TINPAR:
A Parallelizing Compiler for Message-Passing Multiprocessors

Atsushi Kubota Ikuo Miyoshi† Koji Maeyama Shin-ya Goto
Shin-ichiro Mori Hiroshi Nakashima Shinji Tomita
(Kyoto University, Japan)

Abstract

We are developing a parallelizing compiler TINPAR for message-passing multiprocessors. TINPAR parallelizes sequential programs written in an Fortran-like language, which allows users to write programs assuming global name space. It automatically partitions computation and inserts communication codes following owner computes rule. In order to derive efficient codes, this compiler eliminates unnecessary codes generated in parallelizing step and re-schedules the codes to hide communication latency. As a result, TINPAR achieved 45.9 times speedup for matrix multiplication for the problem size 512×512, 35.8 times for Gaussian elimination for the problem size 1024×1024, and 28.1 times for SOR method for the problem size 1024×1024 on the 64 PE AP1000.

1 Introduction

Parallel machines can provide enormous computational power. However, programming for these machines is not easy. Especially for message-passing multiprocessors, users are required to deal with separate address space and insert message-passing codes in writing programs. Obviously, this difficulty prevents these machines from becoming popular.

In our research, we are developing a parallelizing compiler for message-passing multiprocessors, TINPAR[1]. TINPAR parallelizes programs written in extended form of Tiny language, allowing users to write sequential programs with global name space.

Parallelization by TINPAR consists of three steps, restructuring, parallelizing, and optimizing steps: In the first step, source programs are transformed into programs which suit well with the next step to extract maximum parallelism easily. In the second step, the restructured programs are parallelized straightforwardly following owner computes rule. In the third step, elimination of unnecessary codes and transformations to hide communication latency are performed. Dividing the components of the compiler into these steps makes it possible to 1) adopt generic optimizations, 2) evaluate the effectiveness of them, and 3) implement easily. Now we are developing the last two steps and research on the restructuring strategy in the first step through the performance evaluation for parallelized programs.

In the remainder of this paper, we describe the overview of TINPAR in section 2. A run-time communication library is illustrated in section 3. In section 4, we present optimization techniques used in TINPAR. Then, we evaluate the performance of the programs parallelized by TINPAR in section 5. Related work and conclusion are presented in section 6 and 7 respectively.

2 TINPAR:TINy PARallelizer

2.1 Extended Tiny Language

In our system, source programs are written in a Fortran-like language used in Tiny Tool[2]. We extended array declaration so as to specify data distribution, such as one dimensional BLOCK, CYCLIC, and BLOCK_CYCLIC distributions. We also introduced `WHILE` statement and scope rule. The scope rule makes it possible to implement in-line functions.

2.2 Parallelizer Output

TINPAR outputs SPMD style C programs as its object codes. Generating object codes in C has the following advantages.

- It makes TINPAR free from machine dependent low level optimizations leaving it to C compilers.
- It provides high portability. To obtain executable codes, it is only required to include

†Current affiliation: Fujitsu Ltd.
©1995 by the Computer Center of Kyushu University.
6-10-1,Hakozaki,Higashi,Fukuoka,812-81,Japan
Permission to copy without fee all or part of this publication is granted provided that the copies are not made or distributed for commercial advantage.
Otherwise, no part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including phototyping and recording, or by any information storage and retrieval system, without permission from publisher.

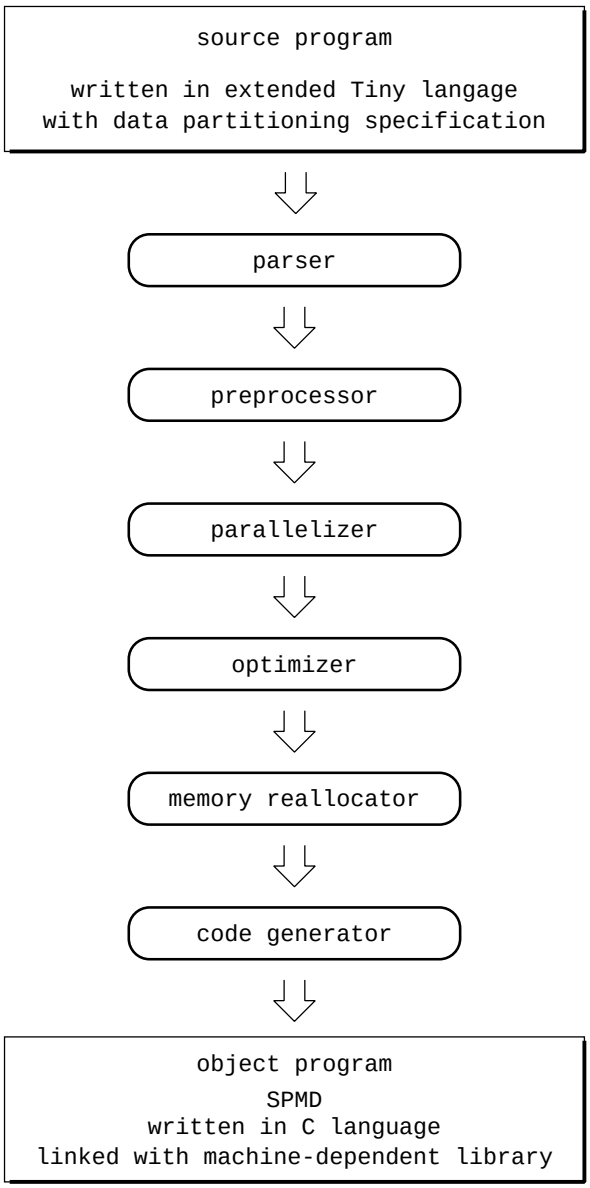

Figure 1: Components of TINPAR

machine-dependent header files and to link a communication library.

In order to handle messages, TINPAR defines machine independent interfaces for the communication library. TINPAR expects that the library has the capability of dynamic message vectorization to combine short messages into a long one.

2.3 Components of TINPAR

TINPAR consists of 6 components as shown in Figure 1.

The role of each component is as follows.

1. *Parser*
   This component parses a source program and generates intermediate codes with a declaration table and an expression tree.

2. *Preprocessor*
   This component performs the following transformations to get a program suitable for parallelization by the succeeding component.

   Idiom recognition
   Straightforward parallelization of reductions, such as summation of array elements, results in poor performance because it may be executed sequentially. However, by changing the sequence of program execution, one can improve the performance.

   Variable caching
   In general, the cost of accessing remote data is very high. Therefore, in the case of accessing the same remote data repeatedly, such as matrix multiplication, variable caching, which makes a local copy of the data, can improve the performance since it reduces communications.

3. *Parallelizer*
   This component parallelizes the source program straightforwardly following owner computes rule under given data distribution specifications. Its parallelizing strategy is shown below.

   - Before each assignment statement to distributed array, an `IF` statement is inserted to determine whether the assignment should be executed on each processor.
   - Reference to distributed array is converted to a `RECV` statement to receive the value from the owner processor. On the other hand, the owner processor have to send the value. Therefore, a `SEND` statement to send the value to the requesting processor and an `IF` statement to determine the owner processor are also inserted.
   - Indices of distributed array are not modified in this phase because the parallel codes are generated assuming that memory space for the whole of distributed arrays are reserved on each processor. However, unnecessary space is removed by the memory reallocator.

4. *Optimizer*
   This component optimizes the object codes parallelized by the preceding component for the following objectives.

   Reducing run-time overhead
   Straightforward parallelization by the parallelizing component explosively increases

```
real a(0:n - 1:*)
real b(0:n - 1:1)
/* Matrix a() is BLOCK distributed and matrix b() is CYCLIC distributed */

for i = 0, n - 1 do
  b(i) = a(i)
endfor
                        ⇓
real a(0:n - 1:*)
real b(0:n - 1:1)

for i = 0, n - 1 do
  if OWNER(a(i)) == GET_CELL_ID() then
    SEND(OWNER(b(i)), a(i))
  endif
  if OWNER(b(i)) == GET_CELL_ID() then
    RECV(OWNER(a(i)), tmp_a)
    b(i) = tmp_a
  endif
endfor
```

Figure 2: Parallelization

run-time overhead, such as the determination of owner processor. Through compile-time analysis, this component reduces the overhead.

Hiding communication latency
Communication latency, which is the time to wait for a message to be delivered, often brings poor performance. It is very important to reduce or hide the latency by rescheduling message-passing codes.

Details of them are described in section 4.

5. *Memory Reallocator*
   This component arranges memory allocations for distributed array. This reallocation involves modification of the indices of the distributed data. Modifying the indices in this phase makes it easy to perform the previous optimizations.

6. *Code Generator*
   This component translates intermediate codes into object codes in C language.

3 Communication Library

Communication codes in the object codes are implemented as the run-time library. In this section, we present the characteristics of this library: dynamic message vectorization and its use.

3.1 Dynamic Message Vectorization

The startup cost for communication is large for the message passing multiprocessors. It is important to reduce the number of the communication invocations. To achieve this, message vectorization has been proposed [4]. It combines several messages into one vectorized long message to reduce the number of the communication invocations.

```
real a(0:n - 1:*)

for i = 0, n - 1 do
  if OWNER(a(i)) == GET_CELL_ID() then
    a(i) = ...
  endif
endfor
                        ⇓
real a(0:⌈n / GET_N_CELL()⌉ - 1)

for i = 0, n - 1 do
  if OWNER(a(i)) == GET_CELL_ID() then
    a(LOCAL_INDEX(a(i))) = ...
  endif
endfor
```

Figure 3: Memory Reallocation

Therefore, TINPAR is provided with the run-time system for dynamic message vectorization. The run-time system takes care of the vectorization which includes 1)allocation and management of temporary data area and 2)sending and receiving of the packed data. Thus, in the object code of TINPAR, one SEND/RECEIVE statement corresponds to packing/unpacking of one datum.

3.2 Implementation of Efficient Communication

Communication based on our dynamic message vectorization is implemented as follows:

1. packing the messages

   When `SEND` statement is executed, The processor packs the messages.

2. sending the message

   The message is sent from the temporary area for the pacekd data on appropriate time.

3. receiving the message

   When the interruption on the arrival of the message occured, message is stored into the area managed by the run-time system.

4. unpacking the message

   The processor unpack the message when `RECEIVE` statement is executed.

4 Optimizations

The optimizing component of TINPAR leaves machine-dependent optimizations to the optimizer of

C compilers of a target machine. Consequently, the optimizations of TINPAR consist of reducing run-time overhead and hiding communication latency.

4.1 Reducing Run-time Overhead

Since the parallelizing strategy of TINPAR is so simple and straightforward, it may bring run-time overhead, such as `IF` statements to determine owner processors. The overhead can be reduced with the following techniques to eliminate unnecessary statements.

4.1.1 Narrowing Loop Bounds

Parallelization by TINPAR which follows owner computes rule often generates `FOR` loops having `IF` statements to determine whether to execute each statement in the original loop body. In general, one processor owns only a part of distributed arrays. Thus there are many iterations in which only `IF` statements are executed. Since this results in poor performance obviously, elimination of such useless iterations is beneficial.

First in this optimization, the component solves conditionals of each `IF` statement in a `FOR` loop to figure out the loop variable range in which the body of the `IF` statement will be executed. Then the union of the all range is computed to find the optimal lower and upper bounds of the loop variable, as shown in Figure 4. By this optimization, useless iterations are eliminated at compile time.

```
for i = 0, n - 1 do
  if OWNER(a(i)) == GET_CELL_ID() then
    a(i) = ...
  endif
endfor
                          ⇓
for i ∈ {i|OWNER(a(i)) == GET_CELL_ID(), \
          0 ≤ i ≤ n - 1} do
  if OWNER(a(i)) == GET_CELL_ID() then
    a(i) = ...
  endif
endfor
```

Figure 4: Narrowing Loop Bounds

4.1.2 Elimination of Unnecessary `IF` Statements

As a result of narrowing loop bounds, some `IF` statements may become unnecessary. Elimination of such statements also improves the performance.

In order to decide whether an `IF` statement is unnecessary or not, this component also computes the loop variable range in which its condition is satisfied. If the computed range includes the whole of the possible loop variable range of surrounding loops, the condition is always true and run-time decision is not necessary. So the `IF` statement can be eliminated as shown in Figure 5. Similarly, if the computed range excludes the possible range, the conditional is always false. By this optimization, not only conditional branch but also calculation of owner processors can be eliminated.

```
for i ∈ {i|OWNER(a(i)) == GET_CELL_ID(), \
          0 ≤ i ≤ n - 1} do
  if OWNER(a(i)) == GET_CELL_ID() then
    a(i) = ...
  endif
endfor
                          ⇓
for i ∈ {i|OWNER(a(i)) == GET_CELL_ID(), \
          0 ≤ i ≤ n - 1} do
  a(i) = ...
endfor
```

Figure 5: Elimination of Unnecessary `IF` Statements

4.1.3 Loop Splitting for Elimination of `IF` Statements

Though the previous optimization can eliminate most of unnecessary `IF` statements, the remainder also has a possibility to be eliminated.

```
for i = 0, n - 1 do
  if OWNER(a(i)) == GET_CELL_ID() then
    a(i) = ...
  endif
  b(i) = ...
endfor
                          ⇓
for i ∈ {i|OWNER(a(i)) == GET_CELL_ID(), \
          0 ≤ i ≤ n - 1} do
  a(i) = ...
  b(i) = ...
endfor
for i ∈ {i|OWNER(a(i)) != GET_CELL_ID(), \
          0 ≤ i ≤ n - 1} do
  b(i) = ...
endfor
```

Figure 6: Loop Splitting for Elimination of `IF` Statements

```
for i = 0, GET_N_CELL() do
  if i != GET_CELL_ID() then
    SEND(i, a(t))
  endif
endfor
                          ⇓
BROADCAST(a(t))
```

Figure 7: Replacing `SEND` with `BROADCAST`

For example, the previous optimization can not

```
for i ∈ {i|OWNER(a(i)) == GET_CELL_ID(), \
          0 ≤ i ≤ n - 1} do
  BROADCAST(a(i))
  tmp_a = a(i)
  if tmp_a < 0 then
    a(i) = - tmp_a
  endif
endfor
for i ∈ {i|OWNER(a(i)) != GET_CELL_ID(), \
          0 ≤ i ≤ n - 1} do
  RECV(OWNER(a(i)), tmp_a)
endfor
                        ⇓
for i ∈ {i|OWNER(a(i)) == GET_CELL_ID(), \
          0 ≤ i ≤ n - 1} do
  tmp_a = a(i)
  if tmp_a < 0 then
    a(i) = - tmp_a
  endif
endfor
```

Figure 8: Removal of Unnecessary SEND/RECV Statements

```
for i = 0, n - 1 do
  a(i) = ...
  if OWNER(b(i)) == GET_CELL_ID() then
    SEND(OWNER(c(i)), b(i))
  endif
  if OWNER(c(i)) == GET_CELL_ID() then
    RECV(OWNER(b(i)), tmp_b)
    c(i) = tmp_b
  endif
endfor
                        ⇓
for i = 0, n - 1 do
  if OWNER(b(i)) == GET_CELL_ID() then
    SEND(OWNER(c(i)), b(i))
  endif
  a(i) = ...
  if OWNER(c(i)) == GET_CELL_ID() then
    RECV(OWNER(b(i)), tmp_b)
    c(i) = tmp_b
  endif
endfor
```

Figure 9: Re-scheduling SEND Statements

eliminate IF statements in the loop shown in Figure 6. However, it makes the previous optimization more applicable to split the loop into two loops in which each conditional of IF statements is either always true or false. By this optimization, together with the previous one, almost all unnecessary IF statements generated in parallelizing step will be eliminated.

4.1.4 Replacing SEND with BROADCAST Statement

It is efficient to utilize the system supported BROADCAST communication library instead of sending the same message to all processors repeatedly. For example in Figure 7, a(t) is send to all processors. Thus SEND statements can be replaced with BROADCAST statement.

4.1.5 Removing Unnecessary SEND/RECV Statements

On message-passing multiprocessors, cost for communication is large, thus unnecessary communication must be eliminated. For example in the upper half of Figure 8 , unless the variable tmp_a is referred after the first loop, the BROADCAST/RECV statements are unnecessary.

In this optimization phase, the data flow analysis is applied so as to eliminate unnecessary communication codes. Such unnecessary RECV statements detected by the data flow analysis are eliminated together with corresponding SEND or BROADCAST statements.

4.2 Hiding Communication Latency

4.2.1 Re-scheduling SEND Statements

On message-passing multiprocessors, messages are buffered by receiver processors until they are needed. This buffering makes it possible that sender processors send required data earlier to hide or reduce the latency.

While the parallelizing component generates a pair of SEND and RECV statements to be neighboring, the optimizing component moves SEND statements in each loop body to the top of the body as shown in Figure 9. Although performing this optimization alone does not give enough effect, it will improve the performance significantly if it is combined with the following optimizations.

4.2.2 Distribution of Loops Having SEND/RECV Statements

If a loop body which has both SEND and RECV statements is small, the re-scheduling described above is not sufficient to hide the whole of the latency. In such cases, it is effective to divide a loop into two portions such that one of the oprtions has only SEND statements as shown in Figure 10. This loop distribution makes the loop containing nothing other than SEND statements and the loop is executed before the other loop having RECV and other statements is executed. Therefore, required data is sent early enough to hide the latency.

```
for i = 0, n - 1 do
  if OWNER(a(i)) == GET_CELL_ID() then
    SEND(OWNER(b(i)), a(i))
  endif
  if OWNER(b(i)) == GET_CELL_ID() then
    RECV(OWNER(a(i)), tmp_a)
    b(i) = tmp_a
  endif
endfor
```

⇓

```
for i = 0, n - 1 do
  if OWNER(a(i)) == GET_CELL_ID() then
    SEND(OWNER(b(i)), a(i))
  endif
endfor
for i = 0, n - 1 do
  if OWNER(b(i)) == GET_CELL_ID() then
    RECV(OWNER(a(i)), tmp_a)
    b(i) = tmp_a
  endif
endfor
```

Figure 10: Distribution of Loops Having `SEND/RECV` Statements

```
for i = 0, n - 1 do
  if OWNER(a(i)) == GET_CELL_ID() then
    SEND(OWNER(b(i)), a(i))
  endif
endfor
for i = 0, n - 1 do
  if OWNER(b(i)) == GET_CELL_ID() then
    RECV(OWNER(a(i)), tmp_a)
    b(i) = tmp_a
  endif
endfor
```

⇓

```
for i = 0, n - 1 do
  if OWNER(a(i)) == GET_CELL_ID() then
    SEND(OWNER(b(i)), a(i))
  endif
endfor
MSG_BUFFER_FLUSH_ALL()
for i = 0, n - 1 do
  if OWNER(b(i)) == GET_CELL_ID() then
    RECV(OWNER(a(i)), tmp_a)
    b(i) = tmp_a
  endif
endfor
```

Figure 11: Insertion of `MSG_BUFFER_FLUSH` Statements

4.2.3 Insertion of `MSG_BUFFER_FLUSH` Statements

As described before, TINPAR defines the machine-independent interfaces for the communication library and expects it to have a capability of dynamic message vectorization. Dynamic message vectorization means that the run-time communication system buffers short messages and combines them into a long message to hide startup cost of message passing.

In this optimization, `MSG_BUFFER_FLUSH` statements to flush the buffers are inserted after the series of `SEND` statements as shown in Figure 11. This insertion makes it possible to buffer short messages to vectorize properly.

4.3 The Sequence of Applying Optimizations

There are several constraints to determine the sequence of applying optimizations as follows:

- Loop distribution should be preceded by narrowing of loop bounds, since the narrowing becomes more applicable if there are less number of statements in a loop body.
- Elimination of unnecessary `IF` statements should be preceded by narrowing of loop bounds and loop splitting.
- Since elimination of unnecessary `SEND/RECV` statements requrires data flow analysis, it is more applicable where unnecessary `IF` statements are eliminated so as to simplify control flow.
- Re-scheduling of `SEND` statements is more applicable if the constraints by control and data dependency are relaxed by loop distribution and elimination of `IF` statements before the re-scheduling.
- Distributing loops having `SEND/RECV` statements is more applicable after `SEND` statements are re-scheduled.
- Inserting `MSG_BUFFER_FLUSH` statements and replacing of `SEND` with `BROADCAST` should be performed after all transformations.

Therefore, the optimizations are performed as the following order.

1. Re-scheduling `SEND` statements,
2. distribution of loops having `SEND/RECV` statements,
3. narrowing loop bounds,
4. eliminating unnecessary `IF` statements,
5. loop splitting,
6. eliminating of unnecessary `IF` statements,
7. eliminating unnecessary `SEND/RECV` statements,
8. re-scheduling `SEND` statements,
9. insertion of `MSG_BUFFER_FLUSH` statements and
10. replacing `SEND` with `BROADCAST` statements.

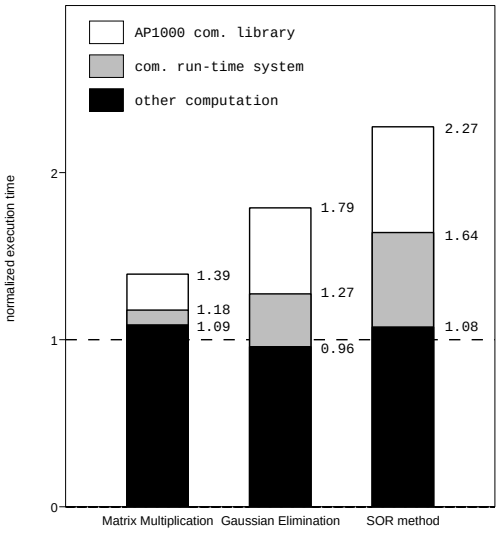

Figure 12: The Ratio of Execution Time

```
const n = 512
real a(0:n - 1:*, 0:n - 1)
real b(0:n - 1:*, 0:n - 1)
real c(0:n - 1:*, 0:n - 1)
real local_b(0:n - 1, 0:n - 1)
/*  All matrices are BLOCK distributed in the first dimension. */

for i = 0, n - 1 do
  for j = 0, n - 1 do
    local_b(i, j) = b(i, j)
  endfor
endfor
for i = 0, n - 1 do
  for j = 0, n - 1 do
    c(i, j) = 0.0
    for k = 0, n - 1 do
      c(i, j) = c(i, j) + a(i, k) * local_b(k, j)
    endfor
  endfor
endfor
```

Figure 13: Matrix Multiplication

5 Performance Evaluation

In this section, we demonstrate parallelization ability of the current version of TINPAR. We present performance evaluation results and discuss overhead of message handling and the effectiveness of each optimizations using several basic application programs.

The following measurements are performed on AP1000[3]. Application programs are parallelized by TINPAR automatically, compiled with SPAR-Compiler C 2.0.1 with `-O4` compilation flag, linked with `lib.fast` cell library, and executed with `-R 0 -LSEND` run-time options.

In the following discussion, the speedup ratio for each program shown in Figure 14, 17 and 20 is calculated based on the execution time of the sequential program which is generated from the same source program for the execution on a single processor of AP1000.

In order to evaluate the overhead of message handling, we also measured the execution time of three programs on AP1000 of 64 cells and analyzed it into the times spent in 1)the AP1000 communication library, 2)our communication run-time library and 3)computation other than message handling. Figure 12 shows the result of this analysis, where the execution time is normalized by 1/64 of the execution time of sequential program.

The effectiveness of each optimization is shown in Figure 15, 18 and 21. In these figures, applied optimization schems of $1-N$ indicates that the code optimization schemes 1 to N mentioned in section 4.3 are applied to the code. Each speedup ratio is calculated by (execution time of non-optimized code) / (execution time of optimized code) on AP1000 of 64 cells.

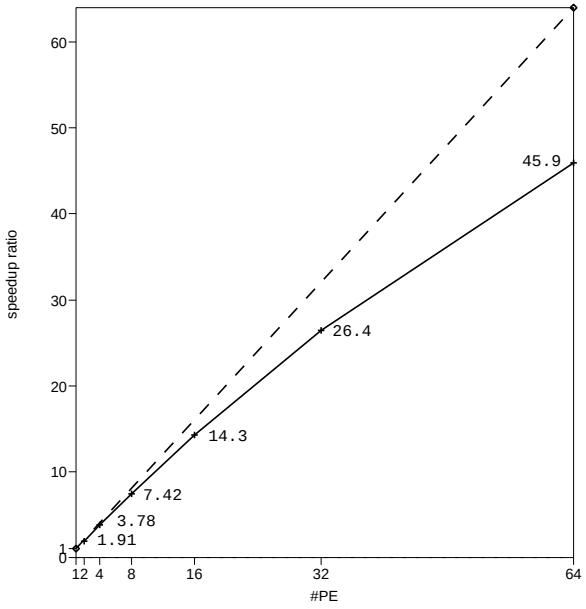

Figure 14: Speedup Ratio for Matrix Multiplication

5.1 Matrix Multiplication

For the first measurement, we used the source program for matrix multiplication shown in Figure 13. In this program, block distribution in the first dimension is specified for each matrix. The array `local_b()` is a temporal local copy of the remote data `b()` replicated on all the processors.

Figure 14 shows the speedup ratio to the number of processors. We achieve 45.9 times speedup with 64 processors.

Figure 15 shows the effectiveness of each optimization. Narrowing of loop bounds (optimization 3) is the most effective optimization to achieve good performance. Elimination of unnecessary `IF` statements

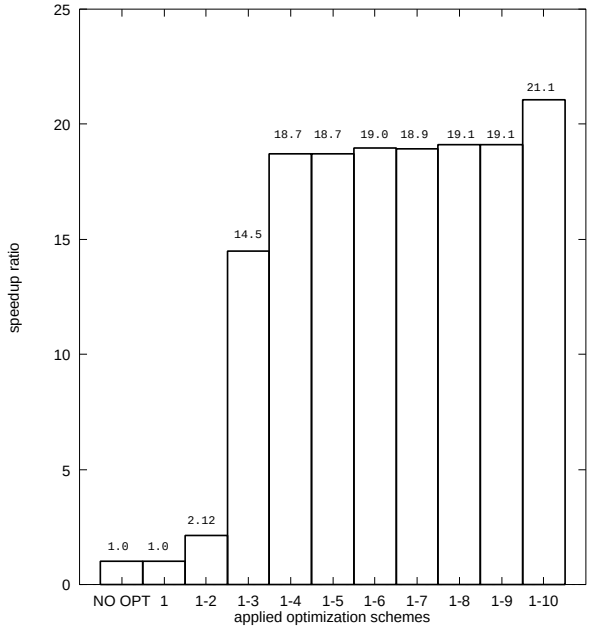

Figure 15: Optimization Effect for Matrix Multiplication

(4) is effective since the narrowing of loop bounds preceded it.

5.2 Gaussian Elimination

We measured the performance using the program for Gaussian elimination shown in Figure 16. For all the arrays, cyclic distribution in the first dimension is specified [1]. The arrays `local_idamax()` and `local_dmax()` are introduced so that the processor which owns a column to be decomposed performs pivoting only with local data.

The temporary copy of the decomposed column is stored into the array `local_a()` on each processor. The assignment to this array is scheduled so that data is sent first to the processor which owns the column to be decomposed on the next iteration of `k`.

Figure 17 shows the speedup ratio to the number of processors. We achieve 35.8 times speedup with 64 processors in comparison with sequential execution.

Figure 18 shows the effectiveness of each optimization. Insertion of `MSG_BUFFER_FLUSH` (optimization 9) is most effective to achieve good performance, since the data `a(k,i)`, assigned to `local_a()` as described above, is forced to be sent by `MSG_BUFFER_FLUSH`. Elimination of unnecessary `SEND/RECV` statements (7) is also necessary, since the broadcast messages, brought by the assignments to scalar variables `abs_a_k_i` and `dmax`, are eliminated.

5.3 SOR

[1] Actually, the array `a()` is distributed in the column orientation because it is transposed when stored.

```
const n = 1024
real a(0:n - 1:1, 0:n - 1)
integer ipvt(0:n - 1)
integer info
temporary integer \
  local_idamax(0:GET_N_CELL() - 1:1)
temporary real local_dmax(0:GET_N_CELL() - 1:1)
temporary real \
  local_a(0:GET_N_CELL() - 1:1, 0:n - 1)
temporary integer l
temporary real abs_a_k_i, dmax, pivot, t
/* All matrices are CYCLIC distributed in the first dimension. */

info = 0
for k = 0, n - 2 do
  local_idamax(OWNER(a(k, k))) = k
  local_dmax(OWNER(a(k, k))) = abs(a(k, k))
  for i = k + 1, n - 1 do
    abs_a_k_i = abs(a(k, i))
    dmax = local_dmax(OWNER(a(k, k)))
    if (abs_a_k_i > dmax) then
      local_idamax(OWNER(a(k, k))) = i
      local_dmax(OWNER(a(k, k))) = abs(a(k, i))
    endif
  endfor
  l = local_idamax(OWNER(a(k, k)))
  ipvt(k) = l
  pivot = a(k, l)
  if (pivot != 0) then
    if (l != k) then
      t = a(k, l)
      a(k, l) = a(k, k)
      a(k, k) = t
    endif
    t = -1.0 / a(k, k)
    for i = k + 1, n - 1 do
      a(k, i) = t * a(k, i)
    endfor
    for cid = OWNER(a(k + 1, i)), \
              GET_N_CELL() - 1 do
      for i = k + 1, n - 1 do
        local_a(cid, i) = a(k, i)
      endfor
    endfor
    for cid = 0, OWNER(a(k + 1, i)) - 1 do
      for i = k + 1, n - 1 do
        local_a(cid, i) = a(k, i)
      endfor
    endfor
    for j = k + 1, n - 1 do
      t = a(j, l)
      if (l != k) then
        a(j, l) = a(j, k)
        a(j, k) = t
      endif
      for i = k + 1, n - 1 do
        a(j, i) = a(j, i) \
          + t * local_a(OWNER(a(j, i)), i)
      endfor
    endfor
  else
    info = k + 1
  endif
endfor
```

Figure 16: Gaussian Elimination

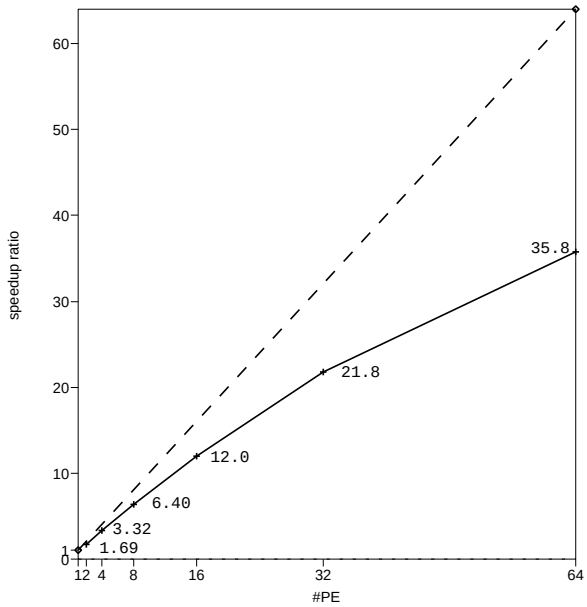

Figure 17: Speedup Ratio for Gaussian Elimination

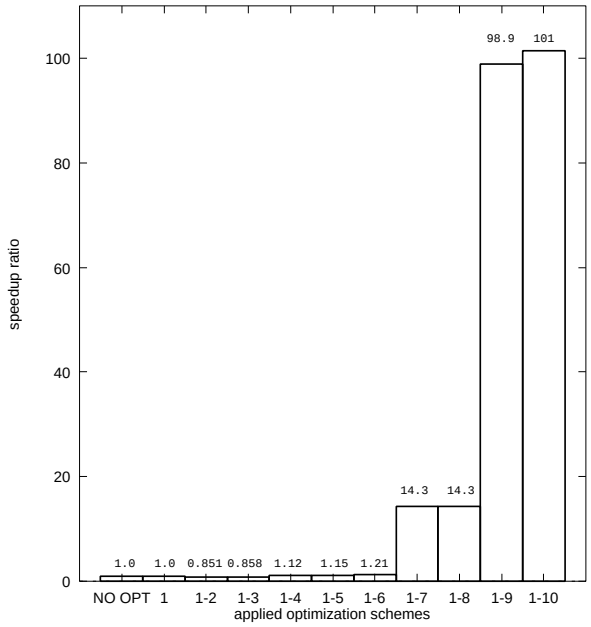

Figure 18: Optimization Effect for Gaussian Elimination

We measured the performance for a code of Successive over relaxation method shown in Figure 19. For the array `a()`, block distribution in the second dimension is specified. The `outer_j` loop is introduced by strip-mining the `j` loop so as to maintain good load balance.

Figure 20 shows the speedup ratio to the number of processors. We achieve 28.1 times speedup with 64 processors in comparison with sequential execution.

Figure 21 shows the effectiveness of each optimization. Narrowing of loop bounds (optimization

```
const n = 1024
const width = 6
real omega
real a(0:n - 1:*, 0:n - 1)
/*  Matrix a() is BLOCK distributed in the first dimension. */

for outer_j = 0, n - 1, width do
  for i = 1, n - 2 do
    for j = MAX(1, outer_j),
            MIN(n - 2, outer_j + width - 1) do
      a(i, j) = a(i, j) * (1 - omega) \
                + (a(i - 1, j) + a(i, j - 1) \
                   + a(i, j + 1) + a(i + 1, j)) \
                  * (omega / 4)
    endfor
  endfor
endfor
```

Figure 19: SOR

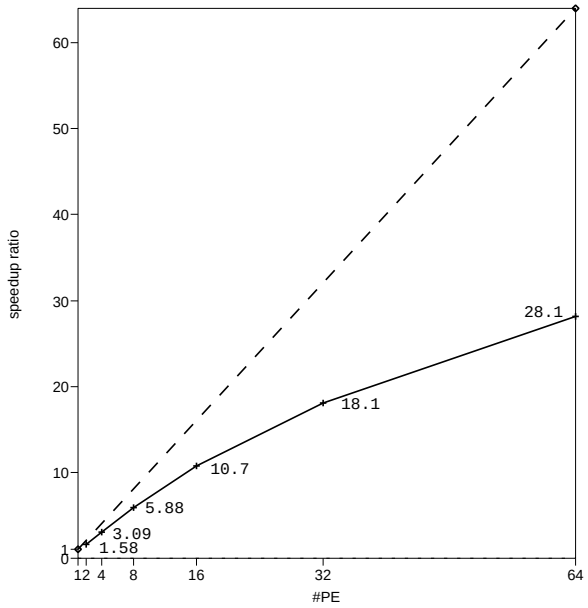

Figure 20: Speedup Ratio for SOR

3), elimination of unecessary `IF` statements (6) and Insertion of `MSG_BUFFER_FLUSH` (9) play important role in optimization. Especially, the strip-mined message described above is forced to be sent by `MSG_BUFFER_FLUSH`.

6 Related Work

Many researchers are developing parallelizing compilers for message-passing multiprocessors. Among their researches, Fortran D [4] takes same strategy for parallelizing using owner computes rule as we do. They use iteration/index set to partition computations and generate communication codes.

The most basic communication optimization performed by the Fortran D compiler is message vectorization. Analysis of inter-processor communication is performed mainly at compile-time. On the

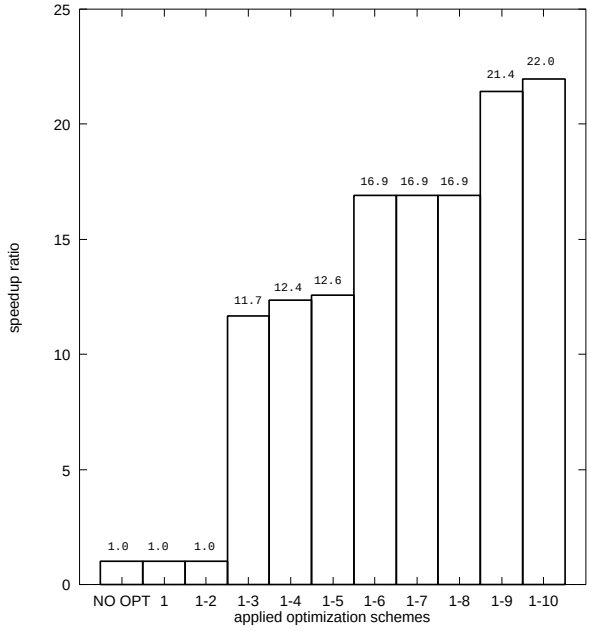

Figure 21: Optimization Effect for SOR

other hand, we adopt dynamic message vectorization so that communication is performed efficiently while remaining flexibility.

They also present several techniques for block-cyclic distribution[6]. These techniques include program analysis to narrow loop bounds, though there are some limitations to apply these techniques.

They use Gaussian elimination program as an example as we do [5]. In order to make pivot selection more efficient, their compiler system recognizes reduction and relaxes the owner computes rule. On the other hand, TINPAR parallelizes the Gaussian elimination program and applies general optimizations as described in this paper. This shows that our technique is applicable to wide range of codes and has large flexibility.

7 Conclusion

In order to make it easy to program for message-passing multiprocessors, we are developing a parallelizing compiler, TINPAR. Parallelization by TINPAR consists of three steps, restructuring, parallelizing, and optimizing steps. In this paper, we described the optimizations performed in the third step and presented that the implemented parallelizer and optimizer parallelized restructured programs with reasonable performance.

In the future work, we will improve the optimizing capability, evaluate the performance with other programs, and design the preprocessing component which performs the restructuring and assists parallelization automatically or semi-automatically.

Acknowledgments

We would like to thank Fujitsu Laboratory Ltd. for offering their AP1000. We also wish to thank all members in Tomita Laboratory of Kyoto University for their contribution through daily discussion. This work was supported in part by the Grant-in-Aid for Developmental Scientific Research #06205001 from the Ministry of Education, Science, Sports and Culture.

References

[1] I. Miyoshi, K. Maeyama, S. Goto, S. Mori, H. Nakashima, and S. Tomita: A Parallelizing Compiler for Message-Passing Multiprocessors, Proc. of JSPP'95, pp.51-58, May 1995 (In Japanese).

[2] M. Wolfe: Tiny: A Loop Restructuring Research Tool, Oregon Graduate Institute of Science and Technology, 1992.

[3] H. Ishihata, T. Horie, S. Inano, T. Shimizu, S. Kato, and M. Ikesaka: Third Generation Message Passing Computer AP1000, Int'l Symp. on Supercomputing, pp.46-55, 1991.

[4] S. Hiranandani, K. Kennedy, and C. Tseng: Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines, Proc. of Supercomputing '91, pp.86-100, 1991.

[5] S. Hiranandani, K. Kennedy, and C. Tseng: Preliminary Experiences with the Fortran D Compiler, Proc. of Supercomputing '93, pp.338-350, 1993.

[6] S. Hiranandani, K. Kennedy, J. Mellor-Crummey, and A. Sethi: Compilation Techniques for Block-Cyclic Distributions, Proc. of 1994 Int'l conf. on Supercomputing, pp.392-403,1994.