# Multi-SPMD programming model with YML and XcalableMP

Miwako Tsuji, Hitoshi Murai, Taisuke Boku, Mitsuhisa Sato, Serge G. Petiton, Nahid Emad, Thomas Dufaud, Joachim Protze, Christian Terboven and Matthias S. Müller

**Abstract** This chapter describes a multi SPMD (mSPMD) programming model and a set of software and libraries to support the mSPMD programming model. The mSPMD programming model has been proposed to realize scalable applications on huge and hierarchical systems. It has been evident that simple SPMD programs such as MPI, XMP, or hybrid programs such as OpenMP/MPI cannot exploit the postpeta- or exa- scale systems efficiently due to the increasing complexity of applications and systems. The mSPMD programming model has been designed to adopt multiple programming models across different architecture levels. Instead of

Miwako Tsuji
RIKEN Center for Computational Science, Kobe, JAPAN e-mail: miwako.tsuji@riken.jp

Hitoshi Murai
RIKEN Center for Computational Science, Kobe, JAPAN e-mail: h-murai@riken.jp

Taisuke boku
Center for Computational Sciences, University of Tsukuba, Tsukuba, JAPAN e-mail: taisuke@ccs.tsukuba.ac.jp

Mitsuhisa Sato
RIKEN Center for Computational Science, Kobe, JAPAN e-mail: msato@riken.jp

Serge G. Petiton
LIFL, Université de Lille, Lille, FRANCE e-mail: serge.petiton@univ-lille.fr

Nahid Emad
Li-Parad, UVSQ, Versailles, FRANCE e-mail: nahid.emad@uvsq.fr

Thomas Dufaud
Li-Parad, UVSQ, Versailles, FRANCE e-mail: thomas.dufaud@uvsq.fr

Joachim Protze
RWTH Aachen University, Aachen, GERMANY e-mail: protze@itc.rwth-aachen.de

Christian Terboven
RWTH Aachen University, Aachen, GERMANY e-mail: terboven@itc.rwth-aachen.de

Matthias S. Müller
RWTH Aachen University, Aachen, GERMANY e-mail: mueller@itc.rwth-aachen.de

invoking a single parallel program on millions of processor cores, multiple SPMD programs of moderate sizes can be worked together in the mSPMD programming model. As components of the mSPMD programming model, XMP has been supported. Fault-tolerance features, correctness checks, and some numerical libraries' implementations in the mSPMD programming model have been presented.

## 1 Introduction

From petascale, post-petascale to exascale, supercomputers will be larger, denser, and more complicated. A huge number of cores will be arranged in a multi-level hierarchy, such as a group of cores in a node, a group or cluster of nodes tightly linked, and a cluster of clusters. Because it is not easy to fully utilize such systems for current programming models such as simple SPMD, OpenMP+MPI, it is essential to adopt multiple programming models across different architecture levels. In order to exploit the performance of such systems, we have proposed a new programming model called the multi-SPMD (mSPMD) programming model, where several MPI programs and OpenMP+MPI programs work together conducted by a workflow programming[14]. To develop each of the mSPMD components in a workflow, XcalableMP (XMP) has been supported. In this chapter, we introduce the mSPMD programming model, and a development and execution environment implemented to realize the mSPMD programming model.

## 2 Background: International collaborations for the post-petascale and exascale computing

There were two important international collaborative projects to plan, implement, and evaluate the multi SPMD programming model. In this section, we describe the projects briefly.

Firstly, Framework and Programming for Post Petascale Computing (FP3C) project conducted during September 2010 - March 2013 aimed to exploit efficient programming and method for future supercomputers. The FP3C project was a French-Japan research project, where more than 10 Universities and research institutes participated. Featured topics of the project were new programming paradigms, languages, methods, and systems for the existing and future supercomputers. The mSPMD programming had been proposed in the FP3C project. Many important features in the mSPMD programming model had been implemented during the project period.

The priority program "Software for Exascale Computing" (SPPEXA) had been conducted to address fundamental research on the various aspects of HPC software during 2013-2015 (phase-I) and 2016-2018 (phase II). The project "MUST Correctness Checking for YML and XMP Programs (MYX) " had been selected as a phase-II

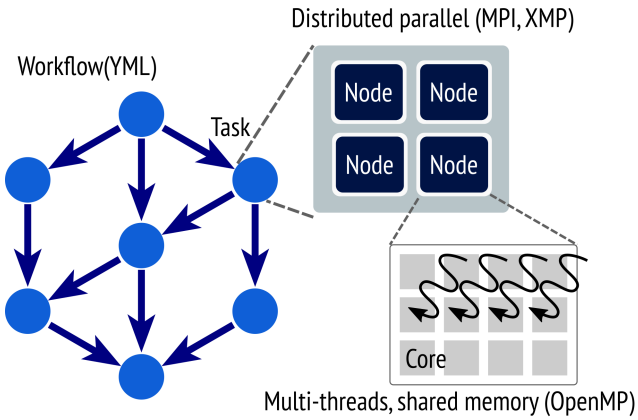



**Fig. 1** An overview of multi-SPMD programming model

program of the SPPEXA. As the name of the project suggested, the MYX project combined MUST, developed in Germany, YML, developed in France, and XMP, developed in Japan to investigate the application of scalable correctness checking methods. The deliverable from the MYX project will be described in section 8.

## 3 Multi SPMD programming model

### 3.1 Overview

While most programming models consider MPI+X such as MPI+OpenMP, or MPI+$X_1$+$X_2 \cdots$, we consider $X_1$+MPI (or XMP) +$X_2$ and propose a multi-SPMD (mSPMD) programming model where MPI programs and OpenMP+MPI programs work together in the context of a workflow programming model. In other words, tasks in a workflow are parallel programs written in XMP, MPI, or their hybrid with OpenMP.

Fig. 1 shows the overview of the mSPMD programming model. In the target systems we have expected, there should be non-uniform memory access (NUMA), general-purpose many-core CPUs, and accelerators such as GPU. We employ a shared memory programming model within a node, or a group of cores, and GPGPU programming on an accelerator. In a group of nodes, we have considered a distributed parallel programming model. Between these groups of nodes, there is a workflow programming model to manage and control several distributed parallel programs and hybrid programs of the distributed parallel and shared programming models. To realize this framework, we support XcalableMP (XMP) to describe the distributed parallel programs in a workflow as well as MPI, which is a de-facto standard for

```
<?xml version="1.0" ?>
<component type="abstract" name="add_mat">
<params>
<param name="A" type="Matrix" mode="in" />
<param name="B" type="Matrix" mode="in" />
<param name="C" type="Matrix" mode="out" />
</params>
</component>
```

**Fig. 2** An example of "Abstract" in YML

```
<?xml version="1.0" ?>
<component type="impl" name="add_mat">
<impl lang="XMP" nodes="CPU:(4x4)">
<templates>
<template name="t" format="block" size="(256,256)" />
</templates>
<distribute>
<param template="t" name="A(256,256)" align="[i][j],(j,i)" />
<param template="t" type="B(256,256)" align="[i][j],(j,i)" />
<param template="t" type="C(256,256)" align="[i][j],(j,i)" />
</distribute>
<source>
<![CDATA[
   ....
#pragma xmp loop(j,i) on t(j, i)
  for(i=0; i<256; i++){
     for(j=0; j<256; j++)
       C[i][j] = A[i][j]+B[i][j];
]]>
</source>
</component>
```

**Fig. 3** An example of "Implementation" in YML

distributed parallel programming. For the shared programming and GPGPU, as well as XMP+OpenMP, MPI+OpenMP, MPI+GPGPU such as Cuda, OpenACC, we support a runtime library called StarPU. The StarPU library[1], which is a task programming library for hybrid architectures, enables us to implement heterogeneous applications in a uniform way. XMP provides an extension to enable work-sharing among CPU cores and GPU [7]. YML[4, 3, 2] – a development and execution environment for a scientific workflow – is used for the workflow execution.

### 3.2 YML

YML[4, 3, 2] is a workflow programming environment for a scientific workflow. YML had been developed to execute a workflow application in a grid and P2P environment and provides the following software:

- Component (task) generator,
- Workflow compiler and
- Workflow scheduler.

The YML workflow compiler supports an original workflow language called YvetteML, which allows us to describe dependency between tasks easily. Some details of the YvetteML are described later, in section. 4.2. A workflow written in the YvetteML would be compiled by the YML workflow compiler into a DAG of tasks. The YML workflow scheduler interprets the DAG to execute the defined workflow. Depending on the available systems, the scheduler uses different middleware, such as XtremWeb for a P2P, OmniRPC[9] for a grid. The YML component generator generates executable programs from "abstract" and "implementation" descriptions of a component. Figure 2 and 3 show examples of "abstract" and "implementation" respectively. Note that in the figure 3, while we show an example using XMP, the original YML had supported neither XMP nor MPI. The XMP and MPI supports were added by extending YML and middleware.

### 3.3 OmniRPC-MPI

YML has been designed to execute workflow applications over various environments, such as clusters, P2P, and single processors. During the execution, the YML workflow scheduler dynamically loads a backend library for its environment. Each of the backend libraries calls APIs defined in middleware libraries. For example, in a grid environment, the OmniRPC backend linked with the OmniRPC middleware library should be loaded.

The OmniRPC [9] is a grid RPC facility for cluster systems. The OmniRPC supports a master-worker programming model, where remote serial programs (rexs) are executed by exec, rsh or ssh.

To realize the mSPMD programming model, we have implemented an MPI backend and extended the OmniRPC to OmniRPC-MPI for a large scale cluster environment. The OmniRPC-MPI library provides the following functions:

- invoke a remote program (worker program) over a specified number of nodes
- communication between the workflow scheduler and the remote programs
  - the scheduler sends a request to execute a certain task to a remote program
  - the scheduler listens to the communicator and receives a termination message from a remote program
- manage remote programs and computational resources

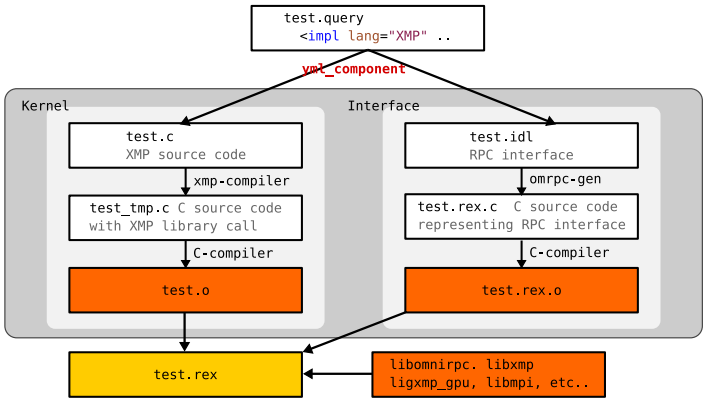



**Fig. 4** YML Component (task) generator extended for the mSPMD programming environment

# 4 Application Development in the mSPMD programming environment

In this chapter, we describe how to develop applications in the mSPMD programming environment.

## 4.1 Task generator

Fig. 4 shows the YML Component generator extended for the mSPMD programming environment. The generator takes an implementation source code, such as the one shown in Fig. 3. Then, combining the implementation and abstract source codes, it generates several intermediate files: 1. an XMP source code, which extracts task procedure itself defined by a user, and 2. an interface definition file, which includes some communication functions used to communicate with a workflow scheduler. The YML Component generator calls 1. an XMP compiler to translate the XMP source code to a C-source code with XMP runtime library calls, and 2. a C-compiler to compile the C-source code generated by the XMP compiler. The YML Component generator calls 1. an OmniRPC-generator to translate the interface definition to a C-source code, and 2. a C-compiler to compile the C-source code generated by the OmniRPC generator. Finally, the YML Component generator calls a linker to link the compiled object files and external libraries such as an MPI library.

During a workflow application execution, the remote programs generated by the YML Component generator are invoked and managed by the YML workflow scheduler and the OmniRPC-MPI middleware.

```
<?xml version="1.0" ?>
<application name="Gauss-Jordan">
<graph>

par(k:=0;nBlocks-1); do
  par
    if (k neq 0) then
      wait(prodDiffA[k][k][k-1]);
    endif
    compute inversion(A[k][k],
          B[k][k]); #step0:inversion
    notify(bInversed[k][k]);
//
    if (k neq nBlocks-1) then
      par (i:=k+1; nBlocks-1)
      do
        wait(bInversed[k][k]);
        compute prodMat(B[k][k],
                            A[k][i]);
        notify(prodA[k][i]);
      enddo
    endif
//
    wait(bInversed[k][k]);
    par(i:=0;nBlocks-1); do
      if(i neq k) then
        compute mProdMat
          (A[i][k],B[k][k],B[i][k]);
        notify(mProdB[k][i][k]);
      endif
      if(k gt i) then
        compute prodMat(
                    B[k][k],B[k][i]);
        notify(prodB[k][i]);
      endif
    enddo
//
    par(i:= 0;nBlocks-1)
    do
      if(i neq k) then
        if(k neq nBlocks - 1) then
          par(j:=k+1; nBlocks-1); do
            wait(prodA[k][j]);
            compute prodDiff(A[i][k],
                    A[k][j],A[i][j]);
            notify
                (prodDiffA[i][j][k]);
          enddo
        endif
        if (k neq 0) then
          par(j:=0;k - 1); do
            wait(prodB[k][j]);
            compute prodDiff(A[i][k],
                    B[k][j],B[i][j]);
          enddo
      endif;endif;enddo;endpar;enddo
</graph>
</application>
```

**Fig. 5** An example of an application written in YvetteML

## 4.2 Workflow development

A workflow application in the mSPMD is defined by a workflow description language called YvetteML. The YvetteML allows us to define the dependencies between tasks easily. Fig. 5 shows an example of the YvetteML, which computes an inversion of a matrix by the block-Gauss-Jordan method. In the YvetteML, the following directives are supported:

**compute** call a task
**par** parallel loop or region
each index of the loop can be executed in parallel, or
each code block defined by **//** in a **par** region can be executed in parallel
**ser** serial loop
**wait** wait until the corresponding signal has been issued by **notify**
**notify** issues a specific signal for **wait**

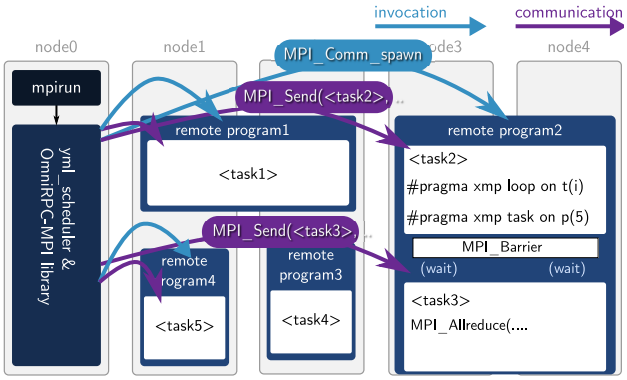



**Fig. 6** A workflow execution of the mSPMD

**Table 1** Specification of K computer

| | |
|---|---|
| CPU | Fujitsu SPARC64VIIIfx, 8 core, 2.00 GHz |
| Memory | 16GB , 64GB/s |
| Cache | L1:32+32KB/core, L2:6MB/core |
| Network | Tofu (6D mesh/torus) Interconnect |
| | 5GB/s x 2 |

## 4.3 Workflow execution

The YML workflow compiler compiles the YvetteML into a directed acyclic graph (DAG), and the YML workflow scheduler interprets the DAG to execute a workflow application.

Fig. 6 illustrates a workflow execution in the mSPMD programming model. First, mpirun kicks the YML workflow scheduler. The YML workflow scheduler, which has been linked with the OmniRPC-MPI library, interprets the DAG of a workflow application and asks the invocation a task specified by YvetteML **compute (task-name)** to the OmniRPC-MPI library. The OmniRPC-MPI library finds a remote program which includes the specified task, and invokes the remote program over the specified number of nodes by calling MPI_Comm_spawn, and sends a request to perform the specific task.

While actual communications, node management, and task scheduling have been supported by the OmniRPC-MPI library, the YML workflow scheduler schedules a “logical” order of tasks based on the DAG of an application.

**Table 2** Tasks in the Block Gauss Jordan workflow application. The input/output of the tasks such as $A_{i,j}, B_{i,j}, C_{i,j}, \cdots$ are blocks of a matrix

| task | description |
|---|---|
| inversion($A_{i,j}$) | compute the inversion of a block $A_{i,j}$ |
| prodMat($A_{i,j}$, $B_{i,j}$) | compute $B_{i,j} = A_{i,j} B_{i,j}$ |
| mProdMat($A_{i,j}$, $B_{i,j}$,$C_{i,j}$) | compute $C_{i,j} = -A_{i,j} B_{i,j}$ |
| prodDiff($A_{i,j}$, $B_{i,j}$,$C_{i,j}$) | compute $C_{i,j} = C_{i,j} - A_{i,j} B_{i,j}$ |

**for** $k$ **from** 0 **to** $p-1$ **do**
  (1) $Inv^{(k)} = [A_{k,k}^{(k)}]^{-1}$
  (2) $\boldsymbol{b}_k^{(k+1)} = Inv^{(k)} \cdot \boldsymbol{b}_k^k$
  **if** $k \neq p-1$ **then**
    **for** $i$ **from** 0 **to** $p-1$ **do**
      **if** $i \neq k$ **then**
        (3) $A_{k,i}^{(k+1)} = Inv^{(k)} \cdot A_{k,i}^{(k)}$
        (4) $\boldsymbol{b}_i^{(k+1)} = b_i^{(k+1)} - A_{i,k}^{(k)} \cdot \boldsymbol{b}_k^{(k+1)}$
        **for** $j$ **from** $k+1$ **to** $p-1$ **do**
          (5) $A_{i,j}^{(k+1)} = A_{i,j}^{(k)} - A_{i,k}^{(k+1)} \cdot A_{k,j}^{(k)}$
        **end for**
      **end if**
    **end for**
  **end if**
**end for**

**Fig. 7** Algorithm of the Block Gauss Jordan method

**Table 3** # of blocks, Block sizes and # of tasks in the Block Gauss Jordan method

| # of blocks | 1x1 | 2x2 | 4x4 | 8x8 | 16x16 |
|---|---|---|---|---|---|
| block size | $32768^2$ | $16384^2$ | $8192^2$ | $4096^2$ | $2048^2$ |
| # of tasks | 3 | 18 | 108 | 696 | 4848 |

## 5 Experiments

In this section, we demonstrate the performance of the mSPMD programming model and our implementation.

Table 1 shows the specification of the K-computer, which has been used for the experiments.

In our experiments, the Block Gauss-Jordan (BGJ) method, which computes the inversion of a matrix $A$, has been considered. Fig. 7 shows the algorithm of the BGJ method. The workflow for the BGJ method written in YvetteML has been shown in Fig. 5. As shown in the table 2, tasks in the workflow process block(s). In order to investigate the performance over different levels of hierarchical parallelism:

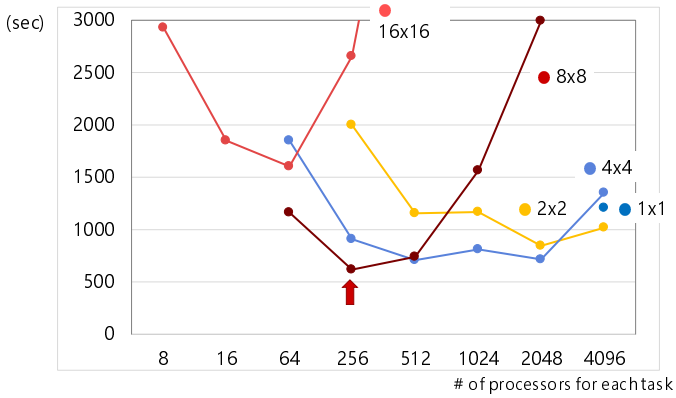



**Fig. 8** Execution time of the BGJ workflow applications

- the total size of the matrix $A$ is fixed to $32768 \times 32768$, but the number of blocks is varied from $1 \times 1$ to $16 \times 16$.
- the total number of processes (cores) for a workflow is fixed to 4096, but the number of processes for each task is varied from 8 to 4096.

Therefore, if we have a single block and assign all processes for a task, then it is almost equivalent to a distributed parallel application. On the other hand, if we divide a matrix into many small blocks and assign a process for each block, it is almost a traditional workflow application. Table 3 shows the block size and the number of tasks for each number of blocks. If we assign 512 processes for each task, then at most 8 tasks can be executed simultaneously.

Figure 8 shows the execution time of the BGJ workflow applications for the number of blocks and the number of processes per task. The results show that the best performance has been realized when we divide a matrix into $8 \times 8$ blocks and assign 256 processes for each task. Our framework of the mSPMD programming model can realize such an appropriate combination of different parallelisms and can allow application developers to control the different parallelism levels easily. On the other hand, the extreme cases — $1 \times 1$ block and $16 \times 16$ blocks — have not performed well. Also, assigning too many processes for small tasks, for example, 2048 processes for $8 \times 8$ blocks, more than 256 processes for $16 \times 16$ processes, show poor performance.

Figure 9 shows the execution timeline (from left to right) of the BGJ workflow application with $8 \times 8$ blocks. As shown in the figure, at the first step, the task of inversion ($B = A^{-1}$) must be executed solely since the other tasks on the first step uses the result of the inversion. After the 2nd step, some of the matrix calculations such as $A = A \times B, C = -(B \times A), C = C - (B \times A)$ on the $k$-th step and the inversion on $k+1$-th step can be overlapped. For other programming models such as flat-MPI, it is not easy to execute tasks or functions on different steps simultaneously. On the

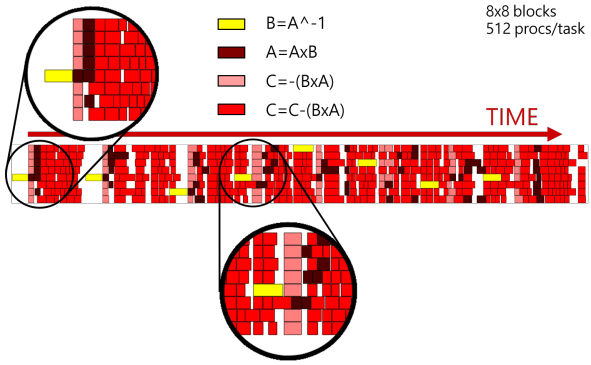



**Fig. 9** Execution timeline of the BGJ workflow application with $8 \times 8$ blocks

other hand, the mSPMD programming model and our programming environment allow application developers to describe this sort of applications easily.

## 6 Eigen solver on the mSPMD programming model

In this section, as a use case of the mSPMD programming model, we introduce an eigen solver implemented on the mSPMD programming model.

### 6.1 Implicitly Restarted Arnoldi Method (IRAM), Multiple Implicitly Restarted Arnoldi Method (MIRAM) and their implementations for the mSPMD programming model

The iterative methods are widely used to solve eigenvalue programs in scientific computation. Implicitly Restarted Arnoldi Method (IRAM) [10] is one of the iterative methods to search the eigen elements $\lambda$ s.t. $A\boldsymbol{x} = \lambda\boldsymbol{x}$ of a matrix $A$.

Figure 10 shows the algorithm of IRAM. IRAM is a technique that combines the implicitly shifted QR mechanism with an Arnoldi factorization and the IRAM can be viewed as a truncated form of the implicitly shifted QR-iteration. After the first $m$-step Arnoldi factorization, the eigen pairs of a Heisenberg matrix $H$ are computed. If the residual norm is enough small, the iteration is stopped. Otherwise, the shifted QR by selecting shifts based on eigenvalues of the Heisenberg matrix is computed. Using these new vectors and $H$ as a starting point, we can apply $p$ additional steps of the Arnoldi process to obtain an $m$-step Arnoldi factorization.

Multiple IRAM (MIRAM) is an extension of IRAM, which introduces two or more instances of IRAM. The instances of IRAM work on the same problem, but

Input: $A, V_m, H_m, F_m$ with $AV_m = V_m H_m + f_m e_m$
$A$ is a matrix of order $n$,
$m$ is krylov subspace size s.t. $m \ll n$,
$k$ is the number of desired eigen pairs s.t. $k < m$.
**for** $i$ **from** 0 **to** convergence **do**
(1) compute the eigenvalue $\sigma$ and eigenvectors of $H_m$
compute residual norm. If converge, stop.
(2) Select a set of $p = m - k$ shifts $(u_1, \cdots, u_p)$ and set $q^T := e_m^T$
(3) **for** $j$ **from** 1 **to** $p$
(1) Factor$[Q_j, R_j]$ = qr$(H_m - u_j I)$
(2) $H_m = Q_j^T H_m Q_j, V_m = V_m Q_j$
(3) $q^H = q^H Q_j$
**end for**
(4) $f_k = v_{k+1}\beta_k + f_m \sigma_k, V_k = V_m(1:n, 1:k), H_k = H_m(1:k, 1:k)$
(5) Apply $p$ additional steps of Arnoldi process to obtain a new $m$ step Arnoldi factorization $AV_m = V_m H_m + f_m e_m^T$
**end for**

**Fig. 10** Algorithm of IRAM

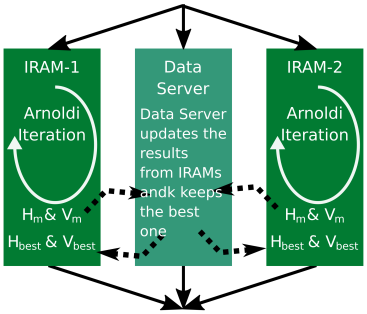



**Fig. 11** Overview of MIRAM on the mSPMD programming model

```
<?xml version="1.0" ?>
<application name="MIRAM">
<graph>
   niram:=3;
   par
      par(eid:=0;niram-1); do
       compute iram_main(eid,...);
      enddo
     //
      compute iram_dsrv(niram,...);
   endpar
</graph>
</application>
```

**Fig. 12** MIRAM workflow

they are initialized with different subspaces $m_1, m_2, \cdots$. At the restarting point, each instance selects the best $(m_{best}, H_{best}, V_{best}, f_{best})$ from $l$ IRAM instances.

In the mSPMD programming model, MIRAM has been implemented, as shown in Figure 11. The source code written in YvetteML is shown in Figure 12. The YML workflow scheduler invokes $l$ IRAM instances and a data server. Each of IRAM instances computes an Arnoldi iteration asynchronously over $n$ nodes and sends the resulting $(m, H, V, f)$ to the data server. The data server keeps the best result and sends it to each IRAM. Each IRAM restarts with the $(m, H, V, f)$ sent by the data server.

**Table 4** Specification of T2K Tsukuba

| | |
|---|---|
| CPU | Opteron Barcelona B8000, 4 cores, 4 sockets, 2.3 GHz |
| Memory | 32GB |
| Network | Fat-tree, full-bisection interconnection quad-rail of InfiniBand |

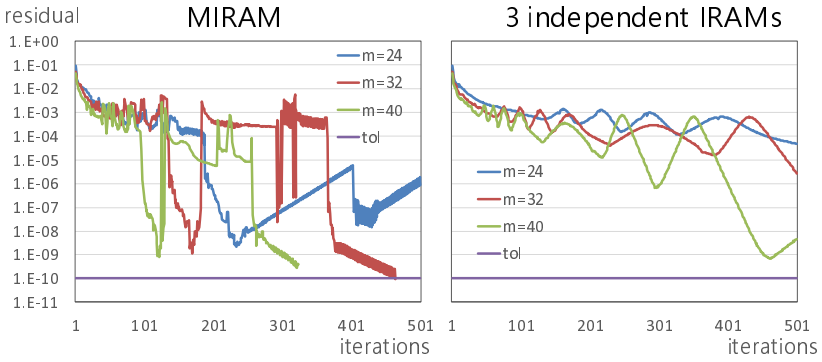



**Fig. 13** Results of MIRAM with 3 IRAM solvers (right) and 3 independent run of IRAM (left)

## 6.2 Experiments

Here, we show the result of experiments on the T2K-Tsukuba supercomputer. The specification of the T2K Tsukuba is shown in Table 4. In the experiments, we use a matrix called *Schenk/nlpkkt240* from the SuiteSparse Matrix Collection [11], where $n = 27,993,600$ and the number of non-zero elements are 760,648,352.

Figure 13 shows the results of MIRAM with IRAM solvers of $m = 24, 32, 40$ (left) and 3 independent run of IRAM solvers of the $m = 24, 32, 40$ (right). While MIRAM converged around 450 iterations, non of 3 IRAMs could not converge until 500 iterations.

This MIRAM example shows that by using the mSPMD programming model, two different accelerations can be achieved. While the workflow programming model of the mSPMD accelerates the convergence of the Arnoldi iterations, the distributed parallel programming model speeds up each iteration of the Arnoldi method.

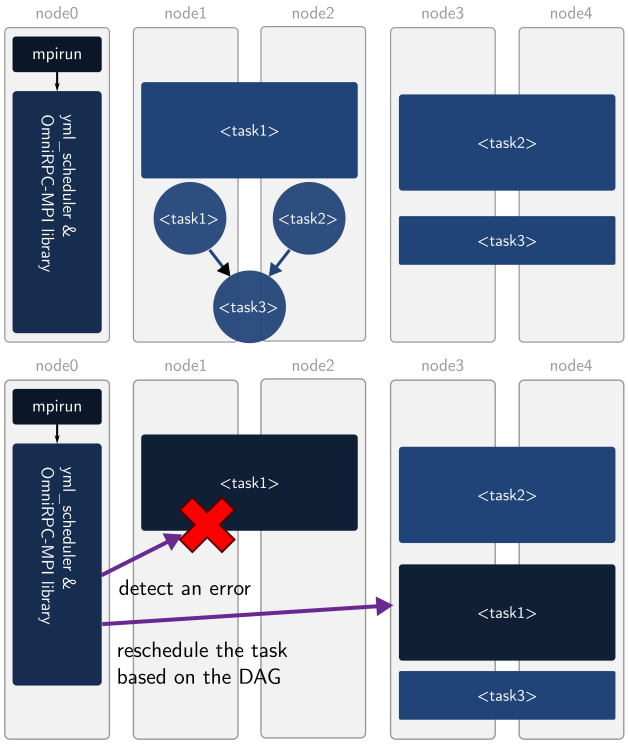



**Fig. 14** Overview of fault tolerance in the mSPMD programming model

## 7 Fault Tolerance Features in the mSPMD programming model

### 7.1 Overview and implementation

As well as scalability and programmability, reliability is an important issue in exascale computing. Since the number of components of an exascale supercomputer should be tremendously large, it is evident that the mean time between failure (MTBF) of a system decreases as the number of the system's components increases. Therefore, fault tolerance becomes essential for systems and applications. Here, we develop a fault tolerance mechanism in an mSPMD programming model, and its development and execution environment. The fault tolerance in the mSPMD programming model can be realized without modifying applications' source codes[13].

Figure 14 illustrates the fault-tolerant mechanism in the mSPMD programming model. If the workflow scheduler can find an error in a task and execute the task again on different nodes, then we can realize a fault tolerance and resilience mechanism automatically.

**Table 5** Specification of a FX10 cluster

| | |
|---|---|
| CPU | Fujitsu SPARC64VIIIIfx, 16 core, 1.65 GHz |
| Memory | 32GB, 85GB/s |
| Cache | L1:32+32KB/core, L2:6MB/core |
| Network | Tofu (6D mesh/torus) Interconnect<br>5GB/s x 2 |

We have extended the OmniRPC-MPI described in 3.3 to detect errors in remote programs and notify the errors to the YML workflow scheduler. For these purposes, heartbeat messages between master and remote programs have been introduced in the OmniRPC-MPI library. If an error is detected in a remote program, then it is reported to the YML workflow scheduler as a return value of existing APIs. The `OmniRpcProbe(Request r)` API has been designed to listen to the status of a requested task in a remote program. This returns `success` if the remote program sends a signal to indicate the requested task `r` has successfully finished. On the other hand, if heartbeat messages from the remote program executing the task `r` have stopped, `OmniRpcProbe(Request r)` returns `fail`.

The YML scheduler re-scheduler the failed task if it receives `fail` signal from the OmniRPC-MPI library. The re-scheduling method is simple; The YML scheduler puts the failed task at the head of the "ready" task-queue.

## 7.2 Experiments

We have performed some experiments to investigate the overhead of the fault detection and the elapsed time when errors occur on a cluster shown in Table 5. The BGJ method shown in section 5 had been used. The size of a matrix is $20480 \times 20480$ and divided into

| # of blocks | 1x1 | 2x2 | 4x4 | 8x8 |
|---|---|---|---|---|
| block size | $20480^2$ | $10240^2$ | $5120^2$ | $2560^2$ |

1024 cores (64 nodes) are used for each workflow, and 64 to 1024 cores are assigned for each task in a workflow.

Firstly, we have considered the overhead of the heartbeat messages used to detect errors in remote programs. Figure 15 shows the performance of the normal and fault-tolerant mSPMD programming executions using between 64 and 1024 compute cores per task. The dotted lines are the results of fault-tolerant mSPMD programming executions, and the solid lines are those of the regular mSPMD programming executions. As shown in the figure, the best combination of the number of blocks and the number of processes per task is $4 \times 4$ blocks and 512 processes for both cases of with and without fault-tolerance support. The overhead of using a heartbeat message is very small and is 2.3% on average and 4.7% at a maximum.

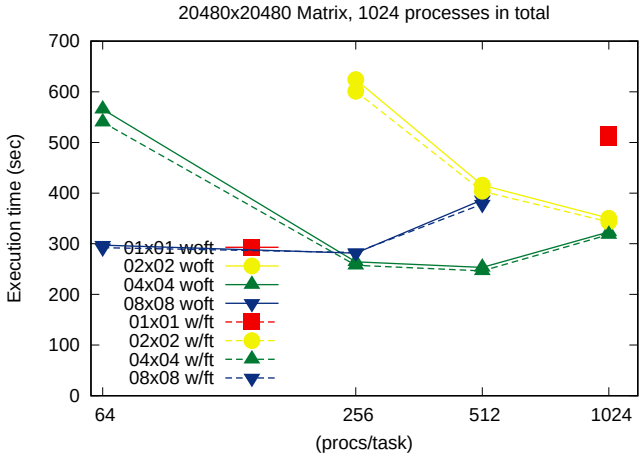



**Fig. 15** Execution time with and without FT for the number of cores for each task. The graph legends show the number of blocks.

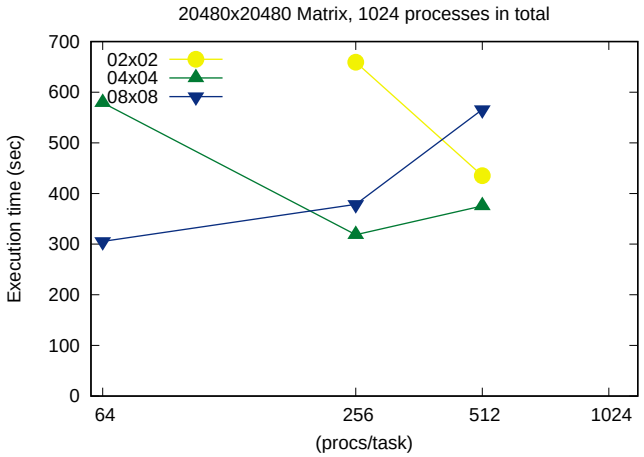



**Fig. 16** Execution time with FT for the number of cores for each task under fake errors. The graph legends show the number of blocks.

Then, we have investigated the behavior and performance of the fault-tolerant mSPMD programming execution when errors occur. Instead of waiting for real errors, we have inserted fake errors that stop heartbeat messages from remote programs randomly with a certain error probability computed by an expected MTBF (90,000 sec). Figure 16 shows the performance of the fault-tolerant mSPMD programming execution under fake errors. Unfortunately, for the case of $1 \times 1$ block and 1024 processes per task, it was not possible to complete the workflow, since the face error ratio used in the experiment is higher than real systems. For the other cases, the applications can be completed. The best combination of the number of blocks and the number of processes per task is $4 \times 4$ blocks and 256 processes while it was 512 processes under the "no-error" condition. This is because the tasks executed on a relatively small number of nodes are relatively easy to recover when they fail.

# 8 Runtime correctness check for the mSPMD programming model

## 8.1 Overview and implementation

The mSPMD programming model has been proposed to realize scalability for large scale systems. Additionally, as we discussed in section 7, we support fault-tolerant features in the mSPMD programming model. In this section, we discuss another important issue in large scale systems, productivity.

One of the reasons for the low productivity in distributed parallel programs is the difficulty of debugging. Several libraries and tools have been proposed to help and debug parallel programs. MUST (Marmot Umpire Scalable Tool) [12, 6, 5] is a runtime tool that provides a scalable solution for efficient runtime MPI error checking. The MUST has supported not only MPI but also XcalableMP (XMP) [8].

In this section, we discuss how to adapt the MUST library to the SPMD programs in the mSPMD programming model and enable the MUST correctness checking for the mSPMD. Computational experiments have been performed to confirm MUST's operation in the mSPMD and to estimate the overhead of the correctness checking.

The mSPMD programming model consists of workflow scheduler, middleware, remote programs, and so on. Each of the remote programs includes user-defined tasks and control sections where the remote program communicates with the workflow scheduler. In this work, we focus on the user-defined tasks within the remote programs, and the correctness check by the MUST library should be applied only to the user-defined tasks. Fig. 17 shows an overview of the application execution in the mSPMD programming model and the target of the correctness check by the MUST library in the mSPMD programming model. While MUST checks the MPI and XMP communications shown in orange letters, MPI_Comm_spawn used to invoke remote programs, MPI_Send used to send a request to the remote programs, must be ignored.

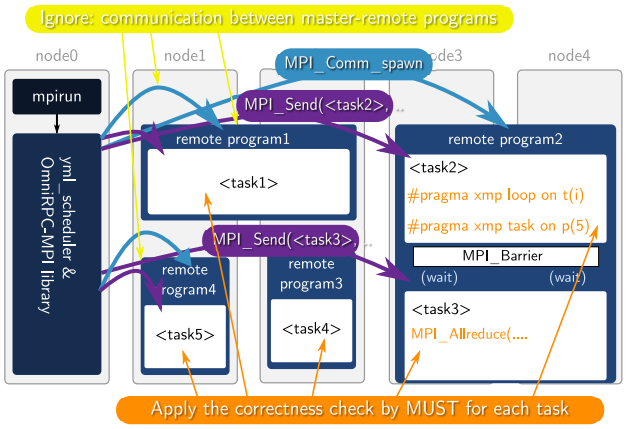



**Fig. 17** Applying MUST to the mSPMD programming model. Only communication functions written in red letters are checked, MPI functions written in black letters are not checked.

```
#define MYX

#ifdef MYX
#define MPI_Comm_rank(comm, rank)\
       PMPI_Comm_rank(comm, rank)

#define MPI_Barrier(comm)\
       PMPI_Barrier(comm)

....

#endif // #ifdef MYX
```

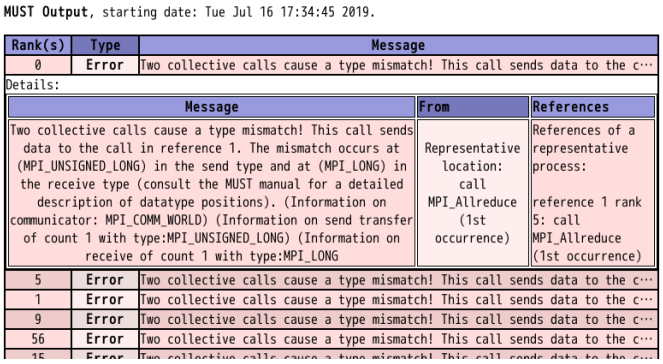
**Fig. 18** The Macro to disable the MUST correctness check

```
#define MPI_Comm_spawn(command, argv, maxprocs,\
 info, root, comm, intercomm, array_of_errcodes)\
       PMPI_Comm_spawn(command, argv, maxprocs+1,\
 info, root, comm, intercomm, array_of_errcodes)
```

**Fig. 19** The Macro to invoke remote programs with $n + 1$ processes where "+1" is kept for MUST

MUST replaces MPI functions starting with MPI_ such as MPI_Send with their own MPI functions, including correctness check and actual communication. The functions starting with MPI_ such as MPI_Send in standard MPI libraries wrap the functions starting with PMPI_ which perform communication. In order to avoid the correctness check for the control sections, we define some macro to use PMPI functions directly (Figure 18). Moreover, to reserve an additional process for MUST in remote programs, we define the macro to invoke remote programs (19).

The original MUST creates an output file named `MUST_Output.html` for each of parallel applications. On the other hand, in the mSPMD programming model, there are one or more parallel applications simultaneously. Therefore, we modify



**MUST Output**, starting date: Tue Jul 16 17:34:45 2019.

| Rank(s) | Type | Message |
|---|---|---|
| 0 | Error | Two collective calls cause a type mismatch! This call sends data to the c··· |

Details:

| Message | From | References |
|---|---|---|
| Two collective calls cause a type mismatch! This call sends data to the call in reference 1. The mismatch occurs at (MPI_UNSIGNED_LONG) in the send type and at (MPI_LONG) in the receive type (consult the MUST manual for a detailed description of datatype positions). (Information on communicator: MPI_COMM_WORLD) (Information on send transfer of count 1 with type:MPI_UNSIGNED_LONG) (Information on receive of count 1 with type:MPI_LONG | Representative location: call MPI_Allreduce (1st occurrence) | References of a representative process: reference 1 rank 5: call MPI_Allreduce (1st occurrence) |

| | | |
|---|---|---|
| 5 | Error | Two collective calls cause a type mismatch! This call sends data to the c··· |
| 1 | Error | Two collective calls cause a type mismatch! This call sends data to the c··· |
| 9 | Error | Two collective calls cause a type mismatch! This call sends data to the c··· |
| 56 | Error | Two collective calls cause a type mismatch! This call sends data to the c··· |

**Fig. 20** Screenshot of the output file generated by MUST in the mSPMD programming model

**Table 6** The specification of the Oakforest PACS

| | |
|---|---|
| CPU | Intel Xeon Phi 7250 (KNL), 68 core, 1.4 GHz |
| Memory | 96GB(DDR) + 16GB(MCDRAM) |
| Network | Intel Omni-Path Network, 100 Gpbs |
| Compiler | intel/2018.1.163 |
| MPI library | impi/2018.1.163 |
| OS | CentOS 7 |

the MUST library to generate different `MUST_Output_<id>.html` files for different remote programs. So far, we give a process id of the rank-0 of a remote program as the `<id>` the output file. Figure 20 shows an example of the output file generated by MUST in the mSPMD programming model.

## 8.2 Experiments

We have performed some experiments to evaluate the execution times and to investigate applications' behaviors with and without the MUST library. In these experiments, the Oakforest-PACS (OFP) system has been used. Table 6 shows the specification of the OFP. For remote programs, we adopt the flat-MPI programming model where each MPI process runs on each core.

We focus on collective communication (MPI_Allreduce) and point-to-point communication (Pingpong) and consider codes with and without error for each. Fig. 21 shows the tasks used in the experiments. From the top to bottom, allreduce (w/o error), allreduce (w/ error, type mismatch), pingpong (w/o error), pingpong (w/ error, type mismatch), allreduce (w/ error, operation mismatch), and allreduce (w/ error, buffer size mismatch). Also, we consider different numbers of iterations and differ-

```
for(i=0; i<Niter; i++){
  MPI_Allreduce(buf, rbuf, 1, MPI_LONG, MPI_SUM, MPI_COMM_WORLD);
  usleep(Usec);
}
```

```
for(i=0; i<Niter; i++){
  if(myrank==0)
  MPI_Allreduce(buf, rbuf, 1, MPI_UNSIGNED_LONG,
                                        MPI_SUM, MPI_COMM_WORLD);
  else
  MPI_Allreduce(buf, rbuf, 1, MPI_LONG, MPI_SUM, MPI_COMM_WORLD);
  usleep(Usec);
}
```

```
 for(i=0; i<Niter; i++){
   if(myrank%2==0)
   MPI_Send(buf, 1, MPI_LONG, dest, tag, MPI_COMM_WORLD);
   else
   MPI_Recv(buf, 1, MPI_LONG, src , tag, MPI_COMM_WORLD, &stat);
   usleep(Usec);
   if(myrank%2==0)
   MPI_Recv(buf, 1, MPI_LONG, src , tag, MPI_COMM_WORLD, &stat);
   else
   MPI_Send(buf, 1, MPI_LONG, dest, tag, MPI_COMM_WORLD);
   usleep(Usec);
 }
```

```
for(i=0; i<Niter; i++){
  if(myrank%2==0)
  MPI_Send(buf, 1, MPI_UNSIGNED_LONG, dest, tag, MPI_COMM_WORLD);
  else
  MPI_Recv(buf, 1, MPI_LONG, src , tag, MPI_COMM_WORLD, &stat);
  usleep(Usec);
  if(myrank%2==0)
  MPI_Recv(buf, 1, MPI_LONG, src , tag, MPI_COMM_WORLD, &stat);
  else
  MPI_Send(buf, 1, MPI_LONG, dest, tag, MPI_COMM_WORLD);
  usleep(Usec);
}
```

```
if(myrank==0)
  MPI_Allreduce(buf, rbuf, 1, MPI_LONG, MPI_MAX, MPI_COMM_WORLD);
else
  MPI_Allreduce(buf, rbuf, 1, MPI_LONG, MPI_MIN, MPI_COMM_WORLD);
```

```
if(myrank==0)
  MPI_Allreduce(buf, rbuf, 2, MPI_LONG, MPI_SUM, MPI_COMM_WORLD);
else
  MPI_Allreduce(buf, rbuf, 1, MPI_LONG, MPI_SUM, MPI_COMM_WORLD);
```

**Fig. 21** The tasks used in experiments. From the top to bottom, allreduce (w/o error), allreduce (w/ data type error), pingpong (w/o error), and pingpong (w/ error), allreduce (w/ operation error), allreduce (w/ data size error)

ent interval seconds between MPI function calls in each test code for the overhead evaluations.

Table 7 shows the applications' behaviors and the statuses of error reports, when applying or not applying MUST. While the datatype conflict and operation conflict errors are reported when we apply the MUST, the applications are completed without any report when we do not apply the MUST even though the results of the reduction should be wrong.

Figure 22 shows the execution time of the mSPMD programming executions with and without the MUST library. Workflow applications include between 1 and 32 tasks of MPI_Allreduce. Figure 23 shows the results for MPI_Send/Recv. Each task uses 32 processes in all experiments. As shown in figure 22, the overhead to check and record errors of collective communication is ignorable if we do not perform communication very intensively. On the other hand, if collective communication functions called very frequently, then the overheads become large even if there is no

**Table 7** Applications' behaviors and the statuses of error reports

| | w/ MUST | w/o MUST |
|---|---|---|
| allreduce w/o error | completed | completed |
| allreduce w/ type conflict error | completed error reports | completed no report |
| pingpong w/o error | completed | completed |
| pingpong w/ type conflict error | completed error reports | completed no report |
| allreduce w/ operation conflict error | completed error reports | completed no report |
| allreduce w/ buffer size conflict error | failed error reports | failed simple error reports |

error. As shown in figure 23, the overhead of the MUST library is small if there is no error in point to point communication functions. However, it takes more time if there are some errors. The fact indicates that there is almost no overhead to check point to point communication, but it takes some time to analyze and record errors in the point to point communication functions.

## 9 Summary

In this chapter, we have introduced the mSPMD programming model and programming environment, where several SPMD programs work together under the control of a workflow program. YML, which is a development and execution environment for scientific workflows, and its middleware OmniRPC, have been extended to manage several SPMD tasks and programs. As well as MPI, XMP, a directive-based parallel programming language, has been supported to describe tasks. A task generator has been developed to incorporate XMP programs into a workflow. Fault-tolerant features, correctness check, and some numerical libraries' implementations in the mSPMD programming model have been presented.

## References

1. Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience - Euro-Par 2009*, 23:187–198, 2011.
2. Olivier Delannoy. *YML: A scientific Workflow for High Performance Computing*. PhD thesis, University of Versailles Saint-Quentin, 2006.
3. Olivier Delannoy, Nahid Emad, and Serge Petiton. Workflow global computing with yml. In *The 7th IEEE/ACM International Conference on Grid Computing*, pages 25–32, 2006.

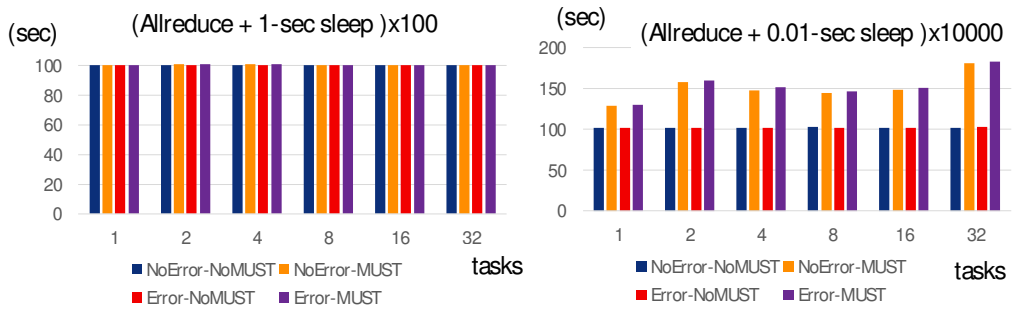



**Fig. 22** Execution time when a workflow includes $1, 2, \cdots 32$ tasks executing MPI_Allreduce repeatedly every 1-sec (left) and every 0.01-sec

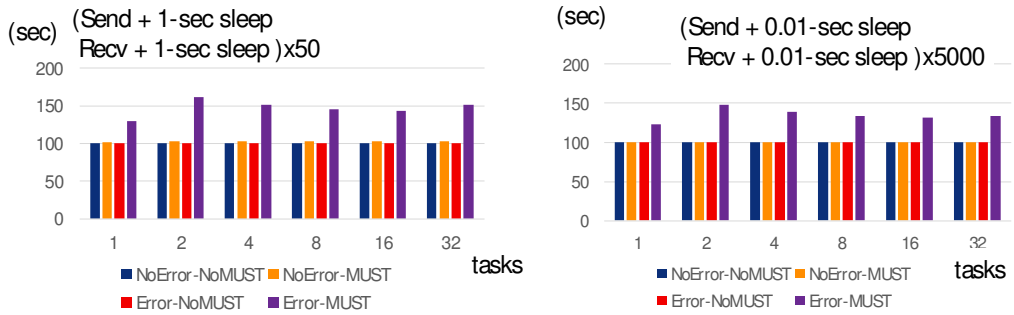



**Fig. 23** Execution time when a workflow includes $1, 2, \cdots 32$ tasks executing MPI_Send/Recv repeatedly every 1-sec (left) and every 0.01-sec

4. Olivier Delannoy and Serge Petiton. A peer to peer computing framework: Design and performance evaluation of yml. In *3rd International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, pages 362–369, 2004.
5. Tobias Hilbrich, Fabian Hasel, Martin Schulz, Bronis R. de Supinski, Matthias S. Muller, and Wolfgang E. Nagel. Runtime MPI collective checking with tree-based overlay networks. In *Proceedings of the 20th European MPI Users' Group Meeting (EuroMPI 13)*, pages 129–134. ACM, 2013.
6. Tobias Hilbrich, Joachim Protze, Martin Schulz, Bronis R. de Supinski, and Matthias S. Muller. MPI runtime error detection with MUST: Advances in deadlock detection. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC12)*. IEEE, 2012.
7. Tetsuya Odajima, Taisuke Boku, Mitsuhisa Sato, Toshihiro Hanawa, Yuetsu Kodama, Raymond Namyst, Samuel Thibault, and Olivier Aumage. Adaptive task size control on high level programming for gpu/cpu work sharing. In *International Symposium on Advances of Distributed and Parallel Computing (ADPC 2013)*, pages 59–68, 2013.
8. Joachim Protze, Christian Terboven, Matthias S. Müller, Serge Petiton, Nahid Emad, Hitoshi Murai, and Taisuke Boku. Runtime correctness checking for emerging programming paradigms. In *Proceedings of the First International Workshop on Software Correctness for HPC Applications*, pages 21–27, 2017.

9. Mitsuhisa Sato, Motonari Hirano, Yoshio Tanaka, and Satoshi Sekiguchi. Omnirpc: A grid rpc facility for cluster and global computing in openmp. In *International Workshop on OpenMP Applications and Tools, 2001*, pages 130–136, 2001.
10. Danny C. Sorensen. *ICASE/LaRC Interdisciplinary Series in Science and Engineering book series (ICAS, volume 4)*, chapter Implicitly Restarted Arnoldi/Lanczos Methods for Large Scale Eigenvalue Calculations, pages 119–165. springer, 1997.
11. SuiteSparse Matrix Collection. https://sparse.tamu.edu/.
12. The MUST Project. https://www.itc.rwth-aachen.de/must.
13. Miwako Tsuji, Serge Petiton, and Mitsuhisa Sato. Fault tolerance features of a new multi-spmd programming/execution environment. In *Proceedings of the First International Workshop on Extreme Scale Programming Models and Middleware SC15*, pages pp.20–27 doi:10.1145/2832241.2832243. ACM, 2015.
14. Miwako Tsuji, Mitsuhisa Sato, Maxime Hugues, and Serge Petiton. Multiple-spmd programming environment based on pGAs and workflow toward post-petascale computing. In *Proceedings of the 2013 International Conference on Parallel Processing (ICPP-2013)*, pages 480–485. IEEE, 2013.