Do moldable applications perform better on failure-prone HPC platforms?

Valentin Le Fèvre[1], George Bosilca[2], Aurelien Bouteiller[2], Thomas Herault[2], Atsushi Hori[3], Yves Robert[1,2], Jack Dongarra[2,4]

[1] Laboratoire LIP, École Normale Supérieure de Lyon & Inria, France
[2] University of Tennessee, Knoxville TN, USA
[3] RIKEN Center for Computational Science, Japan
[4] University of Manchester, UK
{valentin.le-fevre,yves.robert}@inria.fr,
{bosilca,bouteill,herault,dongarra}@icl.utk.edu, ahori@riken.jp

Abstract This paper compares the performance of different approaches to tolerate failures using checkpoint/restart when executed on large-scale failure-prone platforms. We study (i) RIGID applications, which use a constant number of processors throughout execution; (ii) MOLDABLE applications, which can use a different number of processors after each restart following a fail-stop error; and (iii) GRIDSHAPED applications, which are moldable applications restricted to use rectangular processor grids (such as many dense linear algebra kernels). For each application type, we compute the optimal number of failures to tolerate before relinquishing the current allocation and waiting until a new resource can be allocated, and we determine the optimal yield that can be achieved. We instantiate our performance model with a realistic applicative scenario and make it publicly available for further usage.

1 Introduction

Consider a long-running job that requests N processors from the batch scheduler. Resilience to fail-stop errors[5] is provided by a Checkpoint/Restart (CR) mechanism, which is the de-facto standard approach for High-Performance Computing (HPC) applications. After each failure, the application restarts from the last checkpoint but the number of available processors decreases, assuming the application can continue execution after a failure (e.g., using ULFM [3]). Until which point should the execution proceed before requesting a new allocation with N fresh resources from the batch scheduler?

The answer depends upon the nature of the application. For a RIGID application, the number of processors must remain constant throughout the execution. The question is then to decide the number F of processors (out of the N available initially) that will be used as spares. With F spares, the application can tolerate F failures. The application always executes with $N - F$ processors: after each

[5] We use the terms *fail-stop error* and *failure* indifferently.

failure, then it restarts from the last checkpoint and continues executing with $N - F$ processors, the faulty processor having been replaced by a spare. After F failures, the application stops when the $(F+1)$st failure strikes, and relinquishes the current allocation. It then asks for a new allocation with N processors, which takes a *wait time*, D, to start (as other applications are most likely using the platform concurrently). The optimal value of F obviously depends on the value of D, in addition to the application and resilience parameters. The wait time typically ranges from several hours to several days if the platform is over-subscribed (up to 10 days for large applications on the K-computer [24]). The metric to optimize here is the (expected) application yield, which is the fraction of useful work per second, averaged over the N resources, and computed in steady-state mode (expected value for multiple batch allocations of N resources).

For a MOLDABLE application, the problem is different: here we assume that the application can use a different number of processors after each restart. The application starts executing with N processors; after the first failure, the application recovers from the last checkpoint and is able to continue with only $N-1$ processors, albeit with a slowdown factor $\frac{N-1}{N}$. After how many failures F should the application decide to stop[6] and accept to produce no progress during D, in order to request a new allocation? Again, the metric to optimize is the application yield.

Finally, consider an application which must have a given shape (or a set of given shapes) in terms of processor layout. Typically, these shapes are dictated by the algorithm. In this paper, we use the example of a GRIDSHAPED application, which is required to execute on a rectangular processor grid whose size can dynamically be chosen. Most dense linear algebra kernels (matrix multiplication, LU, Cholesky and QR factorizations) are GRIDSHAPED applications, and perform more efficiently on square processor grids than on elongated rectangle ones. The application starts with a square $p \times p$ grid of $N = p^2$ processors. After the first failure, execution continues on a $p \times (p-1)$ rectangular grid, keeping $p-1$ processors as spares for the next $p-1$ failures. After p failures, the grid is shrunk again to a $(p-1) \times (p-1)$ square grid, and so on. We address the same question: after how many failures F should the application stop working on a smaller processor grid and request a new allocation, in order to optimize the application yield?

The major contribution of this paper is to present a detailed performance model and to provide analytical formulas for the expected yield of each application type. Due to lack of space, we instantiate the model for a single applicative scenarios, for which we draw comparisons across application types. Our model is publicly available [21] so that more scenarios can be explored. Notably, the paper qualifies the optimal number of spares for the optimal yield, and the optimal length of a period between two full restarts; it also qualifies how much the

[6] Another limit is induced by the total application memory Mem_{tot}. There must remain at least ℓ live processors such that $Mem_{tot} \leq \ell \times Mem_{ind}$, where Mem_{ind} is the memory of each processor. We ignore this contraint in the paper but it would be straightforward to take it into account.

yield and total work done within a period are improved by deploying MOLDABLE applications w.r.t. RIGID applications.

The rest of the paper is organized as follows. Section 2 provides an overview of related work. Section 3 is devoted to formally defining the performance model. Section 4 provides formulas for the yield of RIGID, MOLDABLE and GRIDSHAPED applications. These formulas are instantiated through the applicative scenario in Section 5, to compare the different results. Finally, Section 6 provides final remarks and hints for future work.

2 Related work

We first survey related work on checkpoint-restart. Then we discuss previous contributions on MOLDABLE applications.

Checkpoint-restart. Checkpoint/restart (CR) is the most common strategy employed to protect applications from underlying faults and failures on HPC platforms. Generally, CR periodically outputs snapshots (*i.e.*, checkpoints) of the application global, distributed state to some stable storage device. When a failure occurs, the last stored checkpoint is retrieved and used to restart the application.

A widely-used approach for HPC applications is to use a fixed checkpoint period (typically one or a few hours), but it is sub-optimal. Instead, application-specific metrics can (and should) be used to determine the optimal checkpoint period. The well-known Young/Daly formula [25,8] yields an application optimal checkpoint period, $\sqrt{2\mu C}$ seconds, where C is the time to commit a checkpoint and μ the application Mean Time Between Failures (MTBF) on the platform. We have $\mu = \frac{\mu_{ind}}{N}$, where N is the number of processors enrolled by the application and μ_{ind} is the MTBF of an individual processor [17].

The Young/Daly formula minimizes platform waste, defined as the fraction of job execution time that does not contribute to its progress. The two sources of waste are the time spent taking checkpoints (which motivates longer checkpoint periods) and the time needed to recover and re-execute after each failure (which motivates shorter checkpoint periods). The Young/Daly period achieves the optimal trade-off between these sources to minimize the total waste.

For RIGID applications, both [18,26] report some experimental study to determine the optimal number of processors and of spares that should be used. Furthermore, the optimal number of resources for a perfectly parallel job is computed via an iterative relaxation procedure in [18] and through analytical formulas in [5].

Moldable and GridShaped applications RIGID and MOLDABLE applications have been studied for long in the context of scientific applications. A detailed survey on various application types (RIGID, MOLDABLE, malleable) was conducted in [10]. Resizing application to improve performance has been investigated by many authors, including [19,6,23,22] among others. A related recent

study is the design of a MPI prototype for enabling tolerance in MOLDABLE MapReduce applications [13].

The TORQUE/Maui scheduler has been extended to support evolving, malleable, and MOLDABLE parallel jobs [20]. In addition, the scheduler may have system-wide spare nodes to replace failed nodes. In contrast, our scheme does not assume a change of behavior from the batch schedulers and resource allocators, but utilizes job-wide spare nodes: a node set including potential spare nodes is allocated and dedicated to a job at the time of scheduling, that can be used by the application to restart within the same job after a failure.

An experimental validation of the feasibility of shrinking application on the fly is provided in [2]. In this paper, the authors used an iterative solver application to compare two recovery strategies, shrinking and spare node substitution. They use ULFM, the fault-tolerant extension of MPI that offers the possibiliity of dynamically resizing the execution after a failure. In [11,15], the authors studied MOLDABLE and GRIDSHAPED applications that continue executing after some failures. They focus on the performance degradation incurred after shrinking or spare node substitution, due to less efficient communications (in particular collective communications). A major difference with our work is that these studies focus on recovery overhead and do not address overall performance nor yield.

3 Performance model

This section reviews the key parameters of the performance model. Some assumptions are made to simplify the computation of the yield. We discuss possible extensions in Section 6.

Application/platform framework. We consider perfectly parallel applications that execute on homogeneous parallel platforms. Without loss of generality, we assume that each processor has unit speed: we only need to know that the total amount of work done by p processors within T seconds requires $\frac{p}{q}T$ seconds with q processors.

Mean Time Between Failures (MTBF). Each processor is subject to failures which are IID (independent and identically distributed) random variables following an Exponential probability distribution of mean μ_{ind}, the individual processor MTBF. Then the MTBF of a section of the platform comprised of i processors is given by $\mu_i = \frac{\mu_{ind}}{i}$ [17].

Checkpoints. Processors checkpoint periodically, using the optimal Young/Daly period [25,8]: for an application using i processors, this period is $\sqrt{2C_i\mu_i}$, where C_i is the time to checkpoint with i processors[7]. We consider two cases to define

[7] In [8], the optimal checkpoitning period is $\sqrt{2C_i\mu_i} + C_i$, but we use $\sqrt{2C_i\mu_i}$ as derived in [17]. Note that both formulas are only first-order approximations and collapse when C_i is small in front of the MTBF μ_i. The exact formula for the optimal checkpointing period is given in [17].

C_i. In both cases, the overall application memory footprint is considered constant at Mem_{tot}, so the size of individual checkpoints is inversely linear with the number of participating/surviving processors. In the first case, the I/O bandwidth is the bottleneck (which is often the case in HPC platforms – it takes only a few processors to saturate the I/O bandwidth); then the checkpoint cost is constant and given by $C_i = \frac{Mem_{tot}}{\tau_{io}}$, where τ_{io} is the aggregated I/O bandwidth. In the second case, the processor network card is the bottleneck (which is the case for in-memory checkpointing, or checkpointing to NVRAM), and the checkpoint cost is inversely proportional to number of active processors: $C_i = \frac{Mem_{tot}}{\tau_{xnet} \times i}$, where τ_{xnet} is the available network bandwidth, and $\frac{Mem_{tot}}{i}$ the checkpoint size.

We denote the recovery time with i processors as R_i. For all simulations we use $R_i = C_i$, assuming that the read and write bandwidths are identical.

Objective. We consider a long-lasting application that requests a resource allocation with N processors. We aim at deriving the optimal number of failures F that should be tolerated before paying the wait time and requesting a new allocation. We aim at maximizing the *yield* $\mathcal{Y}$ of the application, defined as the fraction of time during the allocation length and wait time where the N resources perform useful work. Of course a spare does not perform useful work when idle, and no processor is active during wait time, which explains that the yield will always be smaller than 1. We will derive the value of F that maximizes $\mathcal{Y}$ fro the three application types.

4 Expected yield

This section is the core of the paper. We compute the expected yield for each application type, Rigid, Moldable and GridShaped.

4.1 Rigid application

We first consider a Rigid application that can be parallelized at compile-time to use any number of processors but cannot change this number until it reaches termination. There are N processors allocated to the application. We use $N - F$ for execution and keep F as spares. The execution is protected from failures by checkpoints of duration C_{N-F}. Each failure striking the application will incur an in-place restart of duration R_{N-F}, using a spare processor to replace the faulty one. However, when the $(F+1)^{st}$ failure strikes, the job will have to stop and perform a full restart, waiting for a new allocation of N processors to be granted by the job scheduler.

We define $\mathcal{T}_R$ as the expected duration of an execution period until the application is ready to continue after the $(F+1)^{st}$ failure strikes. We compute $\mathcal{T}_R$ using several first-order approximations. In particular, we ignore scenarios where failures strike during checkpoint, recovery or re-execution, thereby neglecting the probability of two failures within a short time window. Also, we approximate the

time lost after a failure as half the checkpointing period. Finally, we assume an integer number of checkpointing periods in between failures. The first failure is expected to strike after μ_N seconds, the second failure μ_{N-1} seconds after the first one, and so on. Without any overhead, the length of a period would be $\sum_{i=N}^{N-F} \mu_i$. Except for the last failure, each failure incurs some overhead only if it strikes the application. This happens with probability $\frac{N-F}{i}$, where i is the current number of live processors. In that case, the failure requires a restart and some re-execution, namely half the checkpoint period in average. The application always uses $N-F$ processors, hence the checkpoint period remains equal to $\sqrt{2C_{N-F}\mu_{N-F}}$. On the contrary, if the failure strikes a spare, there is no overhead. The last failure always requires a wait time, and then a restart and re-execution. Therefore, we derive:

$$\mathcal{T}_R = \sum_{i=N}^{N-F} \mu_i + \sum_{i=N}^{N-F+1} \frac{N-F}{i} \left(R_{N-F} + \frac{\sqrt{2C_{N-F}\mu_{N-F}}}{2} \right) + D + R_{N-F} + \frac{\sqrt{2C_{N-F}\mu_{N-F}}}{2}$$

What is the total amount of work $\mathcal{W}_R$ computed during a period? During the sub-period of length μ_i, there are $\frac{\mu_i}{\sqrt{2C_{N-F}\mu_{N-F}}}$ checkpoints, each of length C_{N-F}, and each processor works during $\frac{\mu_i}{1+\frac{C_{N-F}}{\sqrt{2C_{N-F}\mu_{N-F}}}}$ seconds. There are $N-F$ processors at work, hence

$$\mathcal{W}_R = (N-F) \cdot \sum_{i=N}^{N-F} \frac{\mu_i}{1 + \frac{C_{N-F}}{\sqrt{2C_{N-F}\mu_{N-F}}}}$$

During the duration $\mathcal{T}_R$ of the period, in the absence of failures and protection, the application could have used all N processors to compute. Thus the effective yield with protection for the application during $\mathcal{T}_R$ is reduced to $\mathcal{Y}_R$:

$$\mathcal{Y}_R = \frac{\mathcal{W}_R}{N \cdot \mathcal{T}_R}$$

4.2 Moldable Application

We now consider a MOLDABLE application that can use a different number of processors after each restart. The application starts executing with N processors; after the first failure, the application recovers from the last checkpoint and is able to continue with only $N-1$ processors after paying the restart cost R_{N-1}, albeit with a slowdown factor $\frac{N-1}{N}$ of the parallel work per time unit.

We define $\mathcal{T}_M$ as the expected duration of an execution period until the $(F+1)^{st}$ failure strikes. Without any overhead, the length of a period would be $\sum_{i=N}^{N-F} \mu_i$, the same as for RIGID applications. But there are few differences. First, each failure strikes the application, since it always uses all live processors. Second, the checkpoint period increases after each failure, since the number of live processors decreases. Third, the re-execution after a failure (except the

last one) incurs a slowdown factor because we move from i processors to $i-1$ processors. Fourth and finally, the re-execution after the last failure is performed faster, because there are more live processors. Altogether, we derive that

$$\mathcal{T}_M = \sum_{i=N}^{N-F} \mu_i + \sum_{i=N}^{N-F+1} \left(R_{i-1} + \frac{i}{i-1} \cdot \frac{\sqrt{2C_i\mu_i}}{2} \right) + D + R_N + \frac{N-F}{N} \frac{\sqrt{2C_{N-F}\mu_{N-F}}}{2}$$

To compute the total amount of work $\mathcal{W}_M$ during a period, we proceed as before and consider each sub-period. During the sub-period of length μ_i, there are $\frac{\mu_i}{\sqrt{2C_i\mu_i}}$ checkpoints, each of length C_i, and each processor works during $\frac{\mu_i}{1+\frac{C_i}{\sqrt{2C_i\mu_i}}}$ seconds. And there are i processors at work during that sub-period. Altogether:

$$\mathcal{W}_M = \sum_{i=N}^{N-F} i \times \frac{\mu_i}{1+\frac{C_i}{\sqrt{2C_i\mu_i}}}, \qquad \text{and } \mathcal{Y}_M = \frac{\mathcal{W}_M}{N \cdot \mathcal{T}_M}$$

where $\mathcal{Y}_M$ is the yield of the MOLDABLE application.

4.3 GridShaped application

Finally, we consider a GRIDSHAPED application, defined as a moldable execution which requires a rectangular processor grid. The application starts with a square $p \times p$ grid of $N = p^2$ processors. After the first failure, execution continues on a $p \times (p-1)$ rectangular grid, keeping $p-1$ processors as spares for the next $p-1$ failures. After p failures, the grid is shrunk again to a $(p-1) \times (p-1)$ square grid, and the execution continues on this reduced-size square grid. After how many failures F should the application stop, in order to maximize the application yield? The derivation of the expected length of a period and of the total work are more complicated for GRIDSHAPED than for RIGID and MOLDABLE. Due to lack of space, we refer to the extended version [12] , as well as to the publicly available software [21], for detailed formulas and an algorithm to to compute the optimal value of F.

5 Applicative scenario

As an applicative scenario, we consider a platform with 22,250 nodes (150^2), with a node MTBF of 20 years, and an application that would take 2 minutes to checkpoint (at 22,250 nodes). In other words, we let $N = 22,500$, $\mu_{ind} = 20y$ and $C_i = C = 120s$. These values are inspired from existing platforms: the Titan supercomputer at OLCF [14], for example, holds 18,688 nodes, and experiences a few node failures per day, implying a node MTBF between 18 and 25 years. The filesystem has a bandwidth of 1.4TB/s, and nodes altogether aggregate 100TB of memory, thus a checkpoint that would save 30% of that system should take in the

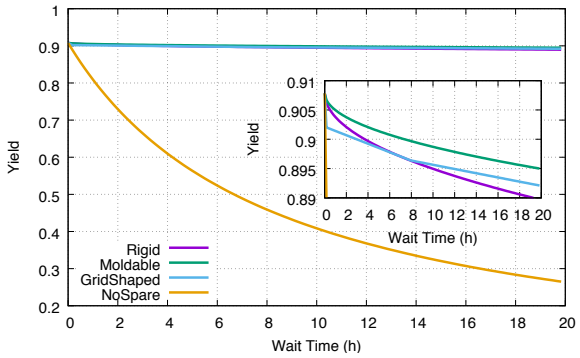

Figure 1: Optimal yield as function of the wait time, for the different types of applications.

order of 2 minutes to complete. Further experiments varying N, μ_{ind} and with several scenarios for checkpoint costs are available in the extended version [12].

Figure 1 shows the yield that can be expected if doing a full restart after an optimal number of failures, as a function of the wait time, for the three kind of applications considered (Rigid, Moldable and GridShaped). We also plot the expected yield when the application experiences a full restart after each failure (NoSpare). First, one sees that the three approaches that avoid paying the cost of a wait time after every failure experience a comparable yield, while the performance of the NoSpare approach quickly degrades to a small efficiency (30% when the wait time is around 14h).

The zoom box to differentiate the Rigid, Moldable and GridShaped yield shows that the Moldable approach has a slightly higher yield than the other ones, but only for a minimal fraction of the yield. This is expected, as the Moldable approach takes advantage of all living processors, while the GridShaped and Rigid approaches sacrifice the computing power of the spare nodes waiting for the next failure. However, the size of the gain is small to the point of being negligible. The GridShaped approach experiences a yield that changes in steps. Both these phenomenons are explained by the next figure.

Figure 2 shows the number of failures after which the application should do a full restart, to obtain an optimal yield, as a function of the wait time, for the three kind of applications considered. We observe that this optimal is quickly reached: even with long wait times (e.g. 10h), 200 to 250 failures (depending on the method) should be tolerated within the allocation before relinquishing it. This is small compared to the number of nodes: less than 1% of the resource should be dedicated as spares for the Rigid approach, and after losing 1% of the resource, the Moldable approach should request a new allocation.

This is remarkable, taking into account the poor yield obtained by the approach that does not tolerate failures within the allocation. Even with a small wait time (assuming the platform would be capable of re-scheduling applica-

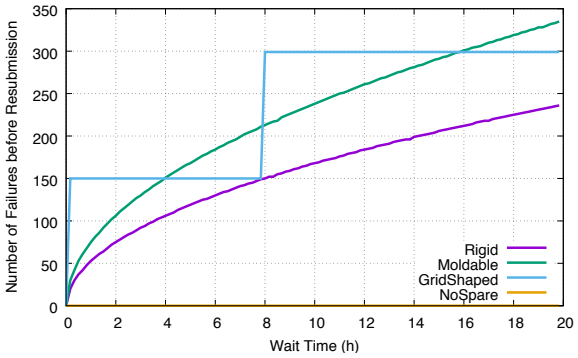

Figure 2: Optimal number of failures tolerated between two full restarts, as function of the wait time, for the different types of applications.

tions that experience failures in less than 2h), Figure 1 shows that the yield of the NoSpare approach would decrease to 70%. This represents a waste of 30%, which is much higher than the recommended waste of 10% for resilience in the current HPC platforms recommendations [7,4]. Comparatively, provisioning only 1% of additional resources as spares within the allocations, would allow to achieve a yield over 88%, for every approach considered, when the wait time does not exceed 20 hours.

The GridShaped approach experiences steps that correspond to using all the spares created when redeploying the application over a smaller grid before relinquishing the allocation. As illustrated in Figure 1, the yield evolves in steps, changing the slope of a linear approximation radically when redeploying over a smaller grid. This has for consequence that the maximal yield is always at a slope change point, thus at the frontier of a new grid size. It is still remarkable that even with very small wait times, it is more beneficial to use spares (and thus to lose a full row of processors) than to redeploy immediately.

Figure 3 shows the length of an allocation providing the optimal yield (best value of F). After such a duration, the job will have to fully restart in order to maintain the optimal yield. This figure illustrates the real difference between the Rigid and Moldable approaches: although both approaches are capable of extracting the same yield, the Moldable approach can do so with significantly longer periods between full restarts. This is important when considering real life applications, because this means that the applications using a Moldable approach have a higher chance to complete before the first full restart, and overall will always complete in a lower number of allocations than the Rigid approach.

Finally, Figure 4 shows an upper limit of the duration of the wait time in order to guarantee a given yield for the three applications. In particular, we see that to reach a yield of 90%, an application which would restart its job at each fault would need that restart to be done in less than 6 minutes whereas the Rigid and GridShaped approaches need a full restart in less than 3 hours

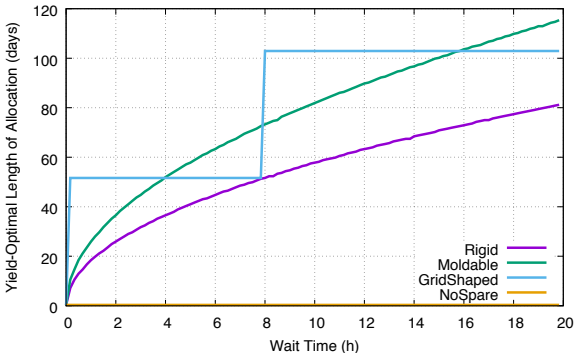

Figure 3: Optimal length of allocations, for the different types of applications.

approximately. This bound goes up to 7 hours for the MOLDABLE approach. In comparison, with a wait time of 1 hour, the yield obtained using NOSPARE is only 80%. This shows that, using these parameters, it seems impossible to guarantee the recommended waste of 10% without tolerating (a small) number of failures before rescheduling the job.

6 Conclusion

In this paper, we have compared the performance of RIGID, MOLDABLE and GRIDSHAPED applications when executed on large-scale failure-prone platforms. For each application type, we have computed the optimal number of faults that should be tolerated before requesting a new allocation, as a function of the wait time. Through a realistic applicative scenario inspired by state-of-the-art platforms, we have shown that the three application types experience an optimal yield when requesting a new allocation after experiencing a number of failures that represents a small percentage of the initial number of resources (hence a small percentage of spares for RIGID applications), and this even for large values of the wait time. On the contrary, the NOSPARE strategy, where a new allocation is requested after each failure, sees its yield dramatically decrease when the wait time increases. We also observed that MOLDABLE applications enjoy much longer execution periods in between two re-allocations, thereby decreasing the total execution time as compared to RIGID applications (and GRIDSHAPED applications lying in between).

Future work will be devoted to exploring more applicative scenarios. We also intend to extend the model in several directions. On the application side, we aim at dealing with non-perfectly parallel applications but instead with applications whose speedup profile obeys Amdahl's law [1]. We will also introduce a more refined speedup profile for GRIDSHAPED applications, with an execution speed that depends on the grid shape (a square being usually faster than an elongated

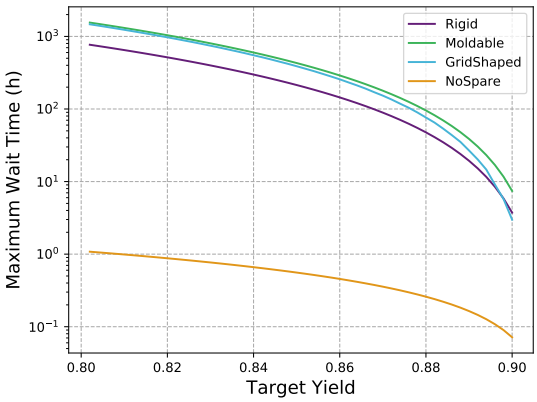

Figure 4: Maximum wait time allowed to reach a target yield.

rectangle). On the resilience side, we will address forward-recovery schemes, such as ABFT [16,9], in replacement of, or in combination with, checkpoint-restart techniques.

References

1. G. Amdahl. The validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485. AFIPS Press, 1967.
2. R. A. Ashraf, S. Hukerikar, and C. Engelmann. Shrink or substitute: Handling process failures in HPC systems using in-situ recovery. *CoRR*, abs/1801.04523, 2018.
3. W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra. Post-failure recovery of MPI communication capability: Design and rationale. *International Journal of High Performance Computing Applications*, 27(3):244–254, 2013.
4. F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir. Toward exascale resilience: 2014 update. *Supercomputing frontiers and innovations*, 1(1), 2014.
5. A. Cavelan, J. Li, Y. Robert, and H. Sun. When Amdahl meets Young/Daly. In *Cluster'2016*. IEEE Computer Society Press, 2016.
6. W. Cirne and F. Berman. Using moldability to improve the performance of supercomputer jobs. *J. Parallel Distrib. Comput.*, 62(10):1571–1601, 2002.
7. CORAL: Collaboration of Oak Ridge, Argonne and Livermore National Laboratorie. Draft CORAL-2 build statement of work. Technical Report LLNL-TM-7390608, Lawrence Livermore National Laboratory, March, 30 2018.
8. J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Comp. Syst.*, 22(3):303–312, 2006.
9. P. Du, A. Bouteiller, et al. Algorithm-based fault tolerance for dense matrix factorizations. In *PPoPP*, pages 225–234. ACM, 2012.

10. P. Dutot, G. Mounié, and D. Trystram. Scheduling parallel tasks approximation algorithms. In J. Y. Leung, editor, *Handbook of Scheduling - Algorithms, Models, and Performance Analysis*. CRC Press, 2004.
11. A. Fang, H. Fujita, and A. A. Chien. Towards understanding post-recovery efficiency for shrinking and non-shrinking recovery. In *Euro-Par 2015: Parallel Processing Workshops*, pages 656–668. Springer, 2015.
12. V. L. Fèvre, G. Bosilca, A. Bouteiller, T. Herault, A. Hori, Y. Robert, and J. Dongarra. Do moldable applications perform better on failure-prone hpc platforms? Research report RR-9174, INRIA, 2018.
13. Y. Guo, W. Bland, P. Balaji, and X. Zhou. Fault tolerant MapReduce-MPI for HPC clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*, pages 34:1–34:12, 2015.
14. S. Gupta, T. Patel, C. Engelmann, and D. Tiwari. Failures in large scale systems: Long-term measurement, analysis, and implications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 44:1–44:12, New York, NY, USA, 2017.
15. A. Hori, K. Yoshinaga, T. Herault, A. Bouteiller, G. Bosilca, and Y. Ishikawa. Sliding substitution of failed nodes. In *Proceedings of the 22Nd European MPI Users' Group Meeting*, EuroMPI '15, pages 14:1–14:10, New York, NY, USA, 2015. ACM.
16. K.-H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Comput.*, 33(6):518–528, 1984.
17. T. Hérault and Y. Robert, editors. *Fault-Tolerance Techniques for High-Performance Computing*. Springer Verlag, 2015.
18. H. Jin, Y. Chen, H. Zhu, and X.-H. Sun. Optimizing HPC fault-tolerant environment: An analytical approach. In *Proc. ICPP'10*, 2010.
19. J. E. Moreira and V. K. Naik. Dynamic resource management on distributed systems using reconfigurable applications. *IBM Journal of Research and Development*, 41(3):303–330, 1997.
20. S. Prabhakaranw. *Dynamic Resource Management and Job Scheduling for High Performance Computing*. PhD thesis, Technische Universität Darmstadt, 2016.
21. Simulation Software. Computing the yield. `https://github.com/vlefevre/continuability`, 2018.
22. R. Sudarsan and C. J. Ribbens. Design and performance of a scheduling framework for resizable parallel applications. *Parallel Computing*, 36(1):48–64, 2010.
23. R. Sudarsan, C. J. Ribbens, and D. Farkas. Dynamic resizing of parallel scientific simulations: A case study using LAMMPS. In *Int . Conf . Computational Science ICCS*, pages 175–184. Procedia, 2009.
24. K. Yamamoto, A. Uno, H. Murai, T. Tsukamoto, F. Shoji, S. Matsui, R. Sekizawa, F. Sueyasu, H. Uchiyama, M. Okamoto, N. Ohgushi, K. Takashina, D. Wakabayashi, Y. Taguchi, and M. Yokokawa. The K computer Operations: Experiences and Statistics. *Procedia Computer Science (ICCS)*, 29:576–585, 2014.
25. J. W. Young. A first order approximation to the optimum checkpoint interval. *Comm. of the ACM*, 17(9):530–531, 1974.
26. Z. Zheng, L. Yu, and Z. Lan. Reliability-aware speedup models for parallel applications with coordinated checkpointing/restart. *IEEE Trans. Computers*, 64(5):1402–1415, 2015.