

Exploring Energy Saving Opportunities in Fault Tolerant HPC Systems

Marina Morán¹, Javier Balladini¹, Dolores Rexachs², and Enzo Rucci³

¹Dpto. Ingeniería de Computadoras, Facultad de Informática, Universidad Nacional del Comahue, Argentina, {marina.moran,javier.balladini}@fi.uncoma.edu.ar

²Dpto. Arquitectura de Computadores y Sistemas Operativos, Universidad Autónoma de Barcelona, España, dolores.rexachs@uab.es

³III-LIDI, Facultad de Informática, Universidad Nacional de La Plata - CIC, La Plata, Buenos Aires, Argentina, erucci@lidi.info.unlp.edu.ar

October 27, 2023

Abstract

Nowadays, improving the energy efficiency of high-performance computing (HPC) systems is one of the main drivers in scientific and technological research. As large-scale HPC systems require some fault-tolerant method, the opportunities to reduce energy consumption should be explored. In particular, rollback-recovery methods using uncoordinated checkpoints prevent all processes from re-executing when a failure occurs. In this context, it is possible to take actions to reduce the energy consumption of the nodes whose processes do not re-execute. This work is an extension of a previous one, in which we proposed a series of strategies to manage energy consumption at failure-time. In this work, we have enriched our simulator and the experimentation by including non-blocking communications (with and without system buffering) and a largest number of candidate processes to be analyzed. We have called the latter as *cascade analysis*, because it includes processes that gets blocked by communication indirectly with the failed process. The simulations show that the savings were negligible in the worst case, but in some scenarios, it was possible to achieve significant ones; the maximum saving achieved was 90% in a time interval of 16 minutes. As a result, we show the feasibility of improving energy efficiency in HPC systems in the presence of a failure.

This is the accepted version of the manuscript that was sent to review to Journal of Parallel and Distributed Computing (ISSN 1096-0848). This manuscript was finally accepted for publication on October 27th, 2023 and its final published version is available online at <https://doi.org/10.1016/j.jpdc.2023.104797>. ©2023. This manuscript version is made available under the CC-BY-NC-ND 4.0 license

Keywords— E nergy saving Fault Tolerance Methods Checkpoint Parallel Applications ACPI DVFS

1 Introduction

Nowadays, improving the energy efficiency of high-performance computing (HPC) systems is one of the main drivers in scientific and technological research. In this context, exploring different ways to reduce energy consumption during the execution of large-scale applications is essential to maintain and increase the enormous computing power achieved. Even more, increasing the number of processing units also requires scalable fault tolerance (FT) methods. Thus, it is relevant to evaluate the energy-saving opportunities presented by these methods.

A message-passing application can be affected by failures on multiple components in distributed memory computing systems. In HPC, some methods allow continuing with the execution of the application in the presence of a failure. One of the most widely used methods is rollback-recovery through the use of checkpoints. When a failure occurs, the application can restart its execution from the checkpoints. Checkpoints can be performed in coordinated (synchronously) or uncoordinated (asynchronously) manner. In the first case, all application processes must stop, perform the checkpoint, and then continue with their execution. When a failure occurs, all processes restart from the last checkpoint. In the second case (uncoordinated checkpoints), the processes usually perform their checkpoint at different times. When a node fails, the processes of the non-failed nodes can continue their execution. As it will be shown, the uncoordinated FT method presents an interesting opportunity to explore power-saving scenarios.

The challenge is to investigate what possibilities exist to reduce energy consumption when one or more processes stop their execution due to communication blocks with processes directly or indirectly affected by the failure. From an energy-saving point of view, how to take advantage of the fact that uncoordinated checkpoints prevent all the application processes from having to go back in the presence of a failure? One possibility is to use the nodes of the processes that did not fail to execute another application of the system job queue, to use those resources for the computation of another application. Another possibility consists of keeping the affected application running and applying a series of strategies on the nodes whose processes do not need to be recovered. However, these processes, at a certain moment of execution, may be affected by long waits due to communications with processes that are recovering. Hence, if waiting is unavoidable, what is the best strategy to consume less energy at that time?

Our objective is to know and manage the energy consumption of an HPC system by applying different strategies. The strategies to be applied depend on the underlying hardware features and the state of the application when a permanent failure occurs¹. Strategies that increase application execution time (in relation to the reference execution time when a failure occurs) are discarded. We consider the use of Dynamic Voltage and Frequency Scaling (DVFS) techniques and system hibernation at the node level, using the Advanced Configuration

¹Permanent failures are those which cause fail-stops in Message Passing Interface (MPI).

and Power Interface (ACPI). By having a characterization of the energy consumption required to execute the application and its communication pattern, it is possible to estimate the energy consumption under certain strategies from the moment of a fault. Then, by using a simulator that we have designed and developed, we can evaluate the use of the strategies. Using a simulator allows us to simplify a real system, reduce costs, and focus on essential features. In our case, it also allows us to have a flexible environment to experiment with different configurations. In addition, it integrates with a pre-existing tool that allows us to view its results.

In [1] we show how the power consumption of checkpoint and restart operations varies by lowering the processor clock frequency. This work, which is the continuation of [2] and [3], presents a revision of the energy model, and the following contributions:

- The definition of a series of strategies for energy saving when a failure occurs; in particular, these strategies are applied to nodes of the processes that do not have to rollback. In [2] we analyzed processes/nodes that communicate directly with the node where the failure occurred. In this work, we increase the number of candidate processes to be analyzed to apply some energy-saving strategy.
- The design and development of a simulator oriented to evaluate the proposed strategies and to select the most convenient one from the energy point of view. In this work, we extended the simulator to include non-blocking communication operations (with and without system buffering) and the analysis of the new candidate processes indicated in the previous contribution.

This work is organized as follows: Section 2 discusses some related works. Next, Section 3 presents some preliminary concepts used in the article and Section 4 describes the strategies evaluation and application. Then, Section 5 presents the energy model and Section 6 describes the simulator. Finally, the experimental results are shown and discussed in section 7, while conclusions and future work are summarized in section 8.

2 Related Work

In recent years the number of papers about energy consumption in HPC systems has been increasing. We mention here some of them, classified according to whether they are related to Fault Tolerance or not.

2.1 Fault Tolerance and energy consumption

Some works evaluate the energy behavior of fault tolerance methods (rollback recovery, replication). In [4], the authors evaluate the consumption of checkpoint and restart operations at different processor clock frequencies and on different input/output devices, and design a runtime software to minimize energy consumption. In [5] propose a computational model, called shadow computing, which provides dynamic adaptive resilience. They use redundancy to implement fault tolerance, through shadow processes, which run at a lower clock rate

than the main process. They present an energy model to estimate the execution speed that minimizes energy consumption. [6] use DVFS to throttle CPU speed during checkpoint writes to measure the energy savings and performance impact. A comparative evaluation of energy consumption in three different rollback-recovery protocols (checkpoint/restart, message logging, and parallel recovery) is done in [7]. They evaluate the three protocols in a cluster with the capacity to measure the power dissipated, and develop an energy model to make projections for large-scale systems under different conditions.

A calibrator that allows measuring the energy consumption of the operations used in three methods of fault tolerance is presented in [8]. These data feed a framework (called Ecofit), to estimate the energy consumption of an application with a given TF method. Another work that also proposes a framework (called PowerCheck) that seeks to minimize energy consumption (but only for checkpoint operations) is the work presented at [9]. The framework consists of a user-space library, which among other tasks interacts with Running Average Power Limiting (RAPL) to measure and limit the power dissipated during the checkpoint.

Some works try to estimate the optimal checkpoint interval in energy terms, like [10] [11] [12].

In [13], the authors propose using non-recovering nodes to execute another application from the system queue, to take advantage of the computing power of those nodes and improve the global performance of the cluster. Unlike this proposal, we also propose other strategies that use DVFS and hibernation on nodes that continue to run, without changing the application.

Finally, [14] is the most similar proposal to this work, since they propose a localized rollback based on the data flow, and reduce the clock frequency of the waiting processes to the minimum possible. We evaluate other strategies, in addition to changing to the minimum frequency, and we do so both for the computation and waits of the processes that continue to execute.

2.2 Energy saving

This work has similarities to others that slow down the non-critical path to consume less power without substantially increasing execution time: [15], [16]. While most of these proposals apply clock frequency adjustments during the application running, we apply it at failure time. For example, [15] uses a per-core Software Controlled Clock Modulation or DVFS to throttle the frequencies of cores not on the critical path of an MPI application. In [16] they adjust the clock frequencies of the tasks in a manner that they arrive just in time at the synchronization moment.

Some works use hardware counters to predict energy consumption, f.e. [17], where they run various machine learning methods, measure hardware counters, and analyze which ones correlate best with runtime, and CPU and memory power consumption. To do this, the authors get some functions to predict the time and energy savings when applying possible optimizations to the applications. In [18] they propose a model for estimating the power consumed by an application using the least amount of performance monitoring counters possible.

In [19], they perform a simulation of an HPC workload to evaluate the energy savings when activating and deactivating nodes according to the computational and power requirements of the cluster. In this work, we also shut down nodes

when the waiting phase is long enough. Other works analyze the active waits of an MPI application and evaluate potential energy savings by changing the clock frequency during those waits, as in [20], or entering a power-gated state executing a sleep call, as in [21]. Active waits are also an energy-saving point in our proposal.

3 Background

The following subsections introduce some concepts that are used in the article, such as the states defined by ACPI, operation modes and active waits in MPI, and rollback recovery mechanism.

3.1 ACPI

The ACPI specification provides an open standard that allows the operating system to manage the power of the computing system and provides advanced mechanisms for energy management². The specification defines a series of global states and substates for the system. In the global *working* G0 state the applications are executed. In this global state, there is one executing state (C0) and some inactive states (C1..Cx). Inside C0, performance states (P states) are defined, which determine performance and power demand. In the global *sleeping* state G1 the computer consumes a small amount of power and applications are not executed. In this state, the context is saved, so the operating system does not need to restart when waking up. The latency for returning to the working state varies on the type of sleeping substates selected (S1-S4). There are two more states, *soft* and *mechanical off* not used in this work. Table 1 shows a brief description of the states and substates.

In this work, we use performance and sleeping states (P and S states). In GNU/Linux, it is possible to change the frequency of the processor (P states) with the userspace governor, which permits modifying the frequency of each core. To change the sleeping state (S states) a string has to be written to the `/sys/power/state` file. In the simulations, when we mention the strategy of sleeping the node we refer to the Suspend-to-RAM (S3) state, and when we mention clock frequency changes, the same change is applied to all cores of the node.

3.2 Communication operations in MPI

The MPI standard provides blocking and non-blocking operations for message passing³. Blocking send operations are characterized by returning as soon as it is safe to reuse the buffer (either because the message was copied into the receiving buffer or a sender's system buffer). In standard mode (MPI_Send, MPI_Recv), it is up to the MPI implementation to decide whether outgoing messages will be buffered. If the system provides buffering, the send call may complete before a matching receive is invoked. If the system does not provide buffering, the send call will not complete until a matching receive operation has been posted, and the data has been moved to the receiver buffer.

²<https://www.uefi.org/specifications>

³<https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf>

Table 1: ACPI global states and substates

Global State	Substates
G0 Working	Processor Power States. C0 : Executing state. A processor that is in the C0 state will also be in a Performance State (Px states). The P0 state means an execution at the maximum capacity of performance and power demand. As the number of P state increases, their performance and demanded power is reduced. The processors implement the P states using the technique of DVFS. C1...Cx : inactive states. Very short transition latencies in comparison to transition latencies between states G0-G1
G1 Sleeping	S1-S4 are idle states of the system associated with G1. Increasing the state number requires less power and longer latency to exit the state. There are no transitions between S1-S4, it is always with S0.
G2 Soft Off	S5 . It consumes a minimum of energy. No applications are running and the system context is not saved.
G3 Mechanical Off	The system is completely off and there is no electrical current running through the circuit.

Non-blocking operations return immediately, and another operation, usually called wait (`MPI_Wait`), is used to verify that it is safe to reuse the buffer for the send case, or that the data is already in the receive buffer for the receive case. This allows us to overlap computation with communication. In the standard mode (`MPI_Isend`, `MPI_Irecv`), if the system buffer is used, the wait will return after copying the data to that buffer. If no system buffer is present, the wait will return after the data is copied to the receiver buffer. Fig. 1 shows the return point of the non-blocking operations with and without using the system buffer. In this work, we consider both blocking and non-blocking operations in their standard mode.

3.3 Active and idle waiting in MPI

In MPI parallel applications, it may happen that one process must wait for another to send or receive a message. During these waits, the process can keep the processor busy by active-waiting, or releasing it, and using polling or interrupts. Active-waiting means polling at the highest available frequency for a signal to be able to react instantly once the signal is received [20]. An active wait keeps the processor busy and consumes energy without doing useful work. An idle wait can affect application performance, due to C states transitions [22]; this is why various MPI implementations provide active wait as the default operating mode. As this operating mode is configurable, in this work we consider both cases.

3.4 Rollback recovery

A consistent global state can be found during a successful and fault-free execution of parallel computing. Inconsistent states can occur because of failures.

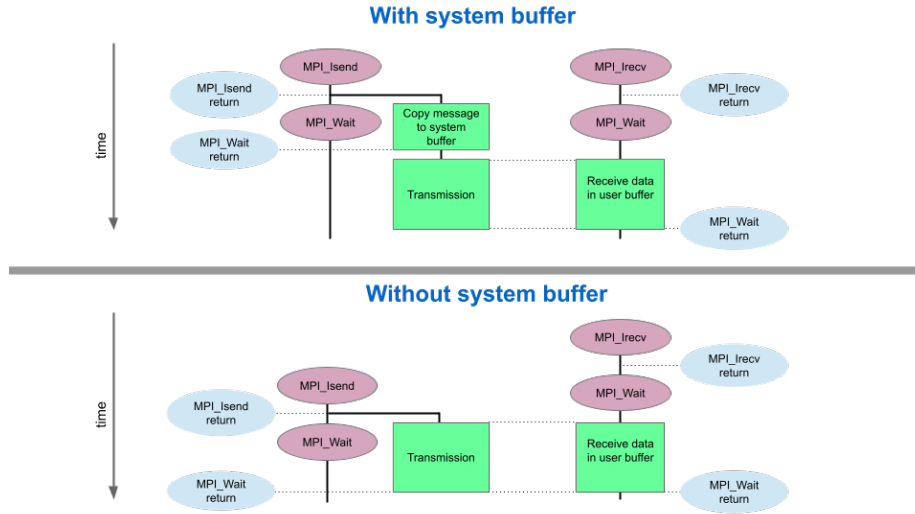


Figure 1: Non-blocking operations return point.

A fundamental goal of any rollback-recovery protocol is to lead the system to a consistent state after a failure. This method consists of periodically saving the state of the application in stable storage, which is known as a *checkpoint*. At failure time, it is possible to restart the application from the last successfully saved state, which is called *restart*. In the case of coordinated checkpoints, a consistent global state is obtained by synchronizing all the processes, stopping their execution, and performing the checkpoint. When a process fails, all processes must restart from the last checkpoint. As we can see, all the application processes re-executing produce energy and time overhead. In the case of uncoordinated checkpoints, processes take their checkpoints independently, avoiding synchronization time and I/O contention [23]. At failure time, only failed processes restart from the last checkpoint, using fewer resources for their recovery than a coordinated checkpoint. However, ensuring a consistent global state is not as straightforward as in the case of coordinated checkpoints. When a process restarts, orphaned and/or lost messages can appear, causing other processes to roll back to ensure consistency. This is called *domino effect*, and there are different techniques to control it, such as the use of message logging [24]. Uncoordinated checkpoints allow the use of advanced checkpoints. If a process is going to block by communication, and its last checkpoint happened a relatively long time ago, the process performs a checkpoint before blocking. In this way, useless waiting time is used by a checkpoint operation. In this work, we consider the use of advanced checkpoints.

Hybrid approaches also exist and they take advantage of coordinated and uncoordinated checkpoints. In this scheme, the processes are divided into groups. Within each group, a coordinated checkpoint scheme is used, but between groups, an uncoordinated checkpoint strategy is followed. There are different criteria for defining groups. For example, all processes running on the same node could be in a group, because when a node fails all its processes must restart[25]. Another way to define groups can be with processes that communi-

cate frequently [26]. The first approach is the one used in the present work.

4 Strategies definition and application

The use of uncoordinated checkpoints as a fault tolerance method of a message passing HPC application enables one to take actions on the nodes that should not recover from the failure. At the time of a failure, it is possible to apply two different options, which depending on the state of the system, will have a different impact on indices such as cluster productivity and energy consumption. One option is to change the application and the other one is to maintain it, but changing the P and S states of the surviving nodes. Both options are defined and analyzed below.

4.1 Change of the application

When a node fails, one possible action to take is to change the application, as proposed in [13]. One of the nodes that did not fail is used for process recovery (restart and re-execution). The rest of the nodes are used to run another application in the job queue. In this way, the productivity of the computer system is increased. To apply this option, the recovery time must be long enough so that it is worth changing the application. While the processes that were running on the failed node recover, the rest of the nodes do useful work allowing another application to move forward.

Switching between applications is a costly task in terms of time and resources because the filesystem has to be accessed multiple times, with consequent energy consumption. The checkpoints are performed on the application that comes out (write operations), the checkpoints of the incoming application are loaded (read operations) and the restart is executed (read operations). These steps must then be repeated to reload the original application. If the re-execution time is long enough to outweigh the cost of these operations, this alternative allows nodes that did not fail to continue running at maximum performance. Allowing another application to proceed while another recovers from a failure improves the overall productivity of the system.

4.2 Changes of the P and S states

This proposal focuses on energy saving and that is why a set of strategies based on changing the P and S states (see subsection 3.1) is evaluated to apply to the nodes that did not fail. This subsection defines the possible strategies to apply after the failure of a node when the application is not changed. In addition, it shows why it is important to consider both direct and indirect process blocking, and discusses some situations to take into account in the application of the strategies.

Fig. 2 shows different scenarios, where two processes, P1 and P2, are running on different nodes. The little squares indicate that the process is performing checkpoints. As uncoordinated checkpoints are used, they may be performed at different times, as shown in the figure. **Case A** shows a failure-free execution, where P1 sends two messages to P2, indicated by t_{send_1} and t_{send_2} . Messages are sent and received in an order scheduled by the application, which

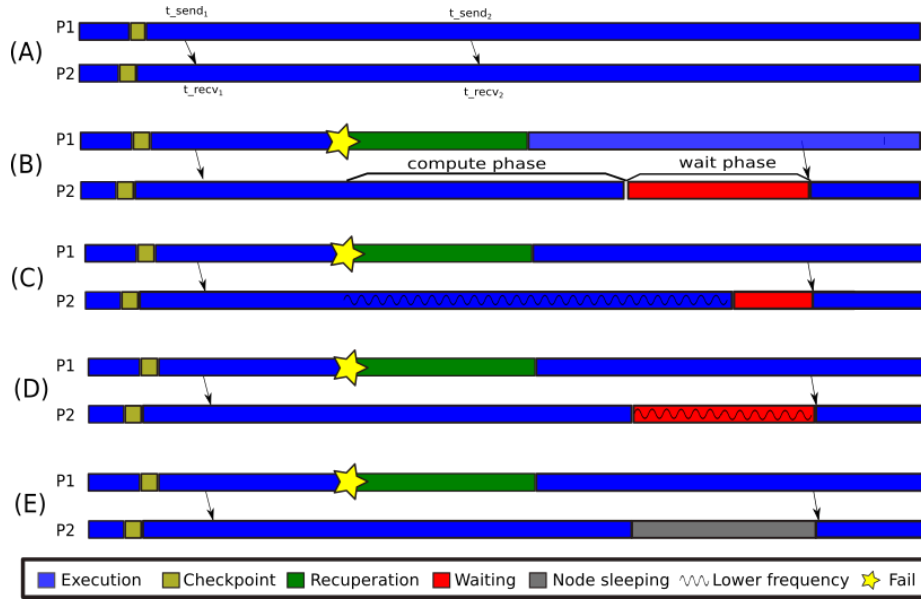


Figure 2: Strategies application for different failure cases. (A) No failure occurs. (B) A failure occurs and no action is taken (reference case). (C) A failure occurs and frequency change for the computational phase is taken. (D) A failure occurs and frequency change in the waiting phase is taken. (E) A failure occurs and sleeping in the waiting phase is taken.

maximizes the time dedicated to useful computation while minimizing the waits for communications (data transfers and synchronizations). In an efficient parallel application, waits not related to the failure are minimal. For simplicity, we consider these waits within the computation phase. **Case B** to **case E** shows an execution where process P1 fails (indicated by a yellow star) and must recover. This figure (and the next one) assumes the sending process always fails. However, this does not affect the application of the strategies, because this application is not affected by whether the failing process is the sender or the receiver. Instead, the analysis focus on the moment in which the alive process gets blocked by communication due to a failure. After a failure, the restart is initiated, and the re-execution begins (indicated in blue). We can see how the sending of the second message is delayed due to the failure, and the process P2 must wait (indicated in red or gray). In **Case B** no action is taken, and it will serve as a reference for the evaluation of the strategies. In this case, the computing and waiting phases are indicated. These phases are defined for processes that do not fail, and are defined as follows:

- The **compute phase** comprises the execution of the application from the moment of failure until it gets blocked because of the failure. This blocking is caused by a communication operation with a process that has failed or that is blocked because of the failure. Communications with other processes are included in the computing phase. In this way, additional waits can appear, not related to the failure. It may also happen that during this phase checkpoints are performed.

- The **waiting phase** begins with the blocking related to the failure and ends when the communication concludes and the process begins to compute again.

At failure time, the best strategy for the computation and waiting phase is determined. The strategy for the computation phase can be to lower the clock frequency. The strategies for the waiting phase can be to sleep the node (in some state S1-S4) or to switch to the minimum clock frequency. The computation and waiting phases altogether, form the *intervention interval*.

In **case C**, the P2 node changes to the selected clock frequency during the computation phase, indicated by the wavy line. This makes P2 submit the second receive operation later, shortening the waiting phase. The selected clock frequency must meet two criteria:

1. The duration of the computation phase running with the selected frequency must not exceed the synchronization time of the process. That is, the selected frequency must not cause waits in other processes.
2. The energy consumption of the compute phase in conjunction with the waiting phase should be lower than the consumption in the reference case (case B).

The selected frequency may be the maximum frequency, in which case there is no change in frequency. For simplicity, we always select a single frequency for the entire computation phase, although it could be the case that it is a combination of two or more frequencies that best meet the criteria just mentioned (applying each frequency during a fraction of the computation phase).

In **case D**, a change of the clock frequency during the waiting phase was decided, indicated by the wavy line. In **case E**, the node was sent to sleep during the waiting phase. Neither of these actions affects the waiting phase duration, but it does impact energy consumption indeed.

The strategies explained above can be applied in combination. For example, it could be the case that the clock frequency is changed for the computation phase and the waiting phase. As we can see, there are several possible scenarios where different actions must be evaluated and managed.

The strategies can be summarized as follows:

- Frequency change for the computational phase (case C).
- Frequency change for the waiting phase (case D).
- Sleeping for the waiting phase (case E).

The evaluation of the strategies for the computing and waiting phase is done altogether, considering the impact on energy consumption and execution time. The selected configuration will be the one that achieves the lowest energy consumption for the *intervention interval*, without affecting the application execution time. For this, the selected frequency for the computation phase of a process should avoid that other processes having to wait for it. Regarding the waiting phase, if the duration of this phase is long enough to sleep the node and subsequently wake up it, achieving a lower energy consumption, then this option is selected. Otherwise, if the waiting operations are configured to be

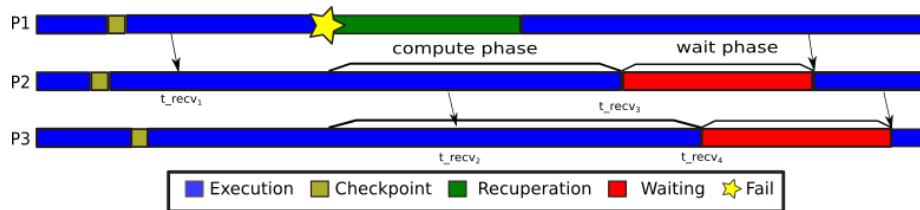


Figure 3: Cascade blocking and communications depth.

active waits, the way to minimize energy consumption is by using the minimum frequency. If the waiting operations are configured to be idle waits, the way to minimize energy consumption is to do nothing [20].

4.2.1 Cascade blocking and communications depth

Now we consider the processes which are indirectly affected by a failure. In Fig. 3, P3 does not block with the failed process; it blocks with an alive process, that is blocked because of the failure. We name *cascade blocking* to these blocking situations, which arise as a result of the failure but are not directly related to the failed process. These blockings are propagated over a set of processes during the execution of the application from the moment of failure. This situation can occur in the immediate following communication, or in the subsequent ones. We define *depth* to be the number of subsequent communications to consider after the failure, looking for one that is blocked for this reason. In Fig. 3, P3 blocks on the second communication with P2, giving a depth of 2. The depth can be calculated by looking at the pattern of communications between each pair of processes and choosing the maximum number of communications found in the pattern.

4.2.2 Estimation of the compute and waiting phases

To choose the strategies an algorithm has been developed that estimates the blocking times of each process. In this algorithm, we call *children* of a process, the processes with which it communicates. The main steps are summarized below:

- First, the lists are filled; *List1* contains the processes of the parent level, initially the processes of the failed node. *List2* contains the child processes of the processes in *List1*, with the times at which they are estimated to block for communication with their parents. For example, for the case shown in Fig. 4, the lists would be as follows:

```
List1={P1}
List2={(P2,t1);(P3,t3)}
```

- The algorithm then visits each *List2* process and checks if it will be blocked earlier by another *List2* process (sibling process), in which case it updates the block time. Continuing with the example, by visiting the first element of *List2*, we see that P2 does not block before with any sibling process. Visiting the second element of *List2*, P3, we see that it blocks with P2

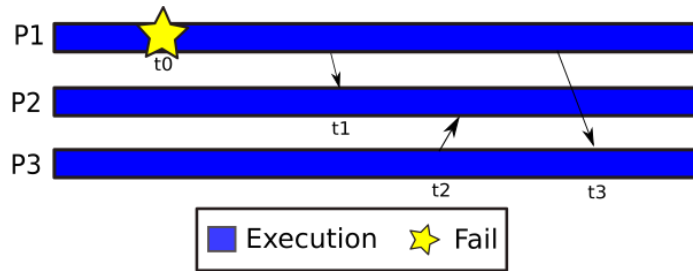


Figure 4: View of three processes at the time of failure t_0 .

at a time before the current block time of P3, and after the block time of P2. That is, P3 blocks at a time t_2 that verifies that $t_1 < t_2 < t_3$. After the update, *List2* would look like this:

$List2 = \{(P2, t_1); (P3, t_2)\}$

- Finally, we proceed to the next level, assigning *List2* to *List1* and emptying *List2*. After this step, the lists would be as follows:

$List1 = \{(P2, t_1); (P3, t_3)\}$
 $List2 = \{\}$

These steps are repeated until there are no more processes affected by the failure and unanalyzed.

The output of the algorithm is the time at which each process is expected to block due to the failure. The complexity of this algorithm is in the order of the number of processes by the number of communications of each process. For more detail, you can see the pseudocode in Algorithm 1.

This algorithm is suitable for applications with long compute phases, which allow the application of some strategies in these phases. In applications with short compute phases, where the impact of the frequency change is likely to be negligible, it may be convenient to estimate only the waiting phases and evaluate the strategies for these phases.

4.2.3 Evaluation of the energy-saving strategies

Once the computing and waiting phases of each process have been estimated, the possible strategies are evaluated. For this, the compute phase of each live process is instantiated with each of the possible clock frequencies, taking into account the restriction of not affecting the reference time. Based on this, the new duration of the waiting phase and the appropriate action to take are evaluated. Then the combination of strategies that obtain the lowest estimated energy consumption in each case is selected. In a previous work [2], we showed a pseudocode for strategy evaluation, which has been implemented in a simulator. This algorithm implements what is described in the energy model in section 5, and is in the order of the number of clock frequencies by the number of processes. The evaluation of the strategies for each process can be computed in parallel.

Algorithm 1 Pseudo-code to analyze cascade-blocked processes

```
1: List1 = ( $P_{failed}$ , 0); // The elements of the lists are a pair (process id,
   block time)
2: List2 = ListGlobal = {};
3: while there are processes that communicate with processes in List1 that do
   not belong to ListGlobal do
4:   for each ( $P_{parent}$ , parent_block_time) in List1 do
5:     for each process  $P_{child}$ , child of  $P_{parent}$  do
6:       comm_time = next_comm( $P_{child}$ ,  $P_{parent}$ ); // Find the next
   communication between both processes
7:       i = 0;
8:       while (comm_time < parent_block_time) AND (i < depth) do
9:         comm_time = next_comm( $P_{child}$ ,  $P_{parent}$ );
10:        i++;
11:       end while
12:       if  $P_{child}$  belongs to List2 then
13:         block_time = get_block_time( $P_{child}$ );
14:         if comm_time < block_time then
15:           Update_List2( $P_{child}$ , comm_time) // Update  $P_{child}$  block time
16:         end if
17:       else
18:         List2 = List2 + ( $P_{child}$ , comm_time); // Add  $P_{child}$  to List2
19:       end if
20:     end for
21:   end for
22:   // List2 Convergence
23:   keep_analyzing = false;
24:   repeat
25:     for each process  $P_i$  in List2 do
26:       if exist  $P_x$  in List2 that communicates with  $P_i$  in time  $t$  such that:
    $P_x$ .block_time <  $t$  <  $P_i$ .block_time then
27:         UpdateList2( $P_i$ ,  $t$ ); // Update  $P_i$  block time
28:         keep_analyzing = true;
29:       end if
30:     end for
31:   until not keep_analyzing
32:   ListGlobal = ListGlobal + L1; // group the processes already analyzed
33:   List1 = List2; // List2 levels up
34:   List2 = {}; // List2 is emptied to incorporate the processes of a new
   level
35: end while
```

4.2.4 Secondary effects of the frequency change in the compute phase

When a node changes its clock frequency to a lower frequency, all the processes running on that node are slowed down. The time in which communication takes place between these processes and other non-slowed (or differently slowed) processes will be affected. This could result in an error in the estimation of the compute and waiting phases of these last processes, and a suboptimal application of the strategies. One possible solution is to evaluate processes in the same order that blocking propagation occurs, resulting in a slowdown propagation. For simplicity, the simulator does not implement it.

4.2.5 Secondary effects of sending a node to sleep

When using non-blocking MPI operations (see Subsection 3.2), sending to sleep a node may affect application execution. For example, consider the following situation. A process issues a non-blocking *send*, and then its node is sent to sleep by the application of the strategy. Later, another process emits the corresponding *receive*, being not able to receive the data that is in the buffer of the sending process because the node of that process is sleeping. This causes an unexpected wait because if the strategy had not been applied, the node would be awake and the communication would have been done. One possibility is, in these cases, not sending the node to sleep. To do this, it would be necessary to keep a registry of the pending non-blocking communications. Another option could be to wake up the node when required. In the latter case, if active waits are used (see Subsection 3.3), and the remaining wait time justifies it, the clock frequency should be changed to the minimum available. That is, there would be a change of strategy in the middle of a waiting phase.

5 Energy model

The model presented below estimates the energy savings obtained when the best strategy is applied. The input of the model is application, fault tolerance and system data, and the estimated duration of the compute and waiting phases of each process, as indicated in Table 2. To calculate the energy consumption during an interval of time we need to know the interval duration and the associated average power dissipation. Power and time can be obtained from characterizations as in [1]. For simplicity, we consider the computation to be homogeneous, i.e. the application dissipates the same power throughout its entire execution. The application communication pattern can be obtained from the execution trace [27]. The duration of the compute and wait phases (see subsection 4.2) have to be estimated at failure time. Table 3 details model parameters for reference. Case B of Fig. 2 reflects the situation where no strategies are applied and serves as a reference.

The energy saving obtained with the application of the strategies is estimated as the sum of the savings in each node, as shown in Eq. (1), where n is the number of nodes where the application is executed.

$$Total_Energy_Saving = \sum_{j=1}^n Energy_Saving_{node-j} \quad (1)$$

Table 2: Energy model inputs

System data	Power and time required to sleep and to wake up a node, downtime.
Application data	Power dissipated and slowdown factor for each frequency during the computation. Pattern and frequency of communication among processes.
Fault tolerance data	Checkpoint and restart duration. The power dissipated and the slowdown of each frequency during checkpoint
Variables	Compute and wait phase duration, number of checkpoints during intervention interval.

The energy savings of each node (Eq. 2) is calculated as the difference between the estimated energy consumption without intervention, ENI (Energy No Intervention) and the minimum estimated energy consumption with intervention, EI (Energy with Intervention).

$$Energy_Saving_{node_j} = ENI(f_{max}) - Min\{EI(f_i)\} \quad (2)$$

The energy consumption without intervention, ENI , is shown in Eq. 3. This equation is instantiated with the maximum available clock frequency.

$$ENI(f_{max}) = E_{comp}(f_{max}) + E_{awake_wait}(f_{max}) \quad (3)$$

The energy consumption with intervention, EI , is shown in Eq. 4. This equation is instantiated with each of the clock frequencies, f_i . Once we have all the estimations using all the frequencies, the lowest one is selected (as indicated in Eq. 2).

$$EI(f_i) = E_{comp}(f_i) + EI_{wait}(f_i) \quad (4)$$

In Eq. 3 and 4, the energy consumed by the node is estimated as the sum of the energy consumed during the computation phase plus the energy consumed during the waiting phase.

In all the equations, the clock frequency (indicated as an argument) refers to the frequency at which the computation phase will be executed. In particular, in the equations referring to the waiting phase, this argument is necessary to estimate the duration of this phase (when the computation phase is executed with that frequency). Different frequencies in the computation phase can cause different actions to be applied in the waiting phase.

5.1 Compute phase energy

To estimate the energy consumed during the computation phase we use Eq. 5.

$$E_{comp}(f_i) = T_{comp}(f_i) \times P_{comp}(f_i) + N_{ckpt} \times T_{ckpt}(f_i) \times P_{ckpt}(f_i) \quad (5)$$

In this equation, the duration of the compute phase is calculated for the selected frequency using the slowdown factor $beta$, as indicated in Eq. 6.

$$T_{comp}(f_i) = T_{comp}(f_{max}) \times \beta(f_i) \quad (6)$$

The slowdown factor can be characterized as in Table 4 and indicates how much slower the application runs with frequencies lower than the maximum one. The duration of the possible checkpoints that are performed in this stage is calculated in the same way (Eq. 7).

$$T_{check}(f_i) = check_time \times \gamma(f_i) \quad (7)$$

The number of checkpoints that are carried out during the intervention interval (usually one or none) is an input to the model. The power dissipated with each frequency, $P_{comp}(f_i)$, is an input to the model.

5.2 Waiting phase energy

To estimate the energy consumed during the waiting phase without intervention (ENI) we use Eq. 8, 9 and 10.

$$E_{awake_wait}(f_i) = \begin{cases} E_{active_wait}(f_i) & \text{if active wait} \\ E_{idle_wait}(f_i) & \text{if idle wait} \end{cases} \quad (8)$$

$$E_{active_wait}(f_i) = T_{wait}(f_i) \times P_{active_wait}(f_i) \quad (9)$$

$$E_{idle_wait}(f_i) = T_{wait}(f_i) \times P_{idle_wait} \quad (10)$$

In Eq. 8, the energy consumption will be determined by the message waiting configuration. If the configuration indicates that active waits are used (Eq. 9), the energy consumed is calculated using the power dissipated by the corresponding frequency. On the other hand, if idle waits are used (Eq. 10), the processor is practically idle, and the energy consumed is calculated using a power that is close to the *base power*. We call *base power* the power dissipated when the node is in Working G0 state, i.e. no jobs are running and most of the cores are in inactive states (see Table 1).

To estimate the energy consumed during the waiting phase with intervention (EI) we use Eq. 11. This equation distinguishes two cases, depending on whether the node goes to sleep or stays awake.

$$EI_{wait}(f_i) = \begin{cases} E_{awake_wait}(f_i) & \text{if no sleeping} \\ E_{sleep_wait}(f_i) & \text{if sleeping} \end{cases} \quad (11)$$

If the node stays awake we use the Eq. 8, 9 and 10 described above. If the node goes to sleep, the Eq. 12 is used. This equation estimates the energy considering the power dissipated during the transitions between the active-sleep-active states and while the node is sleeping.

$$EI_{sleep_wait}(f_i) = T_{go_sleep} \times P_{go_sleep} + T_{sleep}(f_i) \times P_{sleep} + T_{wakeup} \times P_{wakeup} \quad (12)$$

The energy consumed to sleep and wake up the node will depend on the state of the node. For simplicity, we use a single value. To sleep a node, two conditions must be met. First, the waiting time must be greater by a certain factor, μ_1 , than the total time that the node requires to sleep and wake up. This is to prevent sleeping a node for a period shorter or equal to the time it takes to put it to sleep and to wake it up. This condition is expressed in Eq. 13.

$$T_wait(f_i) > \mu_1 \times (T_go_sleep + T_wakeup) \quad (13)$$

Secondly, the energy consumption while sleeping (including the energy consumed going to sleep and waking up) must be lower, by a certain factor μ_2 , than the consumption obtained if the node remains awake. This is to avoid sleeping a node that will have almost the same power savings as staying awake. This condition is expressed in Eq. 14.

$$E_sleep_wait(f_i) < \mu_2 \times E_awake_wait(f_{min}) \quad (14)$$

6 The simulator

We have developed an event-based simulator that uses the SMPL library [28] written in C language. This simulator allows us to evaluate the strategies under different system configurations, different characteristics of the application and different failure times. The main features of the simulator are detailed below:

- The failure of a node in a parallel message-passing application with uncoordinated checkpoints at the system level is simulated. For simplicity, a single process per node is simulated; in particular the most representative process of the node.
- We refer to the representative process of a node as the one that first blocks due to the failure. This simplification is based on the fact that if the processes are assigned to a node by affinity, the communication blocks of all the processes in the node will be at a similar time to that of the simulated process.
- In the same line as the previous point, the strategy selected when evaluating the representative process is applied to the node.
- Checkpoints can be triggered by events or by time; as we seek to simulate transparent checkpoint to the application, we activate it by time.
- The moving ahead of checkpoints is simulated (see subsection 3.4).
- The checkpoint files are stored in a parallel file system external to the nodes.
- The message log and the domino effect have not been considered.
- The messages have a fixed size.
- The overhead caused by the strategies evaluation and implementation is not considered.

Table 3: Parameters

Parameter Name	Description
$E_{comp}(f_i)$	Energy consumed by the node during the computing phase, at frequency f_i .
$E_{awake_wait}(f_i)$	Energy consumed by the node during the waiting phase when it remains awake.
$E_{I_wait}(f_i)$	Energy consumed by the node during the waiting phase when the intervention takes place.
$E_{active_wait}(f_i)$	Energy consumed by the node during the active waiting phase at frequency f_i .
$E_{idle_wait}(f_i)$	Energy consumed by the node during the waiting phase when using idle wait.
$E_{sleep_wait}(f_i)$	Energy consumed by the node during the waiting phase when it goes to sleep.
$T_{ckpt}(f_i)$	Checkpoint duration running at frequency f_i .
$T_{comp}(f_i)$	Computation phase duration when node executes at the frequency f_i .
$T_{go_sleep} T_{wakeup}$	Times required by a node to sleep and wake up, respectively.
$T_{sleep}(f_i)$	Time that node is sleeping when compute phase is executed at frequency f_i , without considering the time to go to sleep and to wake up.
$T_{wait}(f_i)$	Waiting phase duration when computation phase is executed at frequency f_i .
T_{ckpt}	Checkpoint duration.
$P_{go_sleep} P_{wakeup}$	Power dissipated while sleeping and waking up a node, respectively.
P_{sleep}	Power dissipated when the node is sleeping.
$P_{comp}(f_i)$	Power dissipated during computation when running at frequency f_i .
$P_{ckpt}(f_i)$	Power dissipated during checkpoint at frequency f_i .
$P_{active_wait}(f_i)$	Power dissipated during active wait at frequency f_i .
P_{idle_wait}	Power dissipated during the idle wait.
N_{ckpt}	Number of checkpoints during the computation phase (zero or one).
$\beta(f_i) \gamma(f_i)$	Slowdown of instruction and checkpoint execution when frequency f_i is used.
$\mu_1 \mu_2$	Time and energy threshold to determine whether or not to sleep a node.

Table 4: Power and slowdown at different clock frequencies

Frequency (GHz)	Application		Checkpoint	
	Average Power (W)	β	Average Power (W)	γ
2.8	166	1	150	1
2.1	148	1.2	142	1.1
1.7	139	1.5	131	1.2
1.2	126	2.1	125	1.4

- The simulated MPI functions are blocking and non-blocking standard mode (see subsection 3.2).

The simulator inputs are the same as the energy model described in subsection 5, and are detailed in Table 2. At the time of failure, the simulator evaluates the model for each surviving process with each of the clock frequencies provided, and determines the best strategy to apply. The simulator output includes the estimated energy savings when applying the selected strategy, and a trace to visualize the behavior of the application execution. The trace is visualized with the Paraver⁴ tool, a flexible HPC application performance analysis and visualization tool.

7 Experimentation and results analysis

The following subsections describe the experimental work carried out with the simulator. The proposed strategies seek to exploit different opportunities to reduce the energy consumption of the applications affected by the failures. The experiments present specific scenarios to show the validity of the proposal. The results obtained are analyzed and some discussions are at the end of the section.

7.1 Experimental settings

Table 4 shows dissipated power and slowdown factor (β and γ) obtained from measurements on a six-core Intel Xeon E5-2630 node, with a frequency range of 1.2 GHz to 2.8 GHz (with the mechanism Intel Turbo Boost disabled). The base power is 60W. The node sleep and wake times are set at 25 and 5 seconds respectively, and the average powers at 51 and 91 watts respectively. The average power dissipated while the node is sleeping is 12 watts. The checkpoint duration is set to two minutes, and the MPI waits are configured as active waits, except otherwise indicated. The scenarios present four processes (or nodes) and the failing node is the one that hosts process 0.

7.2 Experiments and results analysis

Different simulated scenarios are discussed in this subsection. For each scenario, the following data are shown:

- A table with the particular configuration data of the experiment, as in Table 5.

⁴<http://www.bsc.es/computer-sciences/performance-tools/paraver>

- A table with the actions selected and the energy savings estimations (as in Table 6): In this table, column N indicates the node number, the *Action* column indicates the strategy applied, column T indicates the phase duration, and column TT is the total duration. The last columns show the savings in joules, joules per second, and percent ($Save(J)$, $SaveRate(J/s)$, and $Save(\%)$ respectively). The percent column is calculated by dividing the estimations of the joules saved by the joules used without intervention (ie without applying any action). This metric is useful in an intra-scenario analyses. To compare different scenarios, J/s is preferred.
- The trace graphic: These graphics show the states of each process, the communications between processes, and the time at which any of the corresponding strategies are applied to them. Fig. 5 indicates the states, communication lines and flags in trace graphics. The red blocks indicate communication blocking or wait (sometimes imperceptible due to the zoom applied to the trace). The flags indicate the beginning and the end of the strategy application in that node. If the strategy applied is to sleep the node, the wait is indicated in gray. Fig. 6 shows a communication operation between two processes, which is indicated by a yellow line that joins both processes. The inclination of the line (when appreciated) allows us to see who is the sender and who is the receiver. In blocking communications, Fig. 6 (a), the line goes from the beginning of the send to the end of the receive. In non-blocking communications, Fig. 6 (b), the line goes from the beginning of the send to the end of the receiver’s wait operation (MPI_Wait). In Fig. 6 (b), the wait in the receiver is short (the red block is thin) because the wait operation was successful; otherwise, it would be longest.

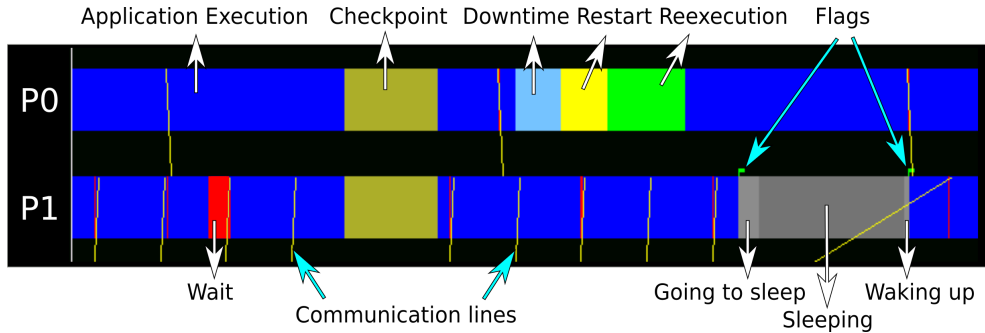


Figure 5: States, communication lines and flags in trace graphics

7.2.1 Scenario 1: Short vs. long re-execution time

In this scenario, we analyze a case with a long re-execution time compared to one where the re-execution time is short. Table 5 shows the simulation settings. In the short re-execution time case (Fig. 7a), the moment of the failure is configured to occur immediately after a checkpoint. The frequency is changed to 2.1 GHz during the computation phase in all processes because lower frequencies increase the synchronization time with the recovering process, and strategies that affect the total execution time are not applied. As this scenario is configured with

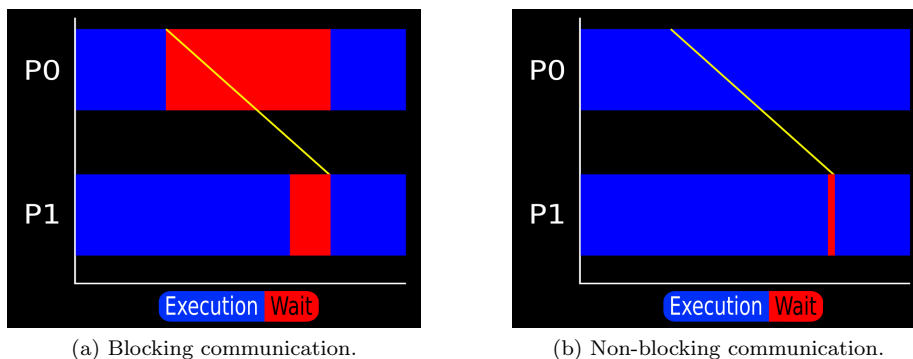


Figure 6: Communications in trace graphics

Table 5: Simulation parameters for Scenario 1: Short vs. long re-execution time

Communication interval	21.6 min.
Communication pattern	Process P0 sends and receives messages from processes P1, P2, and P3.
MPI Waits	Active
MPI operations	Blocking

active waits, and the wait duration is long enough to justify a frequency change (but not to send to sleep), it is changed to the minimum frequency to save energy. With these actions, the three nodes would achieve an energy saving of 14% in a time interval of 13.4 minutes

The long re-execution time case can be seen in Fig. 7b. In this case, the time of failure has been set far from the checkpoint time. As the waiting phases of the three surviving processes are very long, the three nodes are sent to sleep at this phase. Since the nodes will sleep in the waiting phase, it is convenient to arrive at this phase as soon as possible, to obtain greater energy savings. In this scenario, the clock frequencies different from the maximum one implies a longer compute phase, and therefore a shorter waiting phase. This leads to fewer energy savings and this is the reason why the clock frequency is not changed in the compute phase. With these actions, the intervened nodes would be able to consume almost 86% less energy during the intervention interval, which is around 60 minutes. This scenario achieves better results than the previous one due to its long waits where the nodes go to sleep. Figs. 7a and 7b are at different scales to better appreciate each case. Table 6 shows the results obtained.

7.2.2 Scenario 2: Blocking vs. non-blocking MPI operations

In this scenario, we compare the use of blocking and non-blocking operations. Table 7 shows the configuration of the simulation. Fig. 8a shows the use of blocking operations. In this figure, after the failure, process P0 cannot send data to process P1. Without this data, process P1 cannot continue its computation and gets blocked. At this moment process P1 goes to sleep, achieving a saving of 72% in almost 5 minutes.

Fig. 8b shows the use of non-blocking operations. In this case, after the failure, process P1 can complete its communication with Process P0 (the *wait*

Table 6: Selected actions and energy savings for scenario 1: Short vs. long re-execution time

N	Compute phase		Wait phase		TT (m)	Save (J)	Save Rate(J/s)	Save (%)
	Action	T (m)	Action	T (m)				
Short re-execution time								
1	2.1 GHz	12.07	1.2 GHz	1.32	13.39	18,704.5	23.28	14.03
2	2.1 GHz	12.07	1.2 GHz	1.32	13.39	18,705.56	23.28	14.02
3	2.1 GHz	12.07	1.2 GHz	1.32	13.39	18,706.06	23.28	14.02
Long re-execution time								
1	No action	4.37	sleep	56.00	60.37	516,084.73	153.59	85.82
2	No action	4.37	sleep	56.00	60.38	516,085.34	153.59	85.82
3	No action	4.38	sleep	56.00	60.38	516,084.69	153.59	85.82

Table 7: Simulation parameters for scenario 2: Blocking vs. non-blocking MPI operations

Communication interval	5 min.
Communication pattern	Processes P0 and P2 send messages to process 1. Process P3 sends messages to process P2
MPI Waits	Active
MPI operations	Blocking

is successful) because it had been done just before the failure. Then, it continues computing and blocks at the next *wait*. This results in a longer computational phase when compared to the previous case. As can be seen in Figure 8b, process P1 changed the frequency in its computation and waiting phase (indicated by the green flags in the figure), achieving a saving of almost 30% in an interval time similar to the previous case (4 minutes and a half).

This difference in power savings is because the node sleeps in the first case (use of blocking operations), while it changes clock frequency in the second case (use of non-blocking operations). Table 8 shows the results obtained.

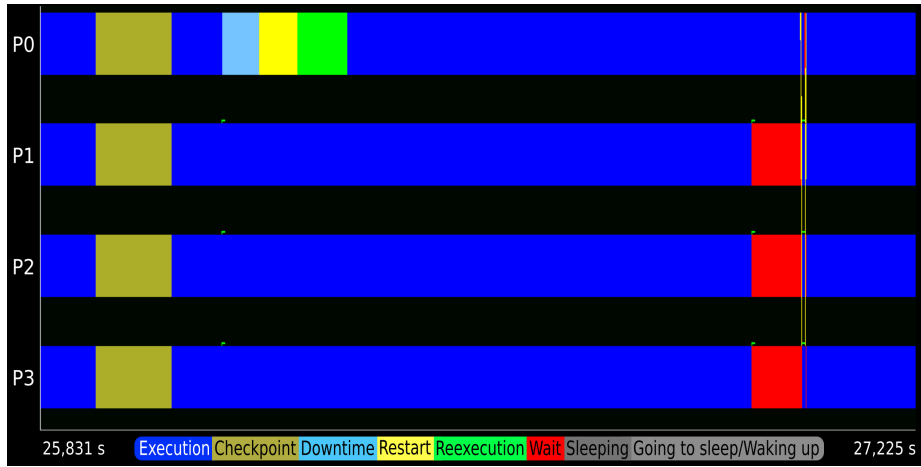
7.2.3 Scenario 3: Active waits vs. idle waits

In this scenario, we compare the use of active waits against idle waits (see subsection 3.3). Table 9 shows the simulation configuration. In both cases, the actions selected for the computation phase are to change the frequency. In the active wait case (Fig. 9a), the action selected for the waiting phase is to change the frequency to the minimum one, indicated by the green flag at the beginning of the phase. In the idle wait case (Fig. 9b), there is no action in the waiting phase.

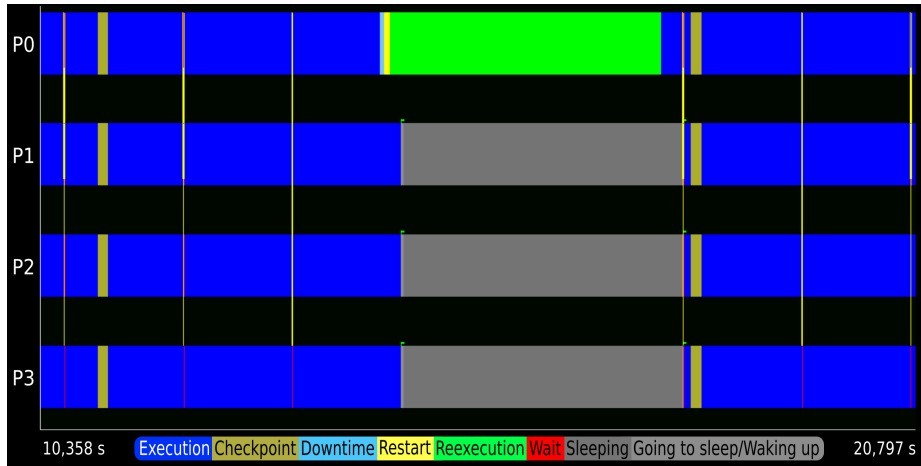
In the first case (active waits), the energy savings are 36%, against the 0.09% for the second case (idle waits), in the same interval of time. This shows the positive impact of the application of the strategy on active waits. Even when the scenario presents short waiting times, if the system is configured with active waiting, the use of the strategies achieves considerable energy savings. Table 10 shows the results obtained.

7.2.4 Scenario 4: With and without system buffers

In this scenario, we analyze the effects of using or not using system buffers during communication operations (see subsection 3.2). Table 11 shows the simulation



(a) Short re-execution time.



(b) Long re-execution time.

Figure 7: Scenario 1

configuration. As indicated in this table, processes 1, 2, and 3 send data to process 0.

Fig. 10a shows the case of non-blocking operations without buffering. As it can be seen, after the failure, the next *wait* operation of the non-failing processes is successful because the communication had taken place just before the failure. On the next *wait* operation, all three processes get blocked because the receiving process (Process P0) is still re-executing. The selected strategy for the three processes is to keep the frequency in the compute phase and sleep in the waiting phase. The observed energy saving is approximately 30% in an interval of 17 minutes.

The system buffers store the sent messages so that several *send* operations (in the case of blocking operations) or several *wait* operations (in the case of non-blocking operations) can be issued successively without blocking. Fig. 10b shows a simulation where system buffering is used. As it can be seen, the chosen

Table 8: Selected actions and energy savings for scenario 2: Blocking vs. non-blocking MPI operations

N	Compute phase		Wait phase		TT (m)	Save (J)	Save Rate(J/s)	Save (%)
	Action	T (m)	Action	T (m)				
Blocking operations								
1	No action	0.86	sleep	3.67	4.53	32,502.30	147.77	72.06
Non-blocking operations								
1	2.1 GHz	2.07	1.2 GHz	2.15	4.22	11,468.73	45.30	27.29

Table 9: Simulation parameters for scenario 3: Active waits vs. idle waits

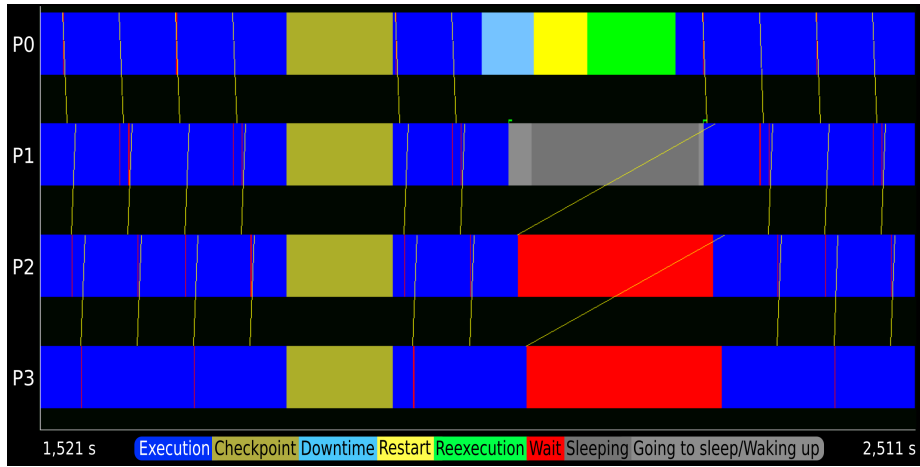
Communication interval	60 sec.
Communication pattern	Process P0 sends and receives messages from processes P1, P2, and P3.
MPI Waits	Active and non-active
MPI operations	Blocking

communication pattern (living processes send data to the failing process) avoids the blocking of live processes, which are not affected by the failure, and therefore do not receive the application of any of the strategies.

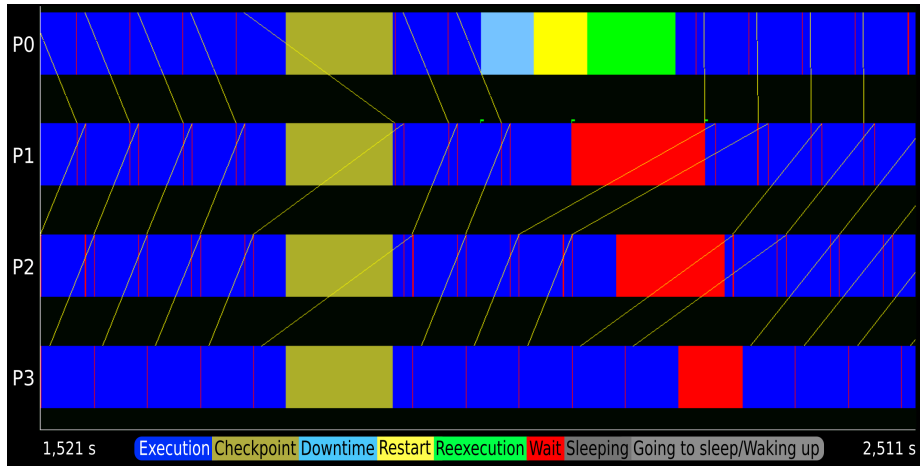
Although the proposed experiment is quite atypical, it serves to magnify and observe the impact that the presence or absence of system buffering has on communications operations when an application becomes desynchronized due to the failure of a node. This is the reason why the simulator is able to enable or disable the use of the system buffering in communication operations. In a typical message-passing application, the processes usually alternate between send and receive operations, so that if the *send* is not blocked (due to the use of the system buffer), the *receive* will be blocked, and the processes can receive the application of the strategies. Beyond the savings achieved in one case or another, the use of the buffer allows the processes to advance in doing useful work, which has a positive impact on energy efficiency. Table 12 shows the results obtained.

7.2.5 Scenario 5: With and without analysis of cascade-blocked processes

This scenario analyzes the effect of incorporating cascade-blocked processes (see section 4) into the processes that receive the application of the strategies. Table 13 shows the configuration of the simulation. Fig. 11a shows a simulation that does not analyze cascade blocked processes. As it can be seen, process P1 blocks with process P0 (failing process) and receives the application of a strategy (sleep). However, processes P2 and P3 get also blocked because of the failure, but in an indirect manner. By including cascade-blocked processes in the analysis, we can note that process P2 communicates with process P1 four times before it get blocked, so we need to raise the depth level to 5 (see section 4 for a definition of *depth*). Fig. 11b shows that the blocking state of process P2 is now detected, and the strategy selected in this case is to send the node to sleep. When this blocking state is included in the analysis, the blocking state of process P3 is also detected, which is cascade-blocked with process P2. The strategy applied to process P3 is to change the frequency during the waiting phase. Although process P3 blocks on the third communication with process



(a) Blocking operations.



(b) Non-blocking operations.

Figure 8: Scenario 2

P2, and a depth of 3 would have been enough, it was necessary to increment the depth to 5 to include the mentioned blocking state in the analysis (because of its relation with process P2).

By including the cascade-blocked processes in the simulator analysis, we cover from a scenario where only one process takes an action to improve energy efficiency (Fig. 11a), to another where the three living processes do the same (Fig. 11b). Analyzing the cascade-blocked processes, allow us to increase energy savings from 32,500 J to almost 80,000 J in intervals between 9 and 14 minutes. Table 14 shows the results obtained.

7.2.6 Scenario 6: With and without checkpoint anticipations

This scenario analyzes the impact of bringing forward checkpoints. Table 15 shows the configuration of the simulation. This scenario compares two cases

Table 10: Selected actions and energy savings for scenario 3: Active waits vs. idle waits

N	Compute phase		Wait phase		TT (m)	Save (J)	Save Rate(J/s)	Save (%)
	Action	T (m)	Action	T (m)				
Active waits								
1	2.1 GHz	0.64	1.2 GHz	2.56	3.20	11,673.84	60.78	36.61
2	2.1 GHz	0.64	1.2 GHz	2.56	3.20	11,674.89	60.75	36.60
3	2.1 GHz	0.65	1.2 GHz	2.56	3.21	11,675.40	60.72	36.58
Non-active waits								
1	2.1 GHz	0.64	No action	2.56	3.20	12.83	0.33	0.09
2	2.1 GHz	0.64	No action	2.56	3.20	12.87	0.33	0.09
3	2.1 GHz	0.65	No action	2.56	3.21	12.92	0.33	0.09

Table 11: Simulation parameters for scenario 4: With and without system buffers

Communication interval	5 min.
Communication pattern	Processes P1, P2, and P3 send messages to process P0.
MPI Waits	Active
MPI operations	Non-blocking

with a long re-execution time.

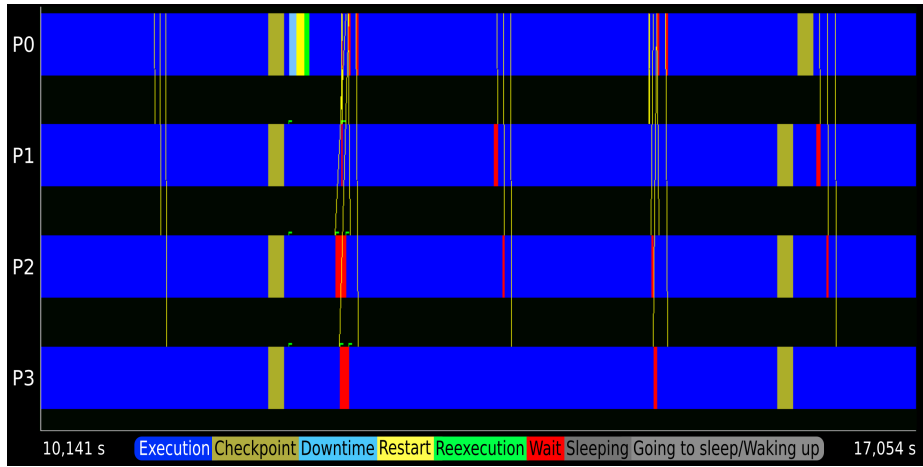
Fig. 10 a) and b) show the cases when the advanced checkpoint anticipations is enabled and disabled in the simulator, respectively.

Figs. 12a and 12b show the cases when checkpoint anticipation is enabled and disabled in the simulator, respectively. The selected strategy in both cases is to maintain the maximum frequency in the compute phase and sleep the nodes in the waiting phase. In the first case, before entering the waiting phase, checkpoints are anticipated. This causes a shorter waiting phase than the second case. The results show slightly higher energy savings in the case where no checkpoint is advanced, (85.8% versus 83%). Although there is a slightly improves in the first case (without advanced checkpoints), in the second case (with advanced checkpoints) it avoids stopping the application later when it might be doing useful computation. Table 16 shows the results obtained.

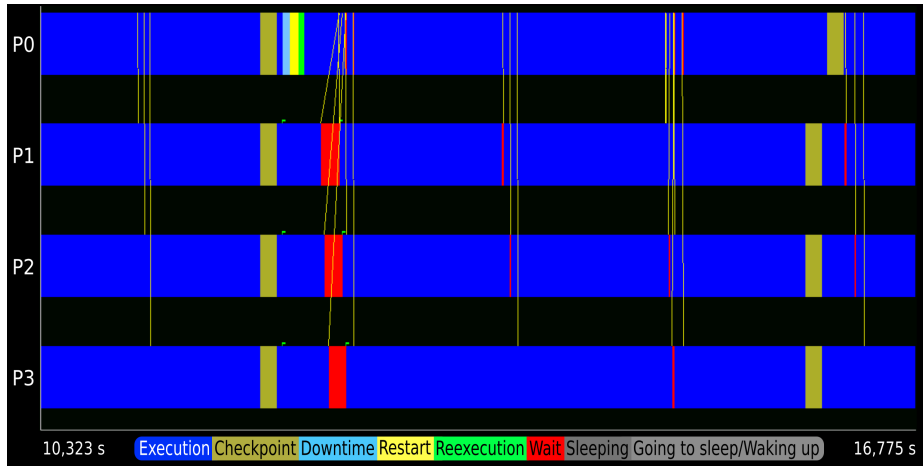
7.2.7 Scenario 7: Matrix Multiplication

In this scenario, we analyze an MPI master-worker matrix multiplication application ($C = A \times B$), because it is well-known, computationally intensive, and representative application of scientific computing. The Table 17 shows the configuration of the simulation. The test application does successive multiplication of different A_i submatrices with B to obtain resultant C_i . To achieve this, the master process broadcast matrix B to the worker processes. After that, the master process distributes parts of A_i , the worker processes compute the multiplication, and return the partial result to the master process. When all the worker finish this stage, the application enter the next one, where the master distributes another part of A_i until completing C .

The application was characterized with the pas2p tool [27], considering matrices of 2048×2048 elements (of type double) and 4 processes/nodes. As a result, the communication intervals between processes were obtained.



(a) Active waits.



(b) Idle waits.

Figure 9: Scenario 3

Three different cases are analyzed: The first case shows a long re-execution time with blocking operations, the second case shows a short re-execution time with blocking operations, and the third case shows a short re-execution time with non-blocking operations (the graphics has different time scales). The Table 18 shows the results obtained for these cases.

In Fig. 13, the case of the long re-execution time from the checkpoint is presented. In this case, the nodes corresponding to the three live processes are sent to sleep, obtaining an estimated saving of 91% (144.929 J), for an intervention time of approximately 15 minutes. This means that, in the 15 minutes that the failed process needs to recover, the energy consumption would be 91% lower than the consumption obtained without applying any of the strategies.

If, on the other hand, the failure occurs near the checkpoint (Fig. 14), the intervention interval is reduced to two and a half minutes, achieving a saving of around 43% (about 10,000J, against almost 145,000J of the previous case).

Table 12: Selected actions and energy savings for scenario 4: With and without system buffers

N	Compute phase		Wait phase		TT (m)	Save (J)	Save Rate (J/s)	Save (%)
	Action	T (m)	Action	T (m)				
Without system buffer								
1	No action	11.25	sleep	6.27	17.51	56549.4	150.36	32.78
2	No action	11.26	sleep	6.26	17.52	56426.2	150.35	32.7
3	No action	11.28	sleep	6.24	17.52	56303	150.34	32.62

Table 13: Simulation parameters for scenario 5: With and without analysis of cascade-blocked processes

Communication interval	5 sec.
Communication pattern	Processes 0 and 2 send messages to process 1, and process 3 sends messages to process 2.
MPI Waits	Active
MPI operations	Blocking

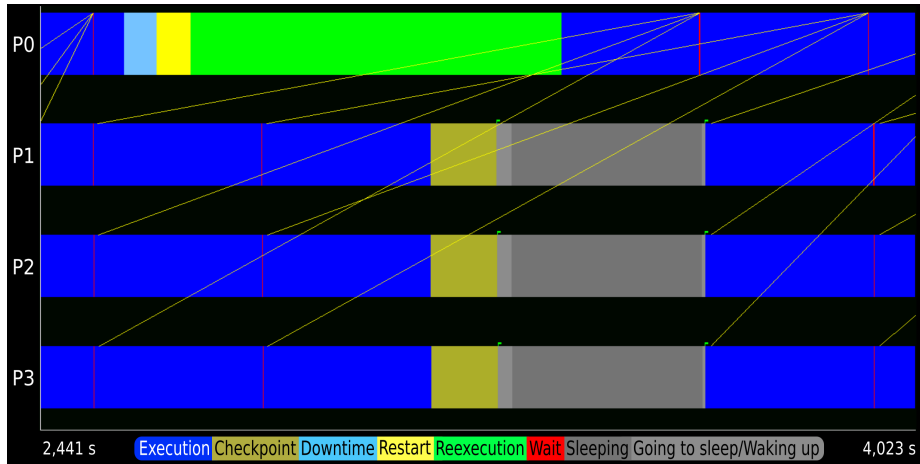
The selected strategy in this case was to change the frequency of live nodes in its computation and waiting phase.

Fig. 15 shows the case of short re-execution time with blocking operations. The selected strategy was the same as in the previous case, changing the clock frequency in the computation and waiting phases of all live processes. However, in this case, the computation phase lasts 9 seconds, against the 2 seconds for the same phase in the version with blocking operations. The saving obtained in this case is around 41% (9,605J, against the 10,022J of the previous case), in the same time interval (two and a half minutes).

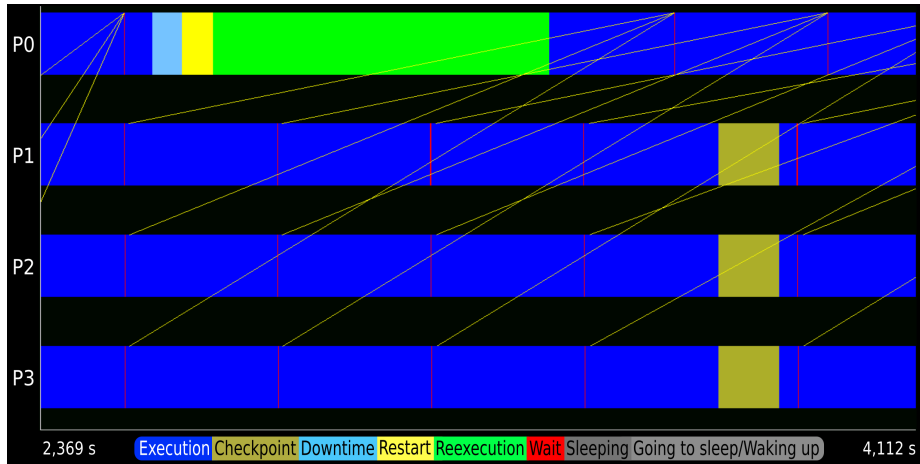
7.3 Discussion

The experimental work with the simulator shows us that the duration of the waiting phase defines the selection of the strategy. If the wait is long enough to send the node to sleep, this will be the chosen option, since the energy savings obtained exceed by far those obtained with any combination of the other strategies. Furthermore, when the strategy of sleeping a node is selected, the model that the simulator implements always selects, for the compute phase, the lowest clock frequency that does not produce an increase in execution time. In this way, the wait is not shortened, and energy savings are maximized. Moreover, if the failure occurs near the checkpoint time, the wait may be short, and in these cases, the duration of the compute phase plays a fundamental role.

In SPMD or non-dynamic Master-Worker applications where blockings propagate rapidly, it might be counterproductive to change the clock frequency in the computation phase to sleep longer during the waiting phase. In applications with loosely coupled communication patterns, where one group of tasks communicates infrequently with another group of tasks, there can be frequency change during the compute and waiting phases, but this will also depend on whether the re-execution (of the recovering processes) is short.



(a) Without system buffers.



(b) With system buffers.

Figure 10: Scenario 4

8 Conclusions and future work

In this work, we have enriched our previous energy model by including non-blocking communications (with and without system buffering) and the cascade-blocking effect. Next, we have extended the simulator by incorporating the new features of the energy model, to evaluate their potential benefits.

While implementing the proposal some issues were identified as being important:

- The *depth* of communications, that is, the number of successful communications that a process can perform before it gets blocked by another living process, due to collateral effects of the failure.
- The slowdown propagation when the clock frequency is changed in the compute phase (in a context with cascade-blocked processes included in

Table 14: Selected actions and energy savings for scenario 5: With and without analysis of cascade-blocked processes

Node	Compute phase		Wait phase		TT (m)	Save (J)	Save Rate (J/s)	Save (%)
	Action	T (m)	Action	T (m)				
Without analysis of cascading blocked processes								
1	No action	4.84	sleep	3.67	8.51	32,517.7	147.77	38.36
With analysis of cascading blocked processes								
1	No action	4.51	sleep	4.00	8.51	35,597.70	148.29	42.00
2	No action	4.93	sleep	4.00	8.93	35,636.20	148.30	40.03
3	No action	12.00	1.2 GHz	02.02	14.02	8,655.07	71.5	6.29

Table 15: Simulation parameters for scenario 6: With and without checkpoint anticipations

Communication interval	4 sec.
Communication pattern	Process P0 sends and receives messages from processes P1, P2, and P3.
MPI Waits	Active
MPI operations	Blocking

the analysis).

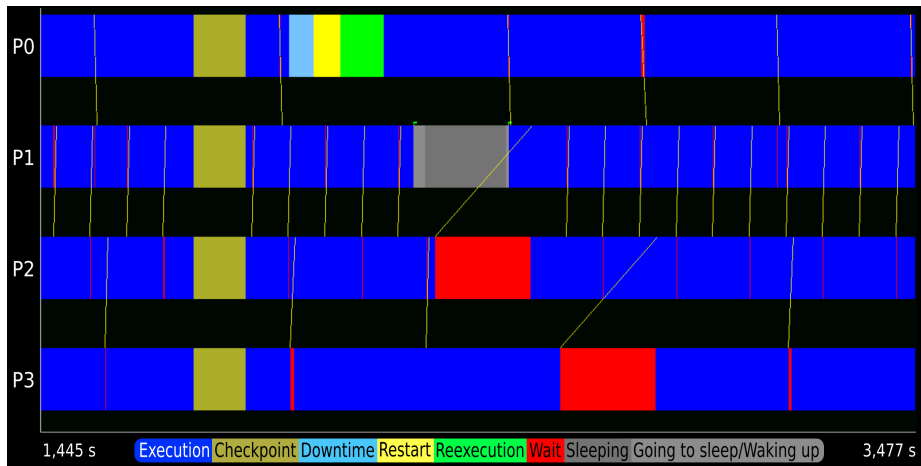
- The need to wake up a node early, because a process on another node needs it awake to continue its execution.

Through the experimental work, we were able to discover some opportunities to reduce energy consumption in these scenarios. The simulations show that the savings were negligible in the worst case, but in some scenarios, it was possible to achieve significant ones; the maximum saving achieved was 90% in an execution time of 16 minutes. More generally, we can conclude that:

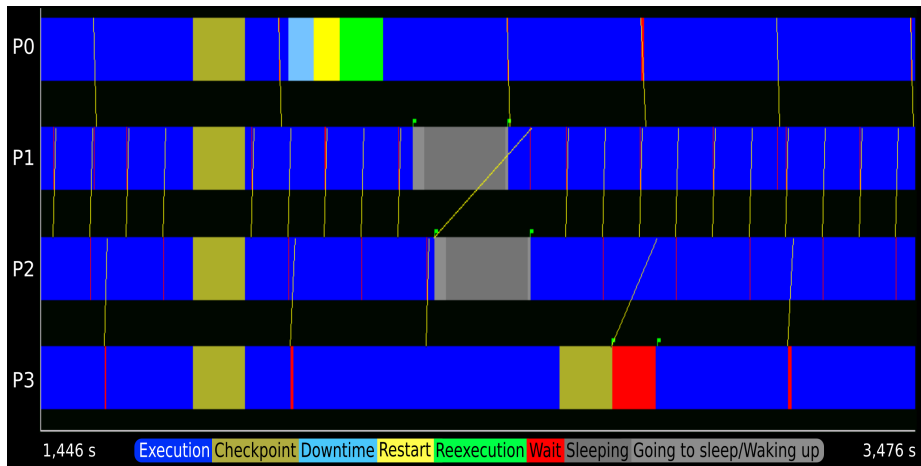
- Non-blocking operations and the use of system buffering in communications present fewer opportunities for the application of strategies because they avoid blocking states and permit the computation to proceed.
- Incorporating cascade-blocked processes shows significant increases in energy savings even when just a few nodes are considered.
- In applications where communication is unusual, the strategies for the compute phase play a key role (since these phases are long).
- In applications where compute phases are short, the duration of the waiting phase defines the selection of the strategy. In particular, if the failure occurs far from the last checkpoint time, the waiting phases will be long and the node should probably be sent to sleep.

As a result, we show the feasibility of improving energy efficiency in HPC systems in the presence of a failure. Among future works, we plan:

- To implement the strategies on a real system as a proof of concept, considering the secondary effects described before
- To include runtime characterization of the application.



(a) Without analysis of cascade-blocked processes



(b) With analysis of cascade-blocked processes

Figure 11: Scenario 5

Funding

This research has been supported by the Agencia Estatal de Investigacion (AEI), Spain and the Fondo Europeo de Desarrollo Regional (FEDER) UE, under contract PID2020-112496GB-I00 and partially funded by the Fundacion Escuelas Universitarias Gimbernat (EUG).

References

- [1] M. Morán, J. Balladini, D. Rexachs, and E. Luque, “Prediction of energy consumption by checkpoint/restart in HPC,” *IEEE Access*, vol. 7, pp. 71791–71803, 2019.

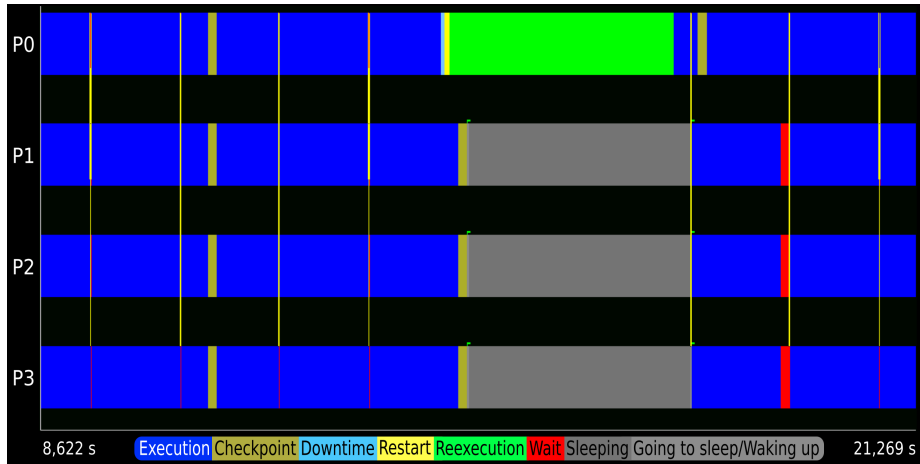
Table 16: Selected actions and energy savings for scenario 6: With and without checkpoint anticipations

Node	Compute phase		Wait phase		TT (m)	Save (J)	Save Rate (J/s)	Save (%)
	Action	T (m)	Action	T (m)				
With moving ahead checkpoints								
1	No action	6.37	sleep	54.00	60.37	497,604.73	153.58	83.02
2	No action	6.37	sleep	54.00	60.38	497,605.34	153.58	83.01
3	No action	6.38	sleep	54.00	60.38	497,604.69	153.58	83.01
Without moving ahead checkpoints								
1	No action	4.37	sleep	56.00	60.37	516,084.73	153.59	85.82
2	No action	4.37	sleep	56.00	60.38	516,085.34	153.59	85.82
3	No action	4.38	sleep	56.00	60.38	516,084.69	153.59	85.82

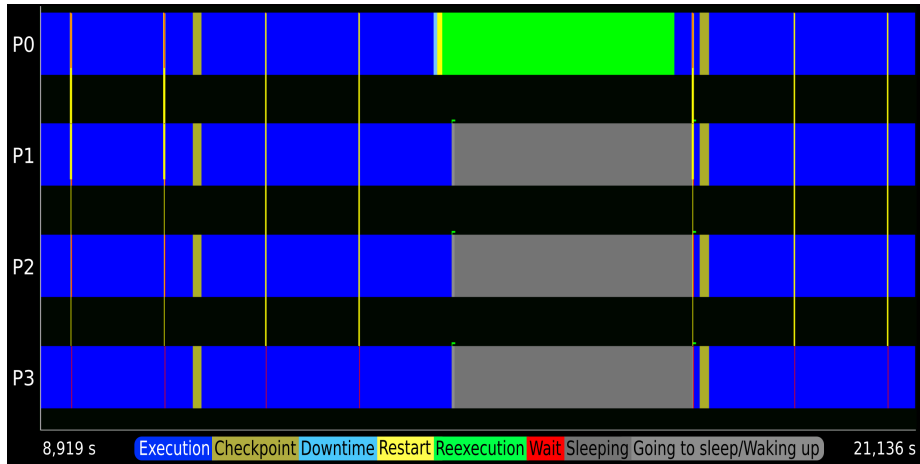
Table 17: Simulation parameters for scenario 7: Matrix multiplication

Communication interval	6.5 sec.
Communication pattern	Process P0 sends and receives messages from processes P1, P2, and P3.
MPI Waits	Active
MPI operations	Blocking (first two cases) and non-blocking (last case)

- [2] M. Morán, J. Balladini, D. Rexachs, and E. Rucci, “Towards management of energy consumption in hpc systems with fault tolerance,” in *2020 IEEE Congreso Biental de Argentina (ARGENCON)*, pp. 1–8, IEEE, 2020.
- [3] M. Morán, J. Balladini, D. Rexachs del Rosario, and E. Rucci, “Some issues to consider in the management of energy consumption in hpc systems with fault tolerance,” in *X Jornadas de Cloud Computing, Big Data & Emerging Topics (La Plata, 2022)*, pp. 17–22, 2022.
- [4] T. Saito, K. Sato, H. Sato, and S. Matsuoka, “Energy-aware i/o optimization for checkpoint and restart on a nand flash memory system,” in *Proceedings of the 3rd Workshop on Fault-tolerance for HPC at extreme scale*, pp. 41–48, ACM, 2013.
- [5] B. Mills, T. Znati, and R. Melhem, “Shadow computing: An energy-aware fault tolerant computing model,” in *2014 International Conference on Computing, Networking and Communications (ICNC)*, pp. 73–77, IEEE, 2014.
- [6] B. Mills, R. E. Grant, K. B. Ferreira, and R. Riesen, “Evaluating energy savings for checkpoint/restart,” in *Proceedings of the 1st International Workshop on Energy Efficient Supercomputing*, pp. 6:1–6:8, ACM, 2013.
- [7] E. Meneses, O. Sarood, and L. V. Kalé, “Energy profile of rollback-recovery strategies in high performance computing,” *Parallel Computing*, vol. 40, no. 9, pp. 536–547, 2014.
- [8] M. el Mehdi Diouri, O. Glück, L. Lefevre, and F. Cappello, “Ecofit: A framework to estimate energy consumption of fault tolerance protocols for hpc applications,” in *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pp. 522–529, IEEE, 2013.
- [9] K. H. R. Rajachandrasekar, A. Venkatesh and D. K. Panda, “Power-check: An energy-efficient checkpointing framework for HPC clusters,” *2015 15th*



(a) With checkpoint anticipations



(b) Without checkpoint anticipations

Figure 12: Scenario 6

IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp. 261–270, 2015.

- [10] M. A. Amrizal and H. Takizawa, “Optimizing energy consumption on HPC systems with a multi-level checkpointing mechanism,” in *2017 International Conference on Networking, Architecture, and Storage (NAS)*, pp. 1–9, IEEE, 2017.
- [11] D. Dauwe, R. Jhaveri, S. Pasricha, A. A. Maciejewski, and H. J. Siegel, “Optimizing checkpoint intervals for reduced energy use in exascale systems,” in *2017 Eighth International Green and Sustainable Computing Conference (IGSC)*, pp. 1–8, IEEE, 2017.
- [12] N. El-Sayed and B. Schroeder, “Understanding practical tradeoffs in HPC checkpoint-scheduling policies,” *IEEE Transactions on Dependable and Se-*

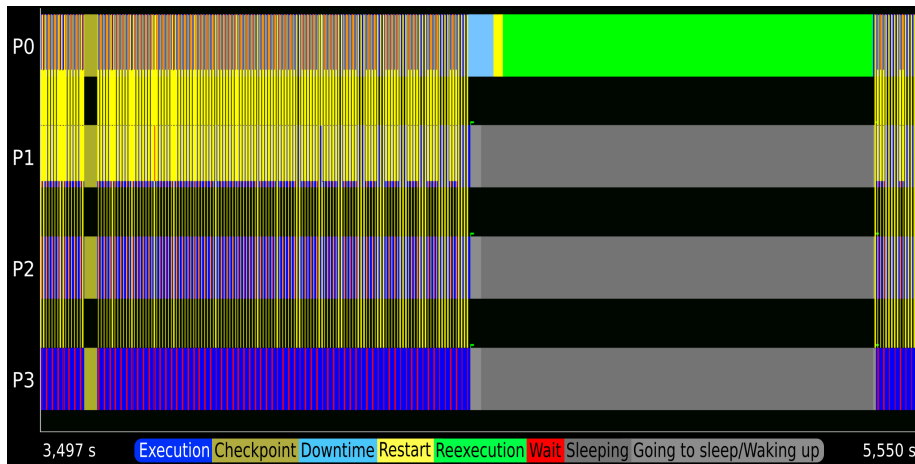


Figure 13: Scenario 7: Matrix multiplication with long reexecution time (blocking operations)

Table 18: Selected actions and energy savings for scenario 7: Matrix multiplication

Node	Compute phase		Wait phase		TT (m)	Save (J)	Save Rate (J/s)	Save (%)
	Action	T (m)	Action	T (m)				
With long reexecution time (blocking operations)								
1	No action	0.09	sleep	15.83	15.92	144,929.96	152.56	91.39
2	No action	0.09	sleep	15.83	15.92	144,929.94	152.56	91.39
3	No action	0.09	sleep	15.83	15.92	144,929.92	152.56	91.39
With short reexecution time (blocking operations)								
1	2.1 GHz	0.03	1.2 GHz	2.33	2.36	10,022.36	70.73	42.61
2	2.1 GHz	0.03	1.2 GHz	2.33	2.36	10,022.40	70.73	42.61
3	2.1 GHz	0.03	1.2 GHz	2.33	2.36	10,022.46	70.72	42.61
With short reexecution time (non-blocking operations)								
1	2.1 GHz	0.16	1.2 GHz	2.20	2.36	9,605.29	67.92	40.92
2	2.1 GHz	0.16	1.2 GHz	2.20	2.36	9,605.62	67.92	40.92
3	2.1 GHz	0.16	1.2 GHz	2.20	2.36	9,605.39	67.92	40.91

cure Computing, vol. 15, no. 2, pp. 336–350, 2018.

- [13] A. Bouteiller, F. Cappello, J. Dongarra, A. Guermouche, T. Héroult, and Y. Robert, “Multi-criteria checkpointing strategies: Response-time versus resource utilization,” in *European Conference on Parallel Processing*, pp. 420–431, Springer, 2013.
- [14] K. Dichev, K. Cameron, and D. S. Nikolopoulos, “Energy-efficient localised rollback via data flow analysis and frequency scaling,” in *Proceedings of the 25th European MPI Users’ Group Meeting*, pp. 1–11, 2018.
- [15] S. Bhalachandra, A. Porterfield, S. L. Olivier, and J. F. Prins, “An adaptive core-specific runtime for energy efficiency,” in *IEEE International Parallel and Distributed Processing Symposium*, pp. 947–956, 2017.
- [16] B. Rountree, D. K. Lowenthal, B. R. De Supinski, M. Schulz, V. W. Freeh, and T. Bletsch, “Adagio: making DVS practical for complex HPC appli-

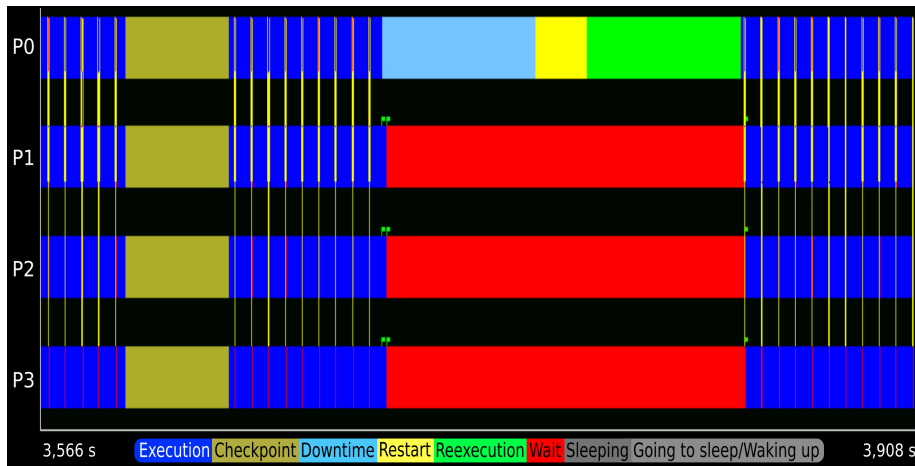


Figure 14: Scenario 7: Matrix multiplication with short reexecution time (blocking operations)

- cations,” in *Proceedings of the 23rd international conference on Supercomputing*, pp. 460–469, 2009.
- [17] X. Wu, V. Taylor, and Z. Lan, “Performance and power modeling and prediction using mummi and ten machine learning methods,” *arXiv preprint arXiv:2011.06655*, 2020.
- [18] S. Wang, H. Chen, and W. Shi, “Span: A software power analyzer for multi-core computer systems,” *Sustainable Computing: Informatics and Systems*, vol. 1, no. 1, pp. 23–34, 2011.
- [19] M. F. Dolz, J. C. Fernández, S. Iserte, R. Mayo, and E. S. Quintana-Ortí, “A simulator to assess energy saving strategies and policies in hpc workloads,” *SIGOPS Oper. Syst. Rev.*, vol. 46, p. 2–9, jul 2012.
- [20] M. Knobloch, B. Mohr, and T. Minartz, “Determine energy-saving potential in wait-states of large-scale parallel programs,” *Computer science-research and development*, vol. 27, no. 4, pp. 255–263, 2012.
- [21] L. Piga, I. Paul, and W. Huang, “Performance boosting opportunities under communication imbalance in power-constrained hpc clusters,” in *2016 45th International Conference on Parallel Processing (ICPP)*, pp. 31–40, IEEE, 2016.
- [22] D. Cesarini, A. Bartolini, P. Bonfà, C. Cavazzoni, and L. Benini, “Count-down: A run-time library for application-agnostic energy saving in MPI communication primitives,” in *Proceedings of the 2nd Workshop on Autotuning and Adaptive Approaches for Energy efficient HPC Systems*, pp. 1–6, 2018.
- [23] S. Levy, B. Topp, K. B. Ferreira, D. Arnold, P. Widener, and T. Hoefler, “Using simulation to evaluate the performance of resilience strategies and process failures,” *Sandia Labs, Tech. Report SAND2014-0688*, 2014.

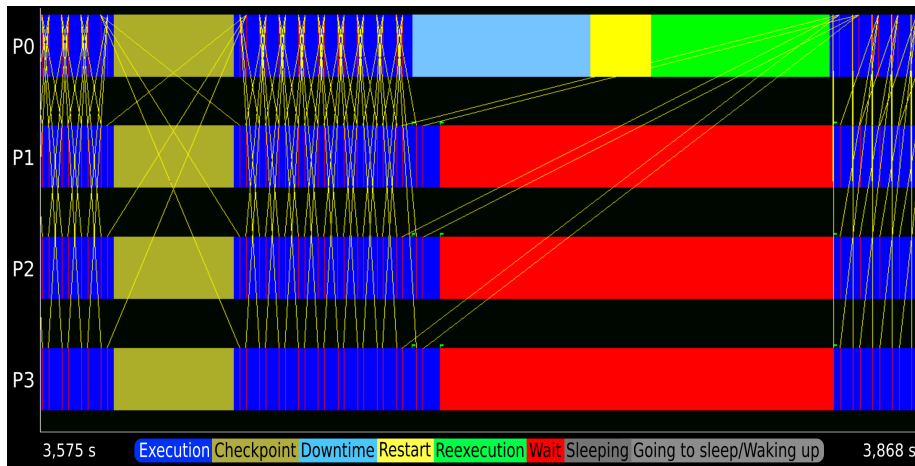


Figure 15: Scenario 7: Matrix multiplication with short reexecution time (non-blocking operations)

- [24] H. Meyer, R. Muresano, M. Castro-León, D. Rexachs, and E. Luque, “Hybrid message pessimistic logging. improving current pessimistic message logging protocols,” *Journal of Parallel and Distributed Computing*, vol. 104, pp. 206–222, 2017.
- [25] M. Castro-León, H. Meyer, D. Rexachs, and E. Luque, “Fault tolerance at system level based on RADIC architecture,” *Journal of Parallel and Distributed Computing*, vol. 86, pp. 98–111, 2015.
- [26] J. C. Ho, C.-L. Wang, and F. C. Lau, “Scalable group-based checkpoint/restart for large-scale message-passing systems,” in *2008 IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–12, IEEE, 2008.
- [27] A. Wong, D. Rexachs, and E. Luque, “Pas2p tool, parallel application signature for performance prediction,” in *International Workshop on Applied Parallel Computing*, pp. 293–302, Springer, 2010.
- [28] M. H. MacDougall, *Simulating computer systems: techniques and tools*. MIT press, 1987.