sum_{i=1}^{K} p_{max}^{(i)} + (N - K) * p_{idle} \tag{2}
$$

Similarly, our system will enforce lower power corridor L, if and only if, the power consumption of the system is greater than the lower power corridor boundary when the

most power consuming application A is running on $N-(K-1)$ nodes. This is shown in Equation 3.

$$L \leq \sum_{i=1}^{K-1} p_{max}^{(i)} + (N-(K-1)) * k_A * p_A \tag{3}$$

5. Evaluation

The power corridor management infrastructure was run as a standard Load Leveler job in SuperMUC Phase 2 [17]. The infrastructure was run on 32 nodes (896 processes) for the forecasting and upper power corridor enforcement tests while the remaining tests were performed on 16 nodes (448 processes). Three EPOP applications (2D Jacobi heat simulation, LU decomposition and Pi calculation) were used to evaluate the infrastructure. This evaluation is a proof-of-concept that the dynamic resource management can be used for enforcing the power corridor on a system with varying power constraints.

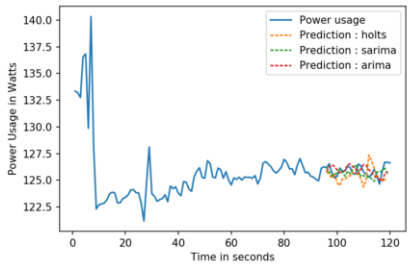
5.1. Forecasting

Power consumption predictions from three different models are shown in Figure 1a. These models were trained on-the-fly and the one with the highest accuracy was chosen by IRM for forecasting. As observed in Figure 1a, the SARIMA model produced more accurate predictions among the different models and was used by IRM to make the scheduling decisions. Accuracy of the model was determined using the Mean Absolute Percentage Error (MAPE).

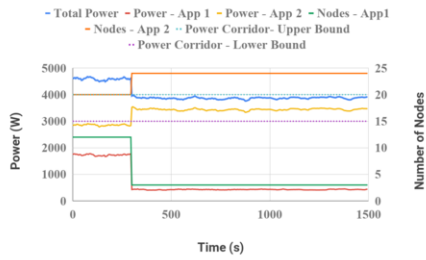
5.2. Upper and Lower Power Corridor Enforcement

Figure 1b and 1c shows the effect of dynamic resource adaptation on system wide power usage. Initially, two EPOP applications were started with 12 nodes for Application 1 and 20 for Application 2. The power corridor was set between 3000 and 4000 Watts. It can be seen from Figure 1b that the upper power bound has already been violated from the start of the applications. Therefore, during the first scheduler pass, the system redistributes the number of nodes. As a result, application 1 was reduced to 3 nodes and application 2 was expanded to 24 nodes. This lead to the reduction of power usage, bringing back the system to the power corridor after 300 seconds. During the next scheduler passes, the forecast module predicts no violation of the power corridor and as a result, the resource configuration remained same.

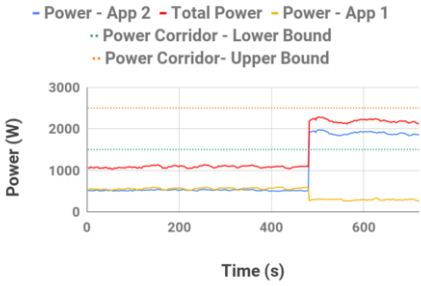
Similarly, the lower power corridor enforcement is shown in Figure 1c. The power corridor was set between 1500 and 2500 watts. Two applications (Jacobi heat simulation and LU decomposition) were started with 4 nodes each. It can be observed that the lower power corridor of the system was violated from the beginning. To enforce the power corridor, IRM shrunk Application 1 to 2 nodes and expanded Application 2 to 14 nodes during the first scheduler pass. As a result, the system was back in the power corridor.



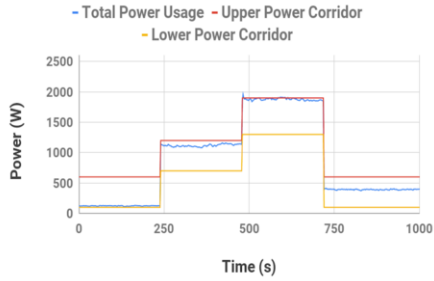
(a) Forecasting the power usage



(b) Upper power corridor enforcement



(c) Lower power corridor enforcement



(d) Dynamic power corridor enforcement

Figure 1. Power Corridor Management

5.3. Dynamic Power Corridor Enforcement

Dynamic power corridor enforcement is shown in Figure 1d. Initially, the power corridor was set between 100 and 600 watts. An invasive Pi calculation application was started on a single node. It can be observed that the system is in the power corridor. Then the power corridor was shifted to 700 and 1200 watts. IRM expanded the application to 9 nodes and brought back the system into the power corridor. The power corridor was then increased to 1300 and 1900 watts. We can observe from Figure 1d that IRM again redistributed the resources to bring the system back in the power corridor. This test simulates dynamically changing power constraints and how the system is responding to it.

6. Conclusion and Outlook

Power corridor management is crucial for supercomputing centers. As more and more renewable energy sources are used for power generation, HPC centers must be flexible in adapting to the energy requirements, since the supply of renewable energy will be varying due to external factors. Resource dynamism and flexible scheduling can be used to accommodate such dynamic scenarios. We have shown in this paper that invasive computing can be used as a mechanism to enforce the power corridor. We were able to regulate power consumption without taking drastic measures, such as killing power-hungry applications. One of the shortcomings of this invasive approach is that frequent

resource redistributions can be expensive. Our ongoing work, a hybrid system which can use DVFS along with invasive computing to manage power requirements, addresses this shortcoming.

Acknowledgements

This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing (SFB/TR 89).

References

[1] H. Chen, M. C. Caramanis, and A. K. Coskun, "The data center as a grid load stabilizer," in *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2014, pp. 105–112.

[2] J. Teich, "Invasive algorithms and architectures invasive algorithmen und architekturen," *it-Information Technology*, vol. 50, no. 5, pp. 300–310, 2008.

[3] M. G. I. Compres Urena, "Towards elastic resource management," in *Proceedings of the 11th Parallel Tools Workshop, September 11-12, 2017*, 2017, to appear.

[4] I. A. Comprés Ureña, "Resource-elasticity support for distributed memory hpc applications," Dissertation, TU Munich, Munich, 2017. [Online]. Available: http://mediatum.ub.tum.de?id=1362721

[5] J. John, "The elastic phase oriented programming model for elastic hpc applications," Master Thesis, TU Munich, Munich, 2018. [Online]. Available: https://mediatum.ub.tum.de?id=1475008

[6] S. Narvaez, "Power model for resource-elastic applications," Master Thesis, TU Munich, Munich, 2018. [Online]. Available: http://mediatum.ub.tum.de?id=1475095

[7] L. V. Kale and S. Krishnan, "Charm++: A portable concurrent object oriented system based on c++," in *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, 1993.

[8] M. Maiterth, G. Koenig, K. Pedretti, S. Jana, N. Bates, A. Borghesi, D. Montoya, A. Bartolini, and M. Puzovic, "Energy and power aware job scheduling and resource management: Global survey — initial analysis," 05 2018, pp. 685–693.

[9] P. Dutot, Y. Georgiou, D. Glesser, L. Lefevre, M. Poquet, and I. Rais, "Towards energy budget control in hpc," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, May 2017, pp. 381–390.

[10] J. Sun, C. Huang, and J. Dong, "Research on power-aware scheduling for high-performance computing system," in *2011 IEEE/ACM International Conference on Green Computing and Communications*, Aug 2011, pp. 75–78.

[11] M. Yadav, "A brief survey of current power limiting strategies," 03 2018.

[12] Y. Liu, G. Cox, Q. Deng, S. C. Draper, and R. Bianchini, "Fastcap: An efficient and fair algorithm for power capping in many-core systems," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2016, pp. 57–68.

[13] "Simple linux utility for resource management," http://slurm.schedmd.com/, accessed: 2019-07-09.

[14] MPICH, "Mpich," https://www.mpich.org/.

[15] K. N. Khan, M. Hirki, T. Niemi, J. K. Nurminen, and Z. Ou, "Rapl in action: Experiences in using rapl for power measurements," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, vol. 3, no. 2, p. 9, 2018.

[16] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *2010 39th International Conference on Parallel Processing Workshops*. IEEE, 2010, pp. 207–216.

[17] Leibniz-Rechenzentrum, "Supermuc petascale system," https://www.lrz.de/services/compute/supermuc/.

[18] S. Mitchell, M. OSullivan, and I. Dunning, "Pulp: a linear programming toolkit for python," *The University of Auckland, Auckland, New Zealand*, 2011.