# Optimized Code Generation for Heterogeneous Computing Environment using Parallelizing Compiler TINPAR


Shin-ya GOTO[1] Atsushi KUBOTA[2] Toshihiko TANAKA[3] Masahiro GOSHIMA[3]
Shin-ichiro MORI[3] Hiroshi NAKASHIMA[4] Shinji TOMITA[3]

1 Sumitomo Electric Industries,Ltd.
1-1-3, Shinmaya, Konohana-ku
Osaka, 554-0024 JAPAN

2 Dep. of Computer Engneering
Hiroshima City Univ.
Hiroshima, 731-3194 JAPAN

3 Graduate School of Informatics
Kyoto University
Kyoto, 606-8185 JAPAN

4 Dep. of Info. and Compt. Sciences
Toyohashi Univ. of Technology
Toyohashi, 441-8580 JAPAN

{sigoto,kubota,t-tanaka,goshima,moris,nakasima,tomita}@kuis.kyoto-u.ac.jp



## Abstract

*This paper presents a compiling technique to generate optimized codes for heterogeneous computing environment. This paper also proposes a new dynamic load redistribution mechanism which can adaptively and dynamically distribute tasks among computers according to their available computing power which may vary during the computation.*

*As the results of the performance evaluation, we could confirm that the generated codes are effectively executed in a heterogeneous computing environment with dynamic load change.*


## 1. Introduction

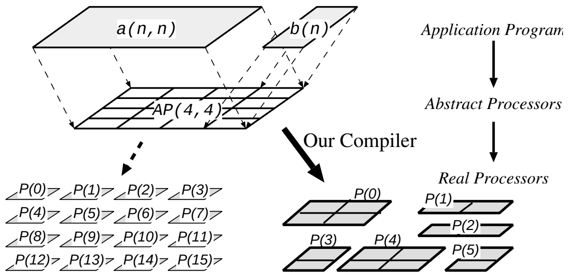
We have been developing a parallelizing compiler TINPAR[10, 7]. TINPAR parallelizes sequential programs written in extended form of Fortran like Tiny language[14], which allows the users to write program assuming a global address space. It automatically partitions computation and inserts communication codes following the owner computes rule. In order to generate high quality codes, we have developed several optimization techniques for TINPAR to reduce communication overhead and to hide communication latency.

So far, we have been focusing on homogeneous computing environments, particularly on the message passing multicomputer like AP1000, as a target of TINPAR. However, we feel strong demands of parallelizing compilers for the heterogeneous computing environment where several kinds of computers collaborate either tightly or loosely with each other on running multiple users' programs. Such compilers are required to have the ability to adaptively and dynamically distribute tasks among computers according to their available computing power which may vary during the computation due to the other users' program executions.

In this paper, we focus on heterogeneous computing environment, *HCE* for short, propose a technique to generate optimized code for the *HCE* and describe its implementation on our parallelizing compiler TINPAR. Similar works[2, 12] have been done for the environment where the number of available processors varies with time, but they assume an homogeneous computing environment.

In our study, we assume a HCE where several computers with different performance are connected to a network and each computer's load is dynamically changed during each computation. To utilize optimization techniques developed for TINPAR, we also assume that the output of the compiler is SPMD code and the compiler conforms to *the owner computes rule*.

In Section 2, we show our code generation strategies for a HCE. Then, in Section 3, we present its implementation in detail. A result of our performance evaluation is shown in Section 4 and the paper is concluded in Section 5.



(a) Homogeneous Environment (b) Heterogeneous Environment

Figure 1. Heterogeneous Data Distribution.

## 2. Code Generation Strategies

### 2.1. Abstract Processor

In our compiler, we have introduced a concept of **abstract processor** (Fig.1). The abstract processor(AP) is an ideal processor which can provide unique processing power at any time. Currently, our definition of abstract processor is the same as that in HPF[4].

As the first step of the compilation, our compiler equally assigns data space, or array template in HPF, to APs. Thus, an equal amount of task is assigned to each AP at the same time.

The next step of compilation is to decide an assignment, or mapping, of APs to real processors(RPs) such that the number of APs assigned to each RP is in proportion to its available computing power (Fig. 1(b)).

For the flexibility of AP assignment, in our compiler, no restriction is put on the mapping of APs to RPs, so that any AP can be assigned to any RP. Therefore, we can easily achieve the precise and dynamic control of load distribution even if APs with multi-dimensional index are given. Furthermore, since the minimal unit of parallel execution is determined by the granularity of the AP, it is easy to control the granularity by adjusting the total number of abstract processors.

### 2.2. Abstract Processor Assignment Strategy

There exists two fundamental models of execution at the RP when two or more APs are assigned to it. They are **the multi-thread model**[8] and **the single-thread model**.

In our compiler, we adopted the **single-thread model** because the overhead of handling irregular data region is much smaller than the overhead of thread scheduling in multi-thread model under currently available operating systems.

In the single-thread model, APs assigned to an RP are combined into a single thread and are executed indivisibly. Accordingly, communications between APs on the same RP are replaced to simple assignment statements. On the other hand, in order to avoide any deadlock, a thoughtful pre-scheduling of APs is required for each thread, instead of the run-time thread scheduling in the multi-thread model.

The compiler for this model should consider the issues of this pre-scheduling and of the handling of irregular data region associated with the combined APs.

The pre-scheduling determines, based on the owner computes rule, the iterations to be executed at each RP to which irregular data area is assigned. Thus, every time an AP-to-RP mapping is changed, this pre-scheduling is performed once as follows.

For each RP, determine a set of iterations to be executed, and make a list of iteration numbers, or an *iteration list*. During the executions of a loop, the RP just refers to its iteration list to know the next iteration to be executed. By introducing the iteration list, we could alleviate the run-time overhead due to the heterogeneous task assignment.

This technique can be regarded as an extension of the Inspector/Executor algorithm[9, 11, 6] which has been proposed as a parallelization technique for irregular loops.

### 2.3. Dynamic Load Distribution

In order to realize a flexible dynamic load distribution mechanism we have developed a new self-scheduling scheme which suits well to the data owner computes rule. In our scheme, all RPs executing together for a given application program negotiate one another for the load balancing at checkpoints.

We define checkpoint, load information, and negotiation scheme as follows.

**Checkpoint** : A real processor, *RP*, is said to arrive at the checkpoint when *RP* completes the works for every APs mapped to it, where the work corresponds to a certain number of iterations in the outmost loop of the task assigned to the AP. We refer to the duration between one checkpoint and the next as *period*.

**Load Information** : The load information of a real processor *RP* for an execution period consists of the number of APs for which *RP* is responsible during this period and the absolute elapsed time in this period.

**Cooperative Scheme** : When an RP arrives at a checkpoint, using its load information, it estimates its own computing power available during the next period. Then, APs are reassigned to RPs so that the number of APs assigned to each RP is in proportion to its estimated available computing power[3].

RPs arriving at a check point attend a barrier synchronization. When all RPs arrive this point, they exit from

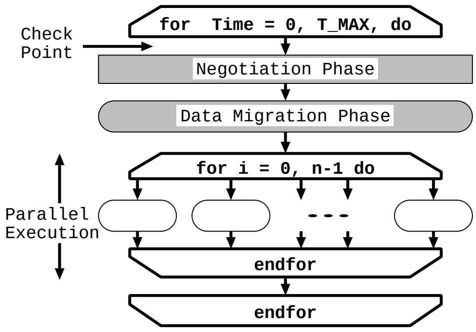



Figure 2. Execution Phase Flow.

this synchronization and enter dynamic load distribution phase(Fig.2). In our compiler, dynamic load distribution phase is composed of the following two phases: *negotiation phase* and *data migration phase*.

#### 2.3.1 Negotiation Phase

In this phase, at first, each RP estimates its computing power in the near future. Then RPs negotiate one another to determine the data distribution ratio according to their estimations. In determining the data distribution ratio, we consider the following issues.

**Time Constance** In our compiler, we place a restriction on the amount of data to be migrated at a checkpoint. It is because, if a large amount of data is migrated from one RP to another, overall load estimation becomes less accurate and thus it may cause unexpected load imbalance and may cause ping-pong effect in the worst case. This restriction determines a sort of "time constant" for our dynamic load distribution mechanism. It may increase the stability of this mechanism though it may make the convergence time longer.

**Threshold** If the difference of any two processors' estimated computing power is not so large, or, if there exists large communication latency between any two processors, it may be better not to perform data migration between them. So we place another restriction on the difference in estimated computing power of any two processors, we call it threshold, to decide whether it is better to perform data migration.

#### 2.3.2 Data Migration Phase

In this phase, data migration is performed among RPs according to the data distribution ratio decided at negotiation phase. Since the owner computes rule is assumed, APs are also migrated following their associated data migrations. After this migration, most successive APs are assigned to the same RP in our current implementation.

When data migration is completed everywhere, each RP starts updating its pre-scheduling information including the iteration list and the table which is used to decide data owner.

## 3. Code Generation for the HCE

### 3.1. Conversion of Data Owner Decision Code

In the intermediate code whose target is the APs, the owner of a data is computed with one division(, or modulo) and one add operations. Hereafter, we express this owner decision function as `OWNER(`*data*`)`. This decision is not so simple in the HCE however.

In our implementation, we provide each real processor with an index table `AP_RP_Map[]` which indicates the relation of the AP number to the RP number, and let the real processor refer to this table to determine the owner of a data.

For example, if the index table

```
AP_RP_Map[8]={0,2,1,2,0,3,3,3}
```

is given, it means that $AP_0$ and $AP_4$ are assigned to $RP_0$, $AP_2$ is assigned to $RP_1$, $AP_1$ and $AP_3$ are assigned to $RP_2$ and so on, where $AP_i$ and $RP_i$ stands for abstract processor *i* and real processor *i*, respectively.

Using this table, the condition of the following if statement to decide whether the AP with id `AP_ID` owns the data `a(i)`:

```
if( OWNER(a(i)) == AP_ID )
                    then  ...
```

is converted as follows with real processor id (`RP_ID`).

```
if( AP_RP_Map[OWNER(a(i))]==RP_ID )
                      then ...
```

Similarly, the calculation of destination AP number in the following statement

```
Send( OWNER(a(i)), a(i+1) )
```

is converted as follows to send this message to the appropriate RP.

```
Send(AP_RP_Map[OWNER(a(i))], a(i+1))
```

### 3.2. Insertion of the Loop Scheduling Code

In the HCE, it is not so easy to determine which iterations should be executed on each RP because 1) the number of APs assigned to each RPs is different, 2) each real processor is responsible for an irregular data area, and furthermore 3) these assignments are dynamically changed during the execution.

In the code generated by our compiler, the decision of which iterations should be executed on an RP is made at *the loop scheduling part* before entering *the loop execution part* in order to reduce *if* statements for owner checking inside the original loop.

### 3.2.1 Code Generation Strategy

With irregularly distributed data, the following information is required for each RP to perform parallel execution of loops which refers this data.

1. Iteration numbers that should be executed on the RP.
2. For each reference in the loop, information of whether the RP has a responsibility for the communication corresponding to the reference.

In the loop scheduling part, each RP examines the owner of each data by using its index table `AP_RP_Map[]` and generates *the iteration list*. In the loop execution part, by referring its iteration list, the RP exactly executes required iterations.

Fig. 3 shows how the sequential code is parallelized. In this figure, *ExecList* stands for the iteration list, and *ExecElem* stands for an element in the list. Note that *ExecElem* is a structured data whose members are iteration number(*iteration*) and an identifier(*condition*) which represents all the conditions of communications in the corresponding iteration.

In this example, each RP examines, for each iteration, if it owns `a(i)` and/or `b(i+1)`, respectively. If it owns at least one of them, it appends corresponding information of iteration number and communication conditions, $I_a, I_b$, to its *ExecList*. Since only the index table `AP_RP_Map[]` is required in this examination and it is already existing in every RP, *ExecList* can be made without any communication.

In the loop execution part, each RP executes only the iterations listed in its own *ExecList*, thus it does not suffer from the overhead of executing unnecessary iterations. In addition, since the data reference pattern in each of these iterations has been classified at the loop scheduling part, RPs can immediately select the appropriate code from several alternatives during the execution. Such a code can be implemented with a `switch` statement which evaluates the *condition* corresponding to each iteration, as shown in Fig. 3(c).

Substitutions, in the intermediate code, accompanied with communications between APs are replaced to simple substitute statements in the output code when these processors are located on the same RPs. The case $I_a \cap I_b$ in Fig.3(c) shows this situation.

If a given loop is executed repeatedly, the loop scheduling for this loop is performed once every time when the abstract-to-real processor mapping is changed. Therefore, the overhead of the loop scheduling can be negligible if the frequency of the data migration is not so high.

```
    real a(N),b(N)
    processors AP(NAP)
    distribute a(BLOCK) onto AP
    distribute b(BLOCK) onto AP
    for i = 0, N - 2 do
      a(i) = b(i+1)
    endfor
```

(a)A Sequential Code

```
makeExecList(){
  ExecList = φ;
  for i=0, n-2 do
    ExecElem.condition = φ
    if( AP_RP_Map[ OWNER(a(i))]= RP_ID ) then
                    /* Own a(i) */
       Append I_a to ExecElem.condition.
    endif
    if( AP_RP_Map[ OWNER(b(i+1))]= RP_ID ) then
                    /* Own b(i+1) */
      Append I_b to ExecElem.condition.
    endif
    if( ExecElem.condition ≠ φ ) then
                    /* Own a(i) or b(i+1) */
      ExecElem.iteration = i
      Append ExecElem to ExecList.
    endif
  endfor
  return( ExecList )
}
```

(b) A Loop Scheduling Code

```
/* Start Execution */
ExecList = makeExecList()
/* Loop Execution Part */
for ExecElem in ExecList do
  i = ExecElem.iteration
  switch ExecElem.condition
    case I_a ∩ I_b:
     /*Own both a(i) and b(i+1)*/
      a(i) = b(i+1)
    case I_a:               /* Own a(i) */
      Recv( AP_RP_Map[OWNER(b(i+1))], tmp)
      a(i) = tmp
    case I_b:               /* Own b(i+1) */
      Send( AP_RP_Map[OWNER( a(i))], b(i+1) )
  endswitch
endfor
```

(c) A Loop Execution Code

Figure 3. Parallelizing a Loop for the HCE.

### 3.2.2 Code Refinement

**(1) Refinement at the Loop Scheduling Part**

In the code shown in Fig.3, all iterations are examined one by one to make *ExecList*. However, some of these iterations may be eliminated from being examined by utilizing the regularity inside the AP.

When data is distributed among APs in the block distribution fashion, it is possible to make *ExecList* by simple calculations because each APs holds a continuous region of data. For example, if an abstract processor $AP_p$ is assigned to an RP, the range $i$ for which the RP can access to the data $X(ai+b)$ without any communication is calculated as follows[10].

$$\frac{(\mathtt{N/NAP})\times AP_p - b}{a} \leq i < \frac{(\mathtt{N/NAP})\times(AP_p+1)-b}{a}$$

Note that `NAP` stands for the AP count. By calculating the range as well for each AP residing in the RP and then synthesizing these ranges, *Execlist* is generated for the RP. Unlike the case shown in Fig.3(b), there is no overhead due to the examination of all iterations.

Similarly, when data is distributed among NAP abstract processors in the cyclic distribution fashion, only the first NAP iterations of the loop corresponding to the dimension along which data is distributed are examined for the loop scheduling and, from the result of this examination, *Execlist* corresponding to the rest of the iterations is deduced based on the periodicity of data assignment.

**(2)Refinement at the Loop Execution Part**

The code shown in Fig.3(c) selects the appropriate data reference pattern for each iteration by evaluating its *condition* in the `switch` statement.

Assuming again that data is distributed among APs in the block distribution fashion, the RP is responsible for some continuous data regions and thus it may select the same reference pattern for a sequence of consecutive iterations.

Therefore, if such a sequence of iterations, with the same condition *condition*, starting at *start* and ending at *end* are grouped into one element of *ExecList*, the selection of the reference pattern for the iteration sequence is performed only once and the selected reference pattern is executed repeatedly for the sequence without any further selection.

## 3.3. Dynamic Load Distribution

As we have argued in Sec.2.3, RPs measure their average computing power during the current execution period and estimate their available computing power for the next period. According to their estimation, data redistribution is done at every checkpoints.

Checkpoints are specified by giving the remap option to any loop as follows.

```
for t = 0, 999 remap 20
```

In this specification, dynamic load balancing is performed every 20 iterations of $t$. In the current implementation, users are allowed to specify checkpoints only to the outermost loop.

Fig.4 shows the procedure of the dynamic load distribution. Assuming that each AP always has the same computational load, the load distribution is carried out so that average execution times for one AP become equal to each other among RPs.

At the checkpoint, *Acomp*$_k(t)$, which is the average execution time for one AP on real processor $k$, is calculated for every $k$ from *Rcomp*$_k(t)$, which is the total execution time of $AP_k(t-1)$ abstract processors during the $(t-1)$th period, excluding the time spent for communication and synchronization, and then the number of APs at real processor $k$ during $t$th period is determined for every $k$ so that estimated execution time on real processor $k$ during $t$th period, *Restim*$_k(t)$ = *Acomp*$_k(t) \times AP_k(t)$, becomes the same for every $k$.

Once $AP_k(t)$ is determined for every $k$, APs are remapped to RPs and all RP start calculating their own new index `AP_RP_Map[]` and *ExecList*. After that they resume their execution of loops.

# 4. Performance Evaluation

We have already implemented the parallelizing compiler using TINPAR and MPICH[1] which provides MPI[5] communication libraries. We then evaluated the quality of the generated code by experiments under both the homogeneous computing environment and the HCE.

These experiments were done using four workstations, including 60MHz SPARC Station 20(SS20 for short) and/or 167MHz SPARC Ultra-1(ULTRA for short), interconnected by a 10Mbps ethernet switch fabric. The organization of four workstations was one of the following combinations: 1)four SS20s, 2)four ULTRAs and 3)two SS20s and two ULTRAs. We assumed there was no other user processes in any workstations during the experiments, except for the experiment to evaluate external load effect.

Application programs used in the experiments were a matrix multiplication program for 512 $\times$ 512 matrices, referred to as MATMUL, and a PDE solver by SOR method with 500 iterations for a problem size of 1024 $\times$ 1024, referred to as SOR.

In both programs, matrices were distributed among APs in the block distribution fashion along the row direction. And the APs were mapped onto the one dimensional RP array. The granularity of data migration was changed in each experiment by adjusting the number of APs. The ratio of the number of APs in each RPs, referred to as data distribution ratio, was also changed in each experiment.

Definitions:

- Period—For a RP, a period corresponds to the execution of $n$ iterations of the outermost loop of all APs assigned to the RP. Note that $n$ is given by the remap option in the original code.
- $N_{AP}$, $N_{RP}$—AP Count, and RP Count, respectively.
- $RP_k$— $k$th Real Processor ($k = 1, 2, ..., N_{RP}$)
- $AP_k(t)$—The number of APs assigned to $RP_k$ during $t$th period.
- $Rcomp_k(t)$—Execution time at $RP_k$ for $(t-1)$th period excluding the time for synchronization and communication.
- $Acomp_k(t)$—Execution time for one AP in $RP_k$ for $(t-1)$th period excluding the time for synchronization and communication.
- $Restim_k(t)$—Estimated execution time at $RP_k$ for $t$th period excluding the time for synchronization and communication.

Procedure:

1. [Negotiation Phase] At the end of $(t-1)$th period, all real processor $RP_k(k = 1, 2, ...N_{RP})$ stop their execution by entering a synchronization phase. When the synchronization is completed, they start their execution for the negotiation.
2. Each RP calculates $Acomp_k(t)$ from $Rcomp_k(t)$ and $AP_k(t-1)$ as follows.
$$Acomp_k(t) = \frac{Rcomp_k(t)}{AP_k(t-1)}$$
3. Determine $AP_k(t)$, for all $k$, such that they hold the following equation.
$$Restim_1(t) = Restim_2(t) = ... = Restim_{N_{RP}}(t)$$
$$\text{where} Restim_k(t) = Acomp_k(t) \times AP_k(t)$$
$$\sum_{k=1,..N_{RP}} AP_k(t) = N_{AP}$$
4. Find an new index `new_AP_RP_Map[]` such that real processor $k$ makes its responsible for successive $AP_k(t)$ iterations, or APs, as long as it's possible.
5. [Data Migration Phase] The data migration takes place here based on the `new_AP_RP_Map[]`.
   - Each RP updates its `AP_RP_Map[]` to `new_AP_RP_Map[]`.
   - Perform data migration for the data whose owner abstract processors are migrated to/from another RP.
   - Each RP makes its *ExecList* for the next period.
6. Start execution for the $t$th period.

Figure 4. The Procedure of the Dynamic Load Distribution.

Table 1. Comparison of the Codes' Quality.

| | Code Type | Abstract Proc. Count | Exec. Time(s) | List Gen. Time(ms) | Scalability |
|---|---|---|---|---|---|
| MAT MUL | Seq | 1 | 126.4 | — | — |
| | Homo | 4 (1×4) | 37.01 | — | 3.40 |
| | Hetero | 4 (1×4) | 37.14 | 0.2 | 3.40 |
| | Hetero | 128 (32×4) | 37.13 | 0.2 | 3.40 |
| | Hetero | 256 (64×4) | 36.79 | 0.2 | 3.43 |
| SOR | Seq | 1 | 300.11 | — | — |
| | Homo | 4 (1×4) | 100.32 | — | 2.99 |
| | Hetero | 4 (1×4) | 98.48 | 0.17 | 3.05 |
| | Hetero | 128 (32×4) | 98.42 | 0.15 | 3.05 |
| | Hetero | 1024 (256×4) | 98.31 | 0.15 | 3.05 |

Seq: Sequential Code
Homo: Homogeneous code, Hetero: Heterogeneous code

## 4.1. Evaluation of the Quality of Codes

In the first experiment, we compared the execution time of the code for the HCE (Heterogeneous code) with the one of the code for homogeneous computing environment(Homogeneous code). In the case of the former code, the time spent to make *ExecList* was also measured for the evaluation. In this experiment, same amount of APs were assigned to each RPs and the dynamic load balancing was not carried out during the experiment.

Table 1 shows the execution times with four SS20, except for the truly sequential code which runs on one SS20. In this experiment, the environment itself was a homogeneous environment.

As shown in this table, we could not find any overhead due to the additional code for the HCE. This is an effect of the code refinement at the loop execution part as we mentioned in sec.3.2.2. We also found that the time spent for generating the *ExecList* was less than one millisecond and was negligibly small.

## 4.2. Evaluation of the Effect of the Data Distribution Ratio

In this experiment, we have examined how execution time varies with the change of data distribution ratio among two SS20s and two ULTRAs.

Fig.5 shows a result of this experiment for SOR. The x-axis of this graph indicates the ratio of the amount of data assigned to two ULTRAs to the total amount of data. The curves of calculation times for the ULTRA and the SS20 does not include the communication time. For the reference, this figure also shows the execution times of SOR under two homogeneous environments where four SS20s and four ULTRAs were used respectively.

As shown in this figure, when the same amount of data

Table 2. Summary of the Convergence Test.

| Number of AP | Time (s) Exec. | Time (s) Redist. [Occurrence] | Data Dist. Ratio Init. U:S | Data Dist. Ratio Converged U:S | Exec. Time (Static) |
|---|---|---|---|---|---|
| 16 | 67.44 | 2.76[1] | 4:4 | 6:2 | 68.58 |
| 32 | 64.99 | 3.03[3] | 8:8 | 11:5 | 58.33 |
| 64 | 65.57 | 2.65[1] | 16:16 | 23:9 | 59.97 |

U: ULTRA, S: SS20

Table 3. Dynamic Behavior to the External Load.

| Program | Exec. Time (s) | Redist. Time (s) [Occurrence] | Final Load Ratio U:U(Loaded):S:S |
|---|---|---|---|
| ① | 357.6 | — | 16:16:16:16 |
| ② | 386.3 | — | 23:23:9:9 |
| ③ | 335.9 | 12.9[32] | 26:16:11:11 |

U: ULTRA, S: SS20

is assigned among all of four machines ($x = 0.5$), total performance in this environment is roughly 10% better than that in the environment with four SS20s. And as $x$ increases to some extent, this performance goes closer to the performance in the environment with four ULTRAs. When $x = 11/(11 + 5)$, these four machines achieve the highest performance. Further increase of $x$ leads to performance decrease. We have got the similar result for MATMUL program though it is not shown here.

## 4.3. Evaluation of the Effect of the Dynamic Data Distribution

To show the effect of dynamic data distribution, two experiments have been made using SOR program with remap option of 50 at the outmost loop for the time step. Two SS20s and two ULTRAs were used as our experimental environment.

### 4.3.1 Convergence Test

The first experiment was to check if data distribution ratio converges into the optimum value even if the maximum computing powers of the collaborating processors are not given or unknown at the beginning of the execution. In this experiment, an equal amount of data was assigned initially to each machine.

Table 2 shows the result of the experiments with three different AP counts. Redistribution time in this table indicates the time spent for measuring the execution time and for scheduling and redistribution. The numbers written between the brackets shows how many redistributions occurred during the execution. "Execution Time(Static)" in the table presents the execution time without dynamic data redistribution but with the static data distribution of the ratio shown as the converged distribution ratio.

For every AP counts of 16, 32 and 64, the data distribution ratio converged into the optimum value after 1$\sim$3 times of data redistributions.

### 4.3.2 Dynamic Behavior Test

The second experiment was to examine how four processors behave when an unexpected load change happen in one of them. SOR program with 2000 time steps is executed with 64 APs assigned to two SS20s and two ULTRAs. At 100 seconds after starting execution, another heavy load was placed on one of the ULTRAs and was continued to run.

The experiment was made by three different runs under the following conditions respectively.

① Initial data distribution ratio of 1 to 1(SS20:ULTRA), and without dynamic data redistribution.

② Initial data distribution ratio of 9 to 11(SS20:ULTRA) which is the converged distribution ratio in the previous experiment and without dynamic data redistribution.

③ Initial data distribution ratio of 1 to 1(SS20:ULTRA), and with dynamic data redistribution.

Table 3 shows the summary of this experiment. As shown in this table, we could find that, if dynamic data redistribution was performed, a part of data assigned to the loaded ULTRA was migrated appropriately to the other ULTRA and SS20s and the execution time became the shortest of all.

## 5. Conclusion

We have proposed a technique to generate optimized code which has the ability to adaptively and dynamically distribute tasks among computers according to their available computing power, and described its implementation using our parallelizing compiler TINPAR[10, 7].

As the results of the performance evaluation, we could confirm that the generated codes are effectively executed in the HCE even if unexpected load change happens. Though it is not investigated yet, we believe these results genelarize to various applications.

In order to improve the performance of dynamic load balancing, we are currently implementing a technique [13] which reduces the number of global communication to decide *Data Distribution Ratio* at negotiation phase. We are also investigating the issue of the threshold and the granu-

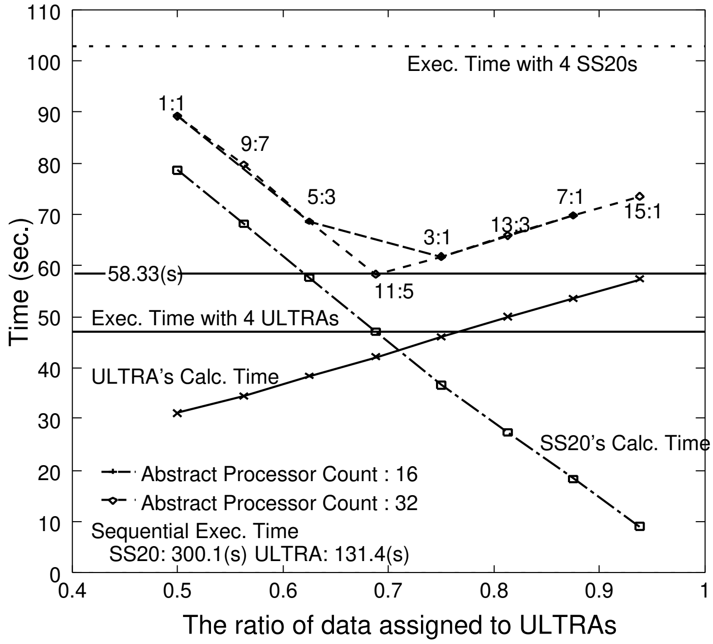



Figure 5. Variation of Execution Times of SOR.

larity of data migration in detail to improve the behavior of the dynamic load balancing.

## Acknowledgment


We would like to thank all the members of Tomita Lab. for their useful discussions. A part of this research is supported by the Grant-in-Aid for Scientific Research, #09680334, #09780268 and #00093007, from the Ministry of Education, Science, Sports, and Culture, Japan.


## References


[1] P. Bridges et al. *Users' Guide to MPICH, a Portable Implementation of MPI*. Argonne National Laboratory, 1995.

[2] G. Edjlali et al. Data parallel programming in an adaptive environment. *Proc. of 9th Int'l Parallel Processing Symp.*, pages 827–832, April 1995.

[3] D. J. Evans and W. Butt. Dynamic load balancing using task-transfer probabilities. *Parallel Computing*, 19(8):897–916, 1993.

[4] H. P. F. Forum. *High Performance Fortran Language Specification*. Rice University, 1995.

[5] M. P. I. Forum. *MPI: Message-Passing Interface Standard*. 1995.

[6] G. Fox et al. *Fortran D language specifications.* Technical Report SCCS 42c, NPAC, Syracuse Univ., Syracuse, NY, 1990.

[7] S. Goto et al. Generating optimized code for heterogeneous computing environment using parallelizing compiler tinpar. *Sig Notes of IPS Japan*, (97-PRO-10-2):7–12, October 1996.

[8] A. S. Grimshaw, J. B. Weissman, et al. Metasystems: An approach combining parallel processing and heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 21(3):257–270, 1994.

[9] A. Kubota et al. A parallelizing compiler technique for loops with irregular accesses –new algorithms to improve the performance of the inspector/executor–. *Trans. of IPS Japan*, 35(4):532–541, April 1994.

[10] Y. Miyoshi et al. TINPAR: A parallelizing compiler for message-passing multiprocessors. *Trans. of IPS Japan*, 37(7):1265–1275, July 1996.

[11] B. Moon et al. Runtime support and dynamic load balancing strategies for structured adaptive applications. *Proc. of the 7th SIAM Conf. on Parallel Processing for Scientific Computing*, pages 575–580, February 1995.

[12] J. Moreira et al. A programming environment for dynamic resource allocation and data distribution. *Proc. 9th Int'l Workshop on Languages and Compilers for Parallel Computing(LCPC'96)*, pages 553–569, August 1996.

[13] S. Tanaka. *hetero-TINPAR: A Parallelizing Compiler for Heterogeneous Computing Environment –Improvement of Dynamic Load Balancing Scheme–*. Bachelor Thesis, Dep. of Info. Sci., Kyoto Univ., 1997.

[14] M. Wolfe. The tiny loop restructuring research tool. *Proc. of Int'l Conf. on Parallel Processing*, II:46–53, 1991.