A new parallel recomputing code design methodology for fast failure recovery

Yunfei Du*, Yuhua Tang, Xinwei Xie
Dept. of Computer, National University, of Defense Technology, Changsha, Hunan, China

Abstract
As the size of large-scale computer systems increases, their mean-time-between-failures are becoming significantly shorter than the execution time of many current scientific applications. Fault-tolerant parallel algorithm (FTPA) is an application-level fault-tolerant approach that can achieve fast self-recovery by parallel recomputing. The method of parallelizing the loops has been used to design the parallel recomputing code for FTPA in our prior work.

In the present paper, we first propose a new parallel recomputing code design methodology. Second, the parallel recomputing code design methodology is automated by exploring the use of compiler technology. Finally, we evaluate the performance of our approach with five programs on Tianhe-1A. The experimental results show that the parallel recomputing code generated by the new method has a higher efficiency of parallel recomputing.

1. Introduction

Large-scale scientific computing is the main driving force for high-performance computing. The unquenchable desire of today's scientists to analyze ever-larger data sets and run ever-larger simulations has driven the size of high-performance computers from thousands, to tens of thousands, and even to hundreds of thousands of processors. The fastest computer system in the world at present, Cray's Titan computer, has 299,008 AMD x86 cores and 46,645,248 CUDA cores [1]. However, as the complexity of a computer system increases, its mean-time-between-failure (MTBF) drastically decreases as well. Indeed, operation of the large IBM/LLNL ASCI White system has revealed that its MTBF is only slightly more than 40 h, despite continuing improvements [2]. Recent studies have pointed out that the MTBF of teraflop and petaflop machines are only on the order of 10^100 h [3]. On the other hand, many scientific applications are designed to run for weeks or even months. Therefore, the MTBF of these computers is becoming significantly shorter than the execution time of many current scientific applications. They must tolerate hardware failures to complete the execution of such applications.

Checkpointing techniques are widely used in the domain of the large-scale systems. For applications on most platforms, such as the IBM Blue Gene and the ASCI machines, Application-level checkpointing (ALC) is the default approach for tolerating hardware failures [4]. There has been much research performed on performance optimizations for checkpointing. Diskless checkpointing is a technique which makes a checkpointing system can operate in the absence of stable storage using processor redundancy and memory redundancy [5]. Based on functional partitioning runtime environment, a flexible solid-state disks based checkpointing system that allows for transparent sharing SSDs of across different nodes are developed, which can reduce the execution time spent in checkpointing [6]. CPPC is a checkpointing tool which can automate the insertion of fault tolerance into message-passing applications [7]. However, all checkpointing techniques require a cold restart of the entire parallel job when a process failed. In cold restart, a complete reload of all processes in the parallel job is conducted. Then all processes have...
to roll back to the last checkpoint to restart the computation from there. This results in long response times for users. Current fault-tolerant techniques redo the computation of the crashed process since the last checkpoint on a single process. As a result, the recovery time of all techniques is no less than the time between the last checkpoint and the crash.

To avoid the cold restart and to speed up the recovery procedure, in our prior papers, we proposed fault-tolerant parallel algorithm (FTPA), a new application-level fault-tolerant approach for large-scale scientific applications [8,9]. FTPA saves data at data-saving points for correct recovery during its execution. When a process fails, the failure is detected by all surviving processes, which recover the computation lost on the failed process through parallel recomputing. Parallel recomputing speeds up the recovery procedure, making the recovery time considerably less than the time between the last checkpoint and the crash.

However, FTPA complicates the coding of the application program, thus increasing the user burden and reducing productivity. To ease FTPA implementation, a source-to-source precompiler tool called GiFT (Get it Fault-Tolerant) explores the use of compiler technology to automate FTPA implementation. In GiFT, the method of auto loop parallelization is used to generate parallel recomputing codes. Loop parallelization can only deal with the do loop, and the parallel recomputing code generated by the loop parallelization cannot achieve high efficiency because of the limitation of auto loop parallelization.

The present paper proposes a new parallel recomputing code design methodology based on template to improve the efficiency of parallel recomputing. The contributions of the current research lie in two aspects.

- A new parallel recomputing code design methodology based on template is described.
- The use of compiler technology to automate the code design methodology is explored.

The rest of the paper is organized as follows. Section 2 reviews FTPA. In Section 3, we describe in detail the basic idea of the parallel recomputing code design methodology based on template, taking parallel \(\pi\) calculation as an example. Section 4 presents the parallel recomputing code design methodology. Section 5 automates the code design methodology. Section 6 evaluates the proposed methodology by experiments. In Section 7, we conclude the paper and discuss future work.

2. Overview of FTPA

Logically, a parallel program has a certain number of program sections, which are code fragments of the parallel program. Without loss of generality, a parallel algorithm has program sections \(S_0, S_1, \ldots, S_n\). The design of FTPA allows the manipulation of each program section into a fault-tolerant program section with the insertion of a data saving section, a failure detection section, and a recovery section, as shown in Fig. 1.

The data-saving section needs to save data, which is a set of variables involved in the execution of the application to guarantee correct recovery for a parallel application.

The failure detection section checks the system failure vector \(FV\) to make it aware which process has failed. Let \(N\) denote the number of processes participating in the execution of an application, then \(FV\) be an \(N\)-tuple, \((F_0, F_1, \ldots, F_{N-1})\), where \(F_i\) represents the failure type of the process \(P_i\).

The recovery section is implemented by transforming, following the SPMD programming paradigm, the original program section. Let \(W_{Sk}^i\) be the workload of the failed process \(P_i\) that executes the program section \(S_k\), and \(W_{RSk}^i\) be the workload on every surviving process that executes the recovery section \(RS_k\) corresponding to \(S_k\). The following relation stands:

\[
W_{Sk}^i = \sum_{0\leq i\leq N-1, i\neq j} W_{RSk}^i
\]

3. Basic idea

In this section, we introduce our new parallel recomputing code design methodology based on template. We take a \(\pi\) calculation program as an example and compare it to auto loop parallelization. FTPA only deals with fail-stop failures and makes the assumption that failures only occur in computations, not in communications.
3.1. A case of $\pi$ calculation

Fig. 2 shows a FORTRAN/MPI $\pi$ calculation program where $S_0$ and $S_1$ are program sections. The program is only for an illustration which uses numerical integration of the function $f(x) = 4/(1 + x^2)$ to calculate the value of $\pi$. We assume that the iteration number $n$ is 25 and the program is executed by 4 and 16 processes, respectively. Fig. 3a and b shows the iterations each process obtains when the program is executed by 4 and 16 processes, respectively. The iterations on $P_i$ when the program is executed by 4 processes is the same as the iterations on $P_i$, $P_{i+4}$, $P_{i+8}$ and $P_{i+12}$ when the program is executed by 16 processes. When the program is executed by four processes, we assume that $P_i$ fail in executing the program section $S_1$. The workload to be recomputed is the workload of $P_i$ executing the program section $S_1$. The major part of that workload is a loop within $S_1$. When $P_i$ is recovered, four processes will be involved in parallel recomputing, designated as recomputing processes. The recomputing processes executing the iterations on $P_i$, $P_{i+4}$, $P_{i+8}$ and $P_{i+12}$ among 16 processes can recover the computation of the loop within $S_1$ lost on $P_i$. Furthermore, programs written in MPI typically employ single-program, multiple-data (SPMD) parallelism with branches based on process rank and process number used to achieve multiple instruction streams. Hence, we just need to revise the values of process rank and process number, and then recompute the variables related to process rank and process numbers. Through doing these, we can use the original loop code to implement the loop code within the recomputing code. The parallel recomputing code for $S_1$ designed by our method based on template is shown in Fig. 4, where lines 9 and 10 are the codes for recalculating the values of process rank and process number.

3.2. Template vs. loop parallelization

In our prior tool GiFT, the parallel recomputing code is generated by loop parallelization, which only deals with loops. To parallelize a loop is to transform it into SPMD codes containing explicit communications. For the program section $S_1$ in Fig. 2, there exists a reduction operation. The parallel recomputing code is designed by parallelizing the reduction operation.

```
1  program main
2   include 'mpi.h'
3 double precision PISDT, mypi, pi, h, sum, x
4 parameter (PISDT = 3.14159d0)
5 integer n, rank, numprocs, i, err

6 call MPI_INIT( err )
7 call MPI_COMM_RANK( ..., rank, err)
8 call MPI_COMM_SIZE( ..., numprocs, err)
9 if ( rank .eq. 0 ) then
10     read(*,*) n
11   endif
12
13   if ( rank .eq. 0 ) then
14     do i=1,numprocs-1
15       call MPI_SEND( n, ..., err)
16     enddo
17 else
18     call MPI_RECV( n, ..., err)
19   endif
20
21   h = 1.0d0/n
22   sum = 0.0d0
23   do 2 i = rank + 1, n, numprocs
24       x = h * (double(i) - 0.5d0)
25       sum = sum + 4.d0 / (1.d0 + x*x)
26   2 continue
27   mypi = h * sum
28   call MPI_ALLREDUCE( mypi, pi, ..., err)
29   if ( rank .eq. 0 ) then
30     write (*, 7) pi, abs(pi-PISDT)
31   7 format( ... )
32   endif
33   call MPI_FINALIZE( err )
34   stop
35 end
```

Fig. 2. A FORTRAN/MPI $\pi$ calculation program.
Fig. 5 shows the parallel recomputing code for $S_1$ generated by loop parallelization. The code from lines 5 to 21 is the result of parallelizing the do loop within $S_1$, and the code from lines 5 to 15 is used to partition the iteration. Compared with the parallel recomputing code in Fig. 4, iteration partitioning is more complicated and the main loop has an extra computation statement. Hence, the parallel recomputing code for $S_1$ generated by loop parallelization has a lower efficiency.

Furthermore, loop parallelization relies on the data-dependence analysis. When the loop has data dependence or the loop has a function call, loop parallelization cannot generate the parallel recomputing code for the loop. The template can do this when the loop satisfies the mapping relation. Compared with the parallel recomputing code generated by loop parallelization, we can conclude two characteristics for our new parallel recomputing code design methodology based on template.

- It can achieve high efficiency of parallel recomputing.
- Within a program section, there must be a workload which satisfies a mapping relation between $numprocs$ and $numprocs * recomp_np$ processes ($recomp_np$ is the number of recomputing processes).
4. A new parallel recomputing code design methodology based on template

The section first proposes the mapping relation and the definition of the template. Then, the design methodology is described.

4.1. The mapping relation

We assume that a parallel program $P$ is executed by $n_1$ processes and $n_2$ processes, respectively, and $n_2 = K \times n_1 (K = 1, 2, \ldots, n_1)$.

**Definition 1.** The mapping relation. The workload $W_t$ distributed on a process when $P$ is executed by $n_1$ processes is the same as on $K$ processes when $P$ is executed by $n_2$ processes. We call such workload $W_t$ satisfying the mapping relation between $n_1$ and $n_2$ processes.

Based on Definition 1, we have the following relationship.

$$W_{1i} = \sum_{j=f(i,1)}^{i} W_{2j}$$  \hspace{1cm} (2)

where $W_{1i}$ and $W_{2j}$ are the workloads $W_t$ distributed on a process when $P$ is executed by $n_1$ and $n_2$, respectively.

The mapping relation can be represented as a five-tuple $M = (P, W_{np1}, n_1, n_2, \text{attribute})$, where $P$ is a parallel program, $W_{np1}$ is the workload on a process satisfying the mapping relation when $P$ is executed by $n_1$ processes, $n_1$ and $n_2$ are process numbers and $n_2 = K \times n_1 (K = 1, 2, \ldots, n_1)$, and attribute is the type of the mapping relation.

When the processes involved in parallel computing is organized as 1D, the attribute has two types: block and cyclic.

For the block type of the mapping relation, the workload on a process when $P$ is executed by $n_1$ processes is the same as the workload on contiguous $K$ processes when $P$ is executed by $n_2$ processes. Fig. 6 shows the block type of the mapping relation. The workload on process $P_i$ when $P$ is executed by $n_1$ processes is the same as the workload on processes $i + K, \ldots, (i+1) \times K - 1$ when $P$ is executed by $n_2$ processes. We have the following formula for the block type of the mapping relation.

$$P_i \rightarrow P_{i\times K}, \ldots, P_{(i+1)\times K - 1}$$ \hspace{1cm} (3)

We have Eq. (3), so Eq. (2) is as follows:
of the processes. Assuming that the processes are organized as
Fig. 9 shows the mapping relation
the mapping relation. The workload on process
the workload on round-robin/C3
organized as 2
We have Eq. (5), so Eq. (2) is as follows:
For the cyclic type of the mapping relation, the workload on a process when
Fig. 7 shows the cyclic type of the mapping
relation. The workload on process Pi when P is executed by np1 processes is the same as
the workload on round-robin K processes when P is executed by np2 processes. We have the following formula for the cyclic type of
the mapping relation.

\[ W_{i1} = \sum_{j=i-K+1}^{i+K-1} W_{2j} \]  

(4)

When the processes involved in parallel computing are organized as 2D, the attribute is closely related to the organization of the processes. Assuming that the processes are organized as R1=C1 when the process number is np1, and the processes are organized as R2=C2 when the process number is np2, where K/1 = Kr, C2/C1 = Kc.
Fig. 8 shows the mapping relation M = (P, Wnp1, np1, np2, (cyclic, block)). In this case, a process Pi among np1 processes can be mapped to Kc*Kc processes among np2 processes. Their process ranks are as follows:

\[
\begin{bmatrix}
K_r \left( \frac{i}{C_1} \right) + C_2 + K_c \left( i - \frac{i}{C_1} \right) + C_1 & \cdots & K_r \left( \frac{i}{C_1} \right) + C_2 + K_c \left( i - \frac{i}{C_1} \right) + C_1 + 1 - 1 \\
\cdots & \cdots & \cdots & \cdots \\
\left( K_r \left( \frac{i}{C_1} + 1 \right) - 1 \right) + C_2 + K_c \left( i - \frac{i}{C_1} \right) + C_1 & \cdots & \left( K_r \left( \frac{i}{C_1} + 1 \right) - 1 \right) + C_2 + K_c \left( i - \frac{i}{C_1} \right) + C_1 + 1 - 1 \\
\end{bmatrix}
\]  

(7)

Fig. 9 shows the mapping relation M = (P, Wnp1, np1, np2, (block, cyclic)). In this case, a process Pi among np1 processes can be mapped to Kc*Kc processes among np2 processes. Their process ranks are as follows:

\[
\begin{bmatrix}
K_r \left( i - \frac{i}{R_1} \right) + R_1 + K_c \left( \frac{i}{R_1} \right) + R_2 & \cdots & K_r \left( i - \frac{i}{R_1} \right) + R_1 + K_c \left( \frac{i}{R_1} \right) + R_2 \\
\cdots & \cdots & \cdots & \cdots \\
K_r \left( i - \frac{i}{R_1} \right) + R_1 + 1 + K_c \left( \frac{i}{R_1} \right) + R_2 & \cdots & K_r \left( i - \frac{i}{R_1} \right) + R_1 + 1 + K_c \left( \frac{i}{R_1} \right) + R_2 \\
\end{bmatrix}
\]  

(8)
4.2. The definition of the template

**Definition 2.** The template. The template is the code for the workload $W_{np1}$ in $M = \langle P, W_{np1}, np1, np2, attribute \rangle$.

Based on Definition 2, the code for the workload $W_{np1}$ is a template between $np1$ and $np2$ processes. When the number of processes changes, the same code may not satisfy the definition of the template.

In Fig. 10, there is the workload $W_{loop}$ which is a three-nested loop of the matrix–matrix multiplication of matrices $A$ and $B$. When the row number $arow$ of matrix $A$ can be divided exactly by $np1$ and $np2$, $W_{loop}$ can satisfy the mapping relation $\langle P, W_{loop}, np1, np2, block \rangle$. Hence, the code from lines 14 to 20 for $W_{loop}$ is a template. When $arow$ cannot be divided exactly by $np1$ or $np2$, $W_{loop}$ does not satisfy the mapping relation. Hence, the code from lines 14 to 20 is not a template. For example, when we multiply two 40$\times$40 matrices, $W_{loop}$ has a mapping relation $\langle P, W_{loop}, 4, 8, block \rangle$. In this case, the code from lines 14 to 20 is a template. However, $W_{loop}$ does not satisfy the mapping relation between 4 and 16 processes. A template does not exist in Fig. 10.

**Definition 3.** The full template. When the code for the workload $W_{np1}$ can satisfy the mapping relation between any two process numbers $np1$ and $np2$, we call the code a full template.

Within the program section $S_1$ in Fig. 2, the workload of the do loop can satisfy the mapping relation between any two process numbers. Hence, the code from lines 23 to 26 is a full template.

Slicing criterion $\langle s, t \rangle$ specifies a location (statement $s$) and a variable ($v$). A backward slice of a program $P$ with respect to the slicing criterion $\langle s, t \rangle$ consists of all statements in the program that may affect the value of the variable $v$ at $s$.

**Definition 4.** The mapping variable of a template. The variable $v$ is used in a template. When a backward slice of a program $P$ with respect to the slicing criterion $\langle s, t \rangle$ contains the non-conditional statements using process rank or process number, we call the variable $v$ a mapping variable of a template.

**Definition 5.** The template slice. The template slice is a backward slice of a program $P$ with respect to all mapping variables of a template at their locations.

When the code in Fig. 10 from lines 14 to 20 is a template, there are eight variables used in the template, namely, $arow\_num$, $brow$, $column$, $i$, $j$, $k$, $A$, and $B$. Fig. 11a shows the program slice with respect to $\langle 14, arow\_num \rangle$, and the slice includes a non-expressional statement using process number $numprocs$. Hence, $arow\_num$ is a mapping variable of a template. Fig. 11b and c shows the program slices with respect to $\langle 15, brow \rangle$ and $\langle 16, column \rangle$, respectively. No statement uses process number
or process rank in the slices. Hence, brow and column are not mapping variables of a template. Fig. 11d shows the program slice with respect to \( h_{17}, \text{i} \), and the slice includes a non-expressional statement using process number \( \text{numprocs} \). Hence, \( \text{i} \) is a mapping variable of a template. Fig. 11e–h shows the program slices with respect to \( h_{17}, \text{j} \), \( h_{17}, \text{k} \), \( h_{17}, \text{A} \), and \( h_{17}, \text{B} \), respectively. No statement uses process number or process rank in the slices. Hence, \( \text{j} \), \( \text{k} \), \( \text{A} \), and \( \text{B} \) are not mapping variables of a template. The template in Fig. 10 only includes two mapping variables of a template: \( \text{arow}_\text{number} \) and \( \text{i} \). We can also observe that the program slice with respect to \( h_{14}, \text{arow}_\text{num} \) is a subset of the program slice with respect to \( h_{17}, \text{i} \). Therefore, the template slice is the program slice with respect to \( h_{17}, \text{i} \).

4.3. The design methodology

The parallel recomputing code design methodology based on template entails the mapping of the workload of the template in a program section among the recomputing processes. For the workload of the non-template in a program section, our methodology is to recompute serially. Our methodology is subdivided into three phases as shown in Fig. 12: determining the templates in program sections, computing the template slices, and designing the parallel recomputing code.

4.3.1. Determining the templates in program sections

Determining the workload in a program section can satisfy the mapping relation between \( \text{numprocs} \) and \( \text{numprocs*recomp_np} \) processes.

4.3.2. Computing the template slices

Computing the template slices involves two steps: (1) Determining the mapping variables in a program section. For each variable \( v \) used in a template, we compute the program slice with respect to variable \( v \) at location \( s \). Based on the computed program slices, we can determine the mapping variables in a program section. (2) Determining the template slice. We can obtain the template slice by computing a program slice with respect to all mapping variables of a template at their locations.

4.3.3. Designing the parallel recomputing code

The parallel recomputing code consists of a template and a non-template. The template includes four parts: the code for computing the mapping relation, the template slice, the template code in the original program section, and the communication code. For example, in Fig. 4, lines 6–16 are the designed parallel recomputing code for the template of do loop; Lines 9 and 10 are the code for computing the mapping relation; Lines 7, 8, and 15 are the communication code; and Lines 11–14 are the template slice and the original template. The template of the do loop only includes a mapping variable \( i \) of a template, so the template slice is the program slice with respect to \( i \) at its location. Line 11 is the template slice, and Lines 11–14 are the original templates. The other code in Fig. 4 is the parallel recomputing code for non-template in a program section, and they are executed only by a process.
When multiple failures occur in a program section, we need to design the parallel recomputing code for each failed process. The only difference between the parallel recomputing code for different failed processes is the rank of the failed process. We recover the failed process according to the order of the failed rank. We can then recover $b_{numprocs/\text{recomp}_\text{np}}$ failures concurrently. When the number of the failed processes is $fn$ and $fn*\text{recomp}_\text{np} \leq numprocs$, we can recover the $fn$ failures concurrently.

5. Implementation

The new parallel recomputing code design methodology based on template aims to design the parallel recomputing code for the templates in a program. This section automates the design methodology in our tool GiFT (Get it Fault Tolerant) which automates the FTPA implementation. It can generate the parallel recomputing code for an MPI/Fortran program with templates. In our implementation, a user uses the compiler directives to determine templates in a program. *Algorithm 1* shows the algorithm for automating our design methodology. The algorithm has four phases: constructing the control flow graph,
computing the interprocedural summary information, computing the template slices, and generating the parallel recomputing code.

**Algorithm 1.** $\text{RecompCodeGene}(P)$

<table>
<thead>
<tr>
<th><strong>Input:</strong> a MPI parallel program $P$</th>
<th><strong>Output:</strong> The parallel recomputing code for each program section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructing MPI-Branch based Interprocedural Control-Flow Graph for $P$;</td>
<td>Computing interprocedural summary information;</td>
</tr>
<tr>
<td><strong>foreach</strong> program section <strong>do</strong></td>
<td>Computing the template slice;</td>
</tr>
<tr>
<td><strong>foreach</strong> a template in program section <strong>do</strong></td>
<td><strong>end</strong></td>
</tr>
<tr>
<td></td>
<td>Determining the mapping variable of a template</td>
</tr>
<tr>
<td></td>
<td>$\text{MappingvarDeter}$;</td>
</tr>
<tr>
<td></td>
<td><strong>end</strong></td>
</tr>
<tr>
<td></td>
<td>Generating the parallel recomputing code;</td>
</tr>
</tbody>
</table>

5.1. The compiler directives

The approach based on the template uses compiler directives to determine the templates in the program sections. In GiFT, there are three kinds of compiler directives: template directives, the number of recomputing process directives, and the mapping relation attribute directives, and the reduction variable directives.

The template directives include $\text{TBB}$ and $\text{TBE}$, which are in the form of !$\text{gift templateblockbegin}$ and !$\text{gift templateblockend}$, respectively. $\text{TBB}$ and $\text{TBE}$ implicitly define the scope of a template. $\text{TBB}$ means the beginning of the template code and $\text{TBE}$ means the end of the template code. The workload of the code within $\text{TBB}$ and $\text{TBE}$ will be distributed among the recomputing processes.

The number of recomputing process directive is $\text{TRN}$, which is in the form of !$\text{gift template recompnumRN}$. $\text{RN}$ is the number of the recomputing processes. $\text{TRN}$ defines a template that satisfies the mapping relation between $np_1$ and $np_1 \times \text{RN}$ processes.

The mapping relation attribute directive is $\text{TMT}$, which is in the form of !$\text{gift template mapping type(attribute[attribute[...]]]}$. The attribute can be block or cyclic. $\text{TMT}$ defines the type of the mapping relation.

The mapping array variable directive is $\text{TA}$, which is in the form of !$\text{gift template array(name[name[...]]]}$. Name is the array of names in a template. $\text{TA}$ defines the arrays which are the mapping variables of a template.

The reduction variable directive is $\text{TR}$, which is in the form of !$\text{gift reduction(operator/intrinsic_procedure_name):list}$. The specific usage of $\text{TR}$ in GiFT is the same as in OpenMP. $\text{TR}$ defines the reduction variables in a template.

Fig. 13 shows the directives added to the full template of do loop in the program section $S_1$ in Fig. 2. The number of recomputing processes is 8, the attribute of the mapping relation is cyclic, and $\text{sum}$ is a reduction variable in the full template. No mapping array variable exists in the full template, so Fig. 13 does not include the directive $\text{TA}$.
5.2. Constructing the control flow graph

The control flow graph is the basis for automating our design methodology. We constructed an MPI-BICFG (MPI-Branch based Interprocedural Control-Flow Graph) using the method in [8]. Each node in an MPI-BICFG is a single statement in a parallel program.

5.3. Computing the interprocedural summary information

Subroutines and functions present in an MPI program can be divided into three categories: user-defined, intrinsic, and MPI calls. We compute the interprocedural summary information for the three categories of subroutines and functions. For user-defined ones, we compute the interprocedural summary information using the method presented in [10]. For each procedure \( P \), a set \( \text{MOD}(P) \) of variables that may be modified by \( P \) is computed, and a set \( \text{USE}(P) \) of variables that may be used by \( P \) are obtained. Intrinsic and MPI calls do not expose their source codes. Fortunately, they only use or define the variables in their real parameter lists. We can compute sets \( \text{MOD}(P) \) and \( \text{USE}(P) \) according to the reference manuals.

For each communication edge \( i \rightarrow \text{MPI-BICFG} j \) in the MPI-BICFG for procedure \( P \), the use information for node \( j \) is

\[
\text{USE}(j) = \text{USE}(j) \cup \text{USE}(i).
\]

5.4. Computing the template slice

Based on the design methodology in Section 4, GiFT entails two steps for computing the template slices. The first is to determine the mapping variables of a template exploring the use of program slicing, and the second is to compute the template slice.

Algorithm 2. MappingvarDeter(CFG, TS)

\[
\begin{aligned}
\text{Input:} & \quad \text{CFG: an MPI-BICFG for a parallel program } P \\
& \quad \text{TS: A template in a program section} \\
\text{Output:} & \quad \text{The mapping variables of a template in a program section} \\
\text{foreach} & \quad \text{used variable } V \text{ in template } TS \text{ do} \\
& \quad \text{Intraprocedural program slicing with respect to } V \text{ at its position } s \\
& \quad \text{Intraprocedural program slicing with respect to } V \text{ at its position } s \\
& \quad \text{if (myrank or numprocs } \in \text{ slice of } < n, V > \text{ ) } \& \& \text{ (myrank or numprocs } \notin \text{ condition expression) then} \\
& \quad \text{Insert } V \text{ into a mapping variable list;}
\end{aligned}
\]

Fig. 13. The compiler directives for a full template of do loop in program section S1.
5.4.1. Determining the mapping variables of a template

We have defined the mapping array variables of a template in the TA directives, so we do not need to compute the program slice for them. For a scalar variable used in a template, the program slice for it will be computed. Then, based on the computed program slice and Definition 4, we can determine whether the scalar variable is a mapping variable of a template. To perform program slicing for an MPI program, interprocedural static slicing is a concern.

Algorithm 2 gives the procedures for determining the mapping variable of a template. There are three procedures: (1) intraprocedural program slicing; (2) interprocedural program slicing; and (3) determining the mapping variables of a template.

Algorithm 3. Intraprocedural program slicing \textit{IntraSlicing}(G, i, V)

\begin{verbatim}
Algorithm 3. Intraprocedural program slicing \textit{IntraSlicing}(G, i, V)

\textbf{Input}: G: an MPI-BICFG for procedure G

A node \(i\) in \(G\) and a scalar variable \(V\) used in \(i\)

\textbf{Output}: Intraprocedural slice with respect to \(<i, V>\)

\textbf{foreach} node in \(G\) \textbf{do}

\hspace{1em} \textbf{if} \ \text{i is n} \text{\ then}

\hspace{2em} \(R^0_C(n) = V;\)

\hspace{1em} \textbf{else}

\hspace{2em} \(R^0_C(j) = \emptyset;\)

\hspace{1em} \textbf{end}

\textbf{end}

\textbf{repeat}

\hspace{1em} \textbf{foreach} edge \(i \rightarrow j\) in \(G\) \textbf{do}

\hspace{2em} \(R^0_C(i) = R^0_C(i) \cup (R^0_C(j) - MOD(i));\)

\hspace{2em} \textbf{if} \(MOD(i) \cap R^0_C(j) \neq \emptyset\) \textbf{then}

\hspace{3em} \(R^0_C(i) = R^0_C(i) \cup USE(i);\)

\hspace{2em} \textbf{end}

\hspace{1em} \textbf{end}

\textbf{until} \(R^0_C\) \text{ changed;}

\textbf{foreach} edge \(i \rightarrow j\) in \(G\) \textbf{do}

\hspace{1em} \textbf{if} \(MOD(i) \cap R^0_C(j) \neq \emptyset\) \textbf{then}

\hspace{2em} \(S^0_C = S^0_C \cup \{i\};\)

\hspace{1em} \textbf{end}

\hspace{1em} \textbf{for} \(k = 0; R^0_C \neq R^0_C; k = k + 1\) \textbf{do}

\hspace{2em} \textbf{foreach} edge \(i \rightarrow j\) in \(G\) \textbf{do}

\hspace{3em} \textbf{if} \(INFL(b) \cap S^k_C(i) \neq \emptyset\) \textbf{then}

\hspace{4em} \(B^k_C = B^k_C \cup \{b\};\)

\hspace{3em} \textbf{end}

\hspace{3em} \textbf{foreach} \(b\) in \(B^k_C\) \textbf{do}

\hspace{4em} \(R^k_C(i) = R^k_C(i) \cup R^0_C(i, USE(b));\)

\hspace{3em} \textbf{end}

\hspace{3em} \textbf{if} \(MOD(i) \cap R^k_C(j) \neq \emptyset\) \textbf{then}

\hspace{4em} \(S^{k+1}_C = S^k_C \cup B^k_C \cup \{i\};\)

\hspace{3em} \textbf{end}

\hspace{2em} \textbf{end}

\hspace{1em} \textbf{end}

\textbf{end}
\end{verbatim}
We use Weiser’s algorithm for intraprocedural slicing [11–13]. We assume that a slicing criterion is \( (i, V) \) where \( i \) is a node in the MPI-BICFG of the parallel program, and \( V \) a scalar variable used in \( i \). Algorithm 3 shows the intraprocedural slicing algorithm, where lines 1 to 11 are to compute the variables and their locations which are directly relevant to \( V \). The set of directly relevant variables at node \( i \) in the MPI-BICFG is denoted as \( R^0_i \). \( S^0_i \) is defined as the set of all nodes \( i \) that define a variable \( v \) that is a relevant at a MPI-BICFG-successor of \( i \). Lines 12–19 is to compute the variables and their locations which are indirectly relevant to \( V \). The range of influence \( \text{INFL}(b) \) of a branch statement \( b \) is defined as the set of statements which are control dependent on \( b \). The branch statements \( B^k_i \) are indirectly relevant due to the influence they have on nodes \( i \) in \( S^k_i \). \( R^{k+1}_i \) is the set of indirectly relevant variables. \( S^{k+1}_i \) are the sets of indirectly relevant statements, and these sets consist of the nodes in \( B^k_i \) together with the nodes \( i \) that define a variable that is \( R^{k+1}_i \)-relevant to a MPI-BICFG-successor \( j \). The fixpoint of the computation of the \( S^{k+1}_i \) sets constitutes the desired program slice.

Algorithm 4. Interprocedural program slicing \( \text{InterSlicing}(GW, s, V) \)

```
if (hasfailed(eq)) then
  if (rank.eq.failedrank) then
    _fpa_h = 1.0d0/n
    _fpa_sum = 0.0d0
  endif
  if (myrank.lt._fpa_recomp_numprocs) then
    call MPI_Bcast(_fpa_h,1,0,failedrank,...)
  end if
  _fpa myrank = failedrank + myrank * numprocs
  _fpa_numprocs = numprocs * _fpa_recomp_numprocs
  do _fpa_i = _fpa myrank + 1, n, _fpa_numprocs
    _fpa_x = _fpa_h * (dblcl(_fpa_i) - 0.5d0)
    _fpa_sum = _fpa_sum + 4.0d0/(1.0d0 + _fpa_x * _fpa_x)
  enddo
  call MPI_REDUCE(_fpa_sum,sum,1,...,failedrank,...)
endif
```

Fig. 14. The optimized code.

We use Weiser’s algorithm for intraprocedural slicing [11–13]. We assume that a slicing criterion is \( (i, V) \) where \( i \) is a node in the MPI-BICFG of the parallel program, and \( V \) a scalar variable used in \( i \). Algorithm 3 shows the intraprocedural slicing algorithm, where lines 1 to 11 are to compute the variables and their locations which are directly relevant to \( V \). The set of directly relevant variables at node \( i \) in the MPI-BICFG is denoted as \( R^0_i \). \( S^0_i \) is defined as the set of all nodes \( i \) that define a variable \( v \) that is a relevant at a MPI-BICFG-successor of \( i \). Lines 12–19 is to compute the variables and their locations which are indirectly relevant to \( V \). The range of influence \( \text{INFL}(b) \) of a branch statement \( b \) is defined as the set of statements which are control dependent on \( b \). The branch statements \( B^k_i \) are indirectly relevant due to the influence they have on nodes \( i \) in \( S^k_i \). \( R^{k+1}_i \) is the set of indirectly relevant variables. \( S^{k+1}_i \) are the sets of indirectly relevant statements, and these sets consist of the nodes in \( B^k_i \) together with the nodes \( i \) that define a variable that is \( R^{k+1}_i \)-relevant to a MPI-BICFG-successor \( j \). The fixpoint of the computation of the \( S^{k+1}_i \) sets constitutes the desired program slice.

Algorithm 4. Interprocedural program slicing \( \text{InterSlicing}(GW, s, V) \)

```
Input: GW: an MPI-BICFG for program
       A node s in procedure P and a scalar variable V used in s
Output: Interprocedural slice with respect to <s, V>
if P calls Q at node i then
  foreach successor of node i do
    ROUT(i) = ROUT(i) ∪ RC(j);
  end
  actual parameters of Q occurred in ROUT(s) are substituted for
  formal parameters of Q;
  V = ROUT(i) ∩ SCOPEQ;
  Intraslicing(GQ, nQ, V);
end
if Q calls P at node i then
  The formal parameters of P occurred in RC(fP) are substituted for
  actual parameters of P;
  V = RC(fP) ∩ SCOPEQ;
  Intraslicing(GQ, i, V);
end
if (node j ∈ P) & & (node i ∈ Q) & &
  (communication edge i → MPI–IBC(F j) then
    V = USE(i);
    Intraslicing(GQ, i, V);
```
Algorithm 4 shows the interprocedural slicing algorithm, which iteratively generates new slicing criteria as follows.

(1) When procedure $P$ which includes the slicing criterion calls procedure $Q$ at node $i$, a new criterion is generated as follows.

$$C_0 = \{n_Q, ROUT(i)_{F \rightarrow A} \cap SCOPE_Q\}$$

where $n_Q$ is the last statement of $Q$, $F \rightarrow A$ represents the actual parameters of $Q$ are substituted for the formal parameters of $Q$, $SCOPE_Q$ are all variables in procedure $Q$. There is an edge $i \rightarrow j$ in an MPI-BICFG, $\text{ROUT}(i) = \cup_j R(i,j) = \cup_j R_C(i,j)$.

(2) When procedure $Q$ calls procedure $P$ which includes the slicing criterion at node $j$, a new criterion is generated as follows.

$$C' = (i, RC(f_P)_{A \rightarrow F} \cap SCOPE_Q)$$

where $f_P$ is the first statement of procedure $P$, $A \rightarrow F$ represents the formal parameters of $P$ are substituted for the actual parameters of $P$.

(3) For a communication statement $j$ in procedure $P$, when there exists a communication edge $i \rightarrow j_{\text{MPI-BICFG}}$, and the communication statement $i$ is in procedure $Q$, a new criterion is generated as follows.

$$C' = (i, V)$$

where $V$ is scalar variables used at $i$.

From the computed program slice, we can determine the mapping variables of a template. Then, we add them into the list of the mapping variables of a template.

### 5.4.2. Computing the template slice

The template slice is the union of the program slices for all mapping variables of a template. However, the mapping scalar variables of a template are only considered, so the template slice we need to get is the union of the program slices for all mapping scalar variables of a template, which is a subset of the template slice.

![Fig. 15. The experimental results for MM.](image-url)

(a) The execution time for MM.  (b) The efficiency for MM.
5.5. Generating the parallel recomputing code

For the non-template in a program section, GiFT generates the parallel recomputing code by replacing the variables defined in the non-template with new variables. The conditional statements are necessary to ensure that the non-template is only executed by a single recomputing process. For the template in a program section, we need to generate the four parts of the code.

5.5.1. Generating the code for computing the mapping relation

Based on the attribute of the mapping relation, GiFT generates the code of computing the equation for the mapping relation between np and np * recomp_np processes.

When the processes are organized as 1D, the equation for the block type of mapping relation is as follows:

\[ \text{recomp}_\text{myrank} = \text{failedrank} \times \text{recomp}_\text{np} + \text{myrank} (0 \leq \text{myrank} < \text{recomp}_\text{np}) \]  

(12)

The equation for the cyclic type of mapping relation is as follows:

\[ \text{recomp}_\text{myrank} = \text{failedrank} + \text{myrank} \times \text{np} (0 \leq \text{myrank} < \text{recomp}_\text{np}) \]  

(13)

When the processes are organized as 2D, the equations for the mapping relation are generated by replacing \( i \) in Eqs. (7) and (8) with the rank of a failed process, respectively.

(a) The execution time for EP.  
(b) The efficiency for EP.

Fig. 16. The experimental results for EP.

(a) The execution time for FT.  
(b) The efficiency for FT.

Fig. 17. The experimental results for FT.
5.5.2. Generating the code of the template slice

The code of the template slice is generated by replacing the rank of process \texttt{myrank} and the number of processes \texttt{numprocs} in the computed template slice with \texttt{recomp\_myrank} and \texttt{np + recomp\_np}, respectively.

5.5.3. Generating the code of the template

The code of the template is generated by replacing variables in the original template of the program section with new variables.

5.5.4. Generating the communication code

For the variables defined prior to the template in a program section and used in that template, GiFT needs to generate the communication code between the recomputing processes to distribute the variables.

When a variable is the mapping array variable, the type of communication is based on the attribute of the mapping relation. For example, when an array \texttt{A} is a mapping array variable in a template and the attribute of the mapping relation is \texttt{block}, \texttt{MPI\_Scatter} is used to distribute array \texttt{A}.

When a variable is the mapping scalar variable, \texttt{MPI\_Bcast} is used to distribute the variables. In this case, if the variable is defined in the non-template of a program section and then used in the template of the same program section, GiFT can generate the code for recomputing the variable in the recomputing processes. Thus, the communication for that variable can be omitted.
In Fig. 2, the variables $h$ and $\text{sum}$ are defined in the program section $S_p$, and are used in the template of do loop. When generating the communication code for the do loop, we can recompute them in the recomputing processes, and the communication code for them can be omitted. Fig. 14 shows the optimized code.

6. Experimental evaluation

6.1. Experimental setting

In this section, we evaluate the performance of FTPA based on template, and compare the performance of FTPA based on template to the performance of FTPA based on loop parallelization.

We use five programs, matrix–matrix multiplication (MM), and four kernels of NAS Parallel Benchmarks (NPB), EP, CG, FT, and MG as specified in Table 1, in which class is the NPB specification information [14]. MM is matrix–matrix multiplication $C = A \times B$, in which the input matrix $A$ is block-row distributed across all processes, and process $P_i$ has the matrix $A_i$, the matrix $B$, and the resultant matrix $C_i$. NPB are a set of benchmarks derived from computational fluid dynamics (CFD) and developed at the NASA Ames research center targeting performance evaluation of parallel and distributed systems. In our experiment, MPI-NPB3.3 was used. FT contains the computational kernel of a 3D Fast Fourier Transform (FFT). MG uses a V-cycle Multi Grid method to compute the solution of the 3D scalar Poisson equation. CG uses a Conjugate Gradient method to compute approximations to the smallest eigenvalues of a sparse unstructured matrix. EP generates pairs of Gaussian random deviates according to a specific scheme.

The five programs are represented as XX in the following discussion. XX-T represents the version based on the template for program XX, and XX-A represents the version based on loop parallelization for program XX. The five programs run on Tianhe-1A. Each node is equipped with dual 2.93 GHz Intel X5670 CPUs, 48 GB memory, 160 GB disk storage, and a YH-Net. The operating system used is Kylix.

We explore the use of GiFT to generate two FTPA versions for each program which has one program section. One is the version based on the template, and the other is the version based on loop parallelization. The different approaches for generating the parallel recomputing code only affect the execution time of FTPA when failures occur, so the failure-free execution times of two FTPA versions are the same. In our experiments, we desire to support the finding that the failure occurs prior to the end of the program. Likewise, we also desire to highlight that the recovery task of FTPA is the complete workload of the failed process. The following three execution times were recorded for all five programs:

- **XX**: The re-execution time of the failed code with classical method.
- **XX-T**: The recovery time of the failed code based on template.
- **XX-A**: The recovery time of the failed code based on loop parallelization.

We use causal message logging protocol with event logger to ensure that only the failed process needs to be recovered. The classical method is coordinated restart, which is sequential restart from checkpoint. In our experiment, the recovery time has two parts, namely, the re-execution time of the failed code and the protocol time.

6.2. Performance

The experimental charts of MM are shown in Fig. 15. Fig. 15a shows three execution times, where MM is the re-execution time of the failed code with classical method, MM-T is the recovery time of the failed code based on template, and MM-A is the recovery time of the failed code based on loop parallelization. Fig. 15b represents the efficiency of parallel computing, which is computed as the ratio of speedup of fast recovery and the number of processes. When we calculate the efficiency, we use the total number of available processes. MM is the efficiency of parallel computing for MM, MM-T is the efficiency of parallel recomputing based on template, and MM-A is the efficiency of parallel recomputing based on loop parallelization. Later, we will use similar representation.

For the given problem size, when 40,960 can be divided evenly by two process numbers, the MM program can satisfy the mapping relation. For 32 and 64 processes, all processes contribute to parallel recomputing for MM-T when one process fails. The efficiency of parallel recomputing for MM-T is higher than MM-A. When the process is 128, the MM program can satisfy the mapping relation for 128 processes and 10,240 processes. Therefore, only 80 processes can be involved in parallel recomputing for MM-T. Thus, the efficiency of parallel recomputing for MM-T is lower than that for MM-A. When the process is 256, because of the limit of the problem size, only 160 processes can be involved in parallel recomputing for MM-T and MM-A. MM-T has higher efficiency than MM-A.

The experimental charts of EP are shown in Fig. 16. The data size in EP is large enough to make the major computation in EP a full template. When one process fails, all processes contribute to parallel recomputing in EP-T. The efficiency of parallel recomputing for EP-T is just a little higher than EP-A. With increased number of processes, the workload of the failed code for a process decreases. The major part of the recovery time is the message logging protocol time.

For MM and EP programs, the parallel recomputing code for the major computation can be generated by loop parallelization. However, compared with the parallel recomputing code generated by the template, loop parallelization has more com-
The efficiency of parallel recomputing for the template is a little higher than loop parallelization when the number of processes involved in parallel recomputing is the same. Our new method can reduce the recovery time of the restart.

The experimental charts of FT are shown in Fig. 17. The processes executing NPB FT are organized as 1D when the number of processes is less than 1024, so the code of major computation in NPB FT is a full template when the number of processes is less than 1024. For 32 processes, all processes contribute to parallel recomputing for FT-T when one process fails. Some loops in FT have function calls, so the parallel recomputing code for the loops cannot be generated by loop parallelization. FT-A re-executes them in serial. The template can deal with the loops. Therefore, the efficiency of parallel recomputing for FT-T is higher than that for FT-A. However, for FT-T, when the number of processes is 64, 128, and 256, only 16, 8, and 4 processes contribute to parallel recomputing, respectively. The number of processes involved in parallel recomputing decreases, so the efficiency of parallel recomputing for FT-T decreases. All processes are involved in parallel recomputing, so the efficiency of parallel recomputing for FT-A can be higher than that for FT-T.

The experimental charts of CG are shown in Fig. 18. The processes executing NPB CG are organized as 2D, so the attribute of the mapping relation in CG are 2D. The code of major computation in CG is a full template when the number of processes is less than 1024. The experimental result of CG is similar to that of FT. For 32 processes, all processes contribute to parallel recomputing for CG-T when one process fails. In CG, some loops have data dependence, so CG-A cannot parallelize the loops in parallel recomputing. The template can deal with the loops. The efficiency of parallel recomputing for CG-T is higher than that for CG-A. For CG-T, with the increased number of processes, the number of processes involved in parallel recomputing decreases. All processes are involved in parallel recomputing, so the efficiency of parallel recomputing for CG-A can be higher than that for CG-T.

For the FT and CG programs, the code of major computation in the programs satisfy the mapping relation when the number of processes is less than \( n \). Some loops in the programs cannot be parallelized in parallel recomputing, so the efficiency of parallel recomputing for the template is higher than loop parallelization when the number of processes is less than \( \sqrt{n} \). However, when the number of processes executing the programs increases, the number of processes that can be involved in parallel recomputing for the template decreases and the efficiency of parallel recomputing for the template decreases. All processes can be involved in parallel recomputing, so the efficiency of parallel recomputing for loop parallelization can be higher than that for the template.

The experimental charts of MG are shown in Fig. 19. The data size in MG is large enough to make the computation in MG a full template. When one process fails, all processes contribute to parallel recomputing in MG-T. In MG, some loops have data dependence, so MG-A cannot parallelize the loops in parallel recomputing. The loops can be parallelized in MG-T. Hence, the efficiency of parallel recomputing for MG-T is higher than that for MG-A.

From the experiments mentioned above, we conclude the following:

- For the programs where all loops can be parallelized by loop parallelization, the efficiency of parallel recomputing for the template is a little higher than loop parallelization when the number of the processes involved in parallel recomputing is the same.
- For the programs where some loops have data dependence or function calls, parallel recomputing based on template can achieve a high performance advantage. When the recomputing workload is high, our new method based on the template can greatly reduce the recovery time of the restart.
- For the programs which satisfy the mapping relation under a certain number of processes, the number of processes involved in parallel recomputing reduces, so the efficiency of parallel recomputing for the template decreases when the number of processes increases.

7. Conclusions

First, this paper proposes a new parallel recomputing code design methodology, and the parallel recomputing code designed by the methodology can achieve a high efficiency of parallel recomputing. Second, the parallel recomputing code design methodology is automated by exploring the use of compiler technology. Finally, we evaluate the performance of our approach with five programs on Tianhe-1A. The experimental results show that the parallel recomputing code generated by our approach can achieve a higher efficiency than the code generated by loop parallelization.

To ease the work of determining the template in a program for GiFT, our future research is geared toward automatic detection of templates. We will also analyze the relationship of the template to loop parallelization. Based on the analysis, we will be able to investigate the hybrid use of template and loop parallelization in an application.

Acknowledgements

This work was supported by the China National Science Foundation (NSF) Grants 61003087, 61272141, 61120106005 and 863 Plan 2012AA01A309.
References


Yunfei Du received his doctor degree from National University of Defense Technology, China in 2008. His research interests focus on parallel and distributed systems, fault tolerance, and scientific computing. He is an assistant professor of School of computer, National University of Defense Technology now.

Yuhua Tang is a professor of School of computer, National University of Defense Technology. Her research interests focus on computer architecture, parallel and distributed systems, and fault tolerance.

Xinwei Xie received his doctor degree from University of New South Wales, Australia in 2012. His research interests focus on program analysis, parallel and distributed systems. He is an assistant professor of School of computer, National University of Defense Technology now.