ICOT Technical Report: TR-526

TR-526

A Multi-Level Load Balancing Scheme for OR-Parallel Exhaustive Search Programs on the Multi-PSI

by
K. Taki, N. Ichiyoshi & M. Furuichi

December, 1989

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

A Multi-Level Load Balancing Scheme for OR-Parallel Exhaustive Search Programs on the Multi-PSI

Masakazu Furuichi
Information Systems and Electronics Development Lab.
Mitsubishi Electric Corporation
5-1-1 Ofuna, Kamakura 247, JAPAN

Kazuo Taki, Nobuyuki Ichiyoshi
Institute for New Generation Computer Technology
1-4-28 Mita, Minato-ku, Tokyo 108, JAPAN

Abstract

Good load balancing is the key to deriving maximal performance from multiprocessors. Several successful dynamic load balancing techniques on tightly-coupled multiprocessors have been developed. However, load balancing is more difficult on loosely-coupled multiprocessors because inter-processor communication overheads cost more. Dynamic load balancing techniques have been employed in a few programs on loosely-coupled multiprocessors, but they are tightly built into the particular programs and not much attention is paid to scalability. We have developed a dynamic load balancing scheme which is applicable to OR-parallel programs in general. Processors are grouped, and work loads of groups and processors are balanced hierarchically. Moreover, it is scalable to any number of processors because of this multi-level hierarchical structure. The scheme is tested for the all-solution exhaustive search Pentomino program on the mesh-connected loosely-coupled multiprocessor Multi-PSI, and speedups of 28.4 times with 32 processors and 50 times with 64 processors have been attained.

1 Introduction

Load balancing is essential in deriving maximal performance from multiprocessors by efficiently utilizing the processing power of the entire system. This is done by partitioning a program into mutually independent or almost independent tasks, and distributing tasks to processing elements (PEs) in order to balance work loads. The study of effective load balancing scheme has been intensely pursued in the last decade in many different areas in the field of tightly-coupled multiprocessors, loosely-coupled multiprocessors and distributed computer systems [1, 2, 4, 5, 6, 7, 9, 11, 14]. Several successful dynamic load balancing techniques on tightly-coupled multiprocessors have been developed [7]. However, dynamic load balancing on loosely-coupled multiprocessors is more difficult because the cost of inter-processor communication must be taken into account.

In certain application programs such as numerical computations, granularity, communication patterns and dependency of tasks can be estimated before the execution, and the best load distribution can be decided statically. However, for many programs it is not the case and dynamic load distribution is needed. A few successful dynamic load balancing techniques on loosely-coupled multiprocessors have been introduced [5, 6], but they are tightly built into the particular programs and scalability is not discussed much.

In this paper, a dynamic load balancing scheme which is in principle applicable to any OR-parallel exhaustive search program is described. In our scheme, processors are grouped, and work load is balanced both at the processor group level and at the processor level. This scheme is scalable to any number of processors. The scheme is tested for the all-solution search Pentomino program on the mesh-connected multiprocessor Multi-PSI [10] and near-linear speedups were attained.

In the following sections, the dynamic load balancing schemes are discussed in detail. Section 2 describes the on-demand load distribution scheme in general. Section 3 describes the multi-level load balancing scheme. Section 4 describes performance measurement and its observation. Section 5 describes the requirements of granularity for load distribution, the measurement analysis, and a method of applying this scheme to a given program. Section 6 concludes the paper.

2 On-Demand Load Distribution

In this section, a simple dynamic load balancing method is described. Load balancing is done by partitioning a program into mutually independent subtasks (Subtask Generation), and distributing subtasks to PEs so as to balance work loads (Subtask Allocation).

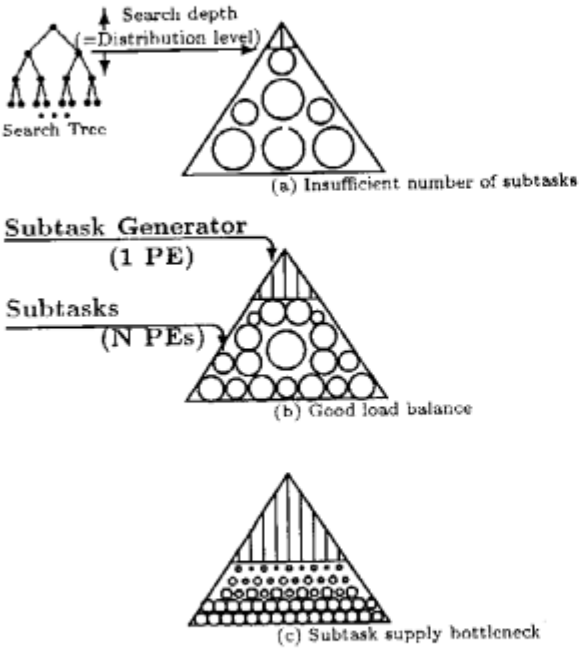


Figure 1: Subtask Generation

2.1 Subtask Generation

Subtask generation is done on one particular processor, which is called the master processor. Figure 1 shows the structure of subtask generation. The large triangle represent the entire search space, the OR-tree. The small triangle covered with vertical lines is the subtask generator, and it is executed on the master processor until the search reaches the certain tree depth which is called the distribution level. The circles are the generated subtasks, and their sizes indicate the sizes of subtasks. The size, or granularity of subtasks becomes smaller and the number of subtasks becomes larger as the distribution level gets deeper. Generated subtasks are distributed to PEs according to the task allocation strategy.

There are conflicting requirements to the granularity of subtasks. On the one hand, it should be small so that there are a large number of subtasks to make lots of processors busy. On the other hand, it should be significantly larger than the distribution overhead, since subtask supply becomes a bottleneck and much of the processing power would be wasted otherwise.

For example, Figure 1(a) illustrates a situation where the granularity of subtasks is large so that distribution overhead is low, but the number of subtasks may not be large enough to balance work loads. Figure 1(c) describes a situation where the number of subtasks is large enough to balance loads, but the distribution overhead might be too large. Therefore, some optimal distribution level as in situation like Figure 1(b) should be found which satisfies the requirements of low distribution overhead and of large number of subtasks.

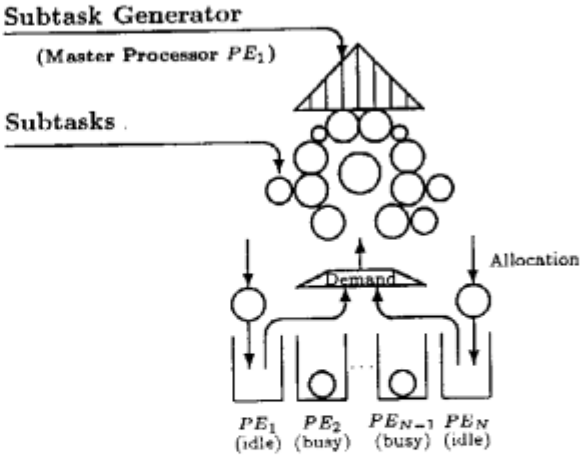


Figure 2: On-Demand Dynamic Load Distribution

2.2 Subtask Allocation

Generated subtasks are distributed to idle PEs in order to balance work loads. When a PE becomes idle, it sends a message to the master PE, requesting a new subtask. Figure 2 shows the structure of on-demand dynamic load distribution, the details are listed below.

1. Since all PEs are idle at the beginning, they send demand messages to the subtask generator.
2. Subtask generator distributes subtasks to idle PEs.
3. Each PE executes the subtask, and when it has completed execution of the subtask, it sends a demand message to the subtask generator.

There is some delay between the time when a PE becomes idle and the time when it is supplied with a new subtask. If it is not negligible compared to the average execution time of a subtask, a double buffering of subtasks should be introduced.

3 Multi-Level Dynamic Load Balancing Scheme

3.1 Subtask Supply Bottleneck

The simple load balancing method described in the previous section has one problem: it does not scale. As the number of processors increases, the rate of subtask execution eventually becomes larger than the rate of subtask supply. In other words, subtask supply becomes a

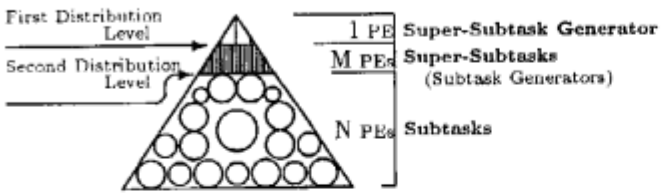


Figure 3: Two-Level Subtask Generation

bottleneck. Let N_{gen} be the number of subtasks generated and supplied per unit of time, and N_{exec} the number of subtasks solved per unit of time. Subtask supply becomes a bottleneck when $N_{gen} < N_{exec}$.

The bottleneck can be removed by reducing N_{exec} by making the subtask grains larger. However, this also makes the number of subtasks smaller, and loads cannot be balanced. Thus, we should increase N_{gen} by employing more processors for subtask generation.

3.2 Multi-Level Load Balancing

Figure 3 shows the structure of two-level subtask generation which solves this bottleneck problem. A super-subtask generator is allocated to one master PE. It divides the task into super-subtasks until the search reaches the first distribution level. Subtask generators are allocated to M PEs. They divide the super-subtasks into subtasks to distribute over the processors.

In our scheme, each subtask generators is in charge of a certain fixed number of processors, which form a processor group (PG). N processors are grouped into M processor groups, therefore, each PG is composed of $\frac{N}{M}$ PEs and a certain PE in a PG is called the group master PE.

At the first level distribution, super-subtasks are distributed to idle group master PEs to balance the loads of PGs. At the second level, subtasks are distributed to idle PEs to balance the loads of PEs which belong to a PG(Figure 4).

The two-level load balancing overcomes the subtask supply bottleneck. However, if the number of processors N becomes still larger, the subtask generator must feed an increasing number of processors, causing a subtask supply bottleneck. In such a case, three or more load balancing levels (multi-level load balancing) should be introduced.

3.3 Group Merging

The multi-level load balancing scheme with processor grouping is scalable, and has good locality because the super-subtasks are local to the processor groups. In practice, however, the number of super-subtasks may not be large enough, causing a load imbalance between

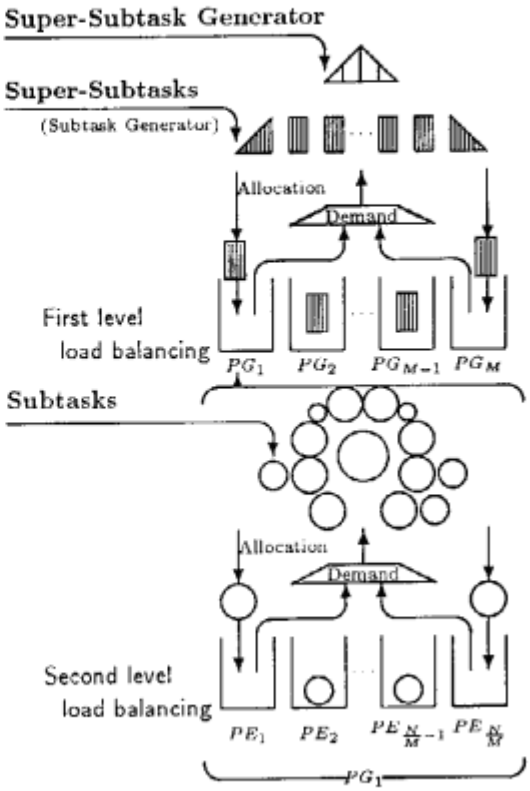


Figure 4: Two-Level Load Balancing

processor groups. To partially solve this problem, we introduced the group merging mechanism.

Figure 5 describes the structure of group merging. In Figure 5(1), N PEs are grouped into four PGs. Therefore, at the first level distribution, super-subtasks are distributed to 4 PGs dynamically. Suppose there are five super-subtasks, and the first four are distributed to PG_1, PG_2, PG_3 and PG_4. If all subtasks on PG_1 are completed in execution, the fifth super-subtask is distributed to PG_1. At this state, four PGs are all busy with generating subtasks and executing subtasks. When PG_2 finishes all work and becomes idle, there is no super-subtask left to be allocated. In such a case, PG_2 is merged into PG_1 (Figure 5(2)). In the same way, when PG_3 becomes idle, PG_3 is merged into PG_4 (Figure 5(3)), and at last, all PEs are merged into PG_1.

The problems with group merging is that as the groups are merged and become bigger, a subtask generator must feed an increasing number of processors, causing a subtask supply bottleneck. For now, the group merging scheme seems to work, but we will need to devise a better scheme that does not cause subtask supply bottleneck.

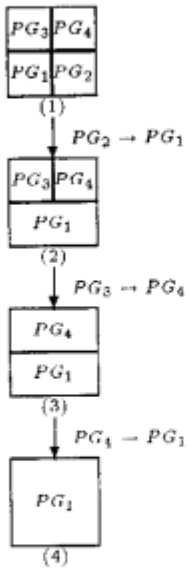


Figure 5: Group Merging

4 Measurements and Evaluation

We tested a two-level load balancing using the all solution search program of Packing Piece Puzzle, an OR-Parallel exhaustive search program on the Multi-PSI, a loosely-coupled multiprocessor. In this section, we describe the program, the hardware and the parallel-language used, then the measurements and observation are given.

4.1 Description of the Program

Packing Piece Puzzle is a puzzle, in which a rectangular box of containing pieces with various shapes are given (Figure 6). The problem is to find all possible ways to pack the pieces into the box. This puzzle is known as the Pentomino puzzle when the pieces are each made up of 5 squares.

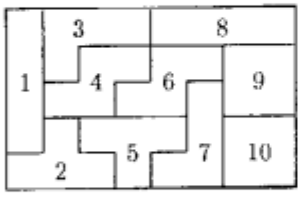


Figure 6: Packing Piece Puzzle

To solve this puzzle, the program starts with the empty box, and finds all possible placements of a piece to cover the square at the top left corner, then, for each of those placement, finds all possible placements of a piece (out of the remaining pieces) to cover the uncovered square which is the topmost leftmost, and so on until the box is completely filled. Each partly filled box defines an OR-node, where the possible placements of a piece to cover the uncovered topmost leftmost square define child nodes.

The program does a top-down exhaustive search of this OR-tree. Here, deepening the tree depth corresponds to packing one piece. Number of OR-nodes increases as the search level deepens, but since some OR-nodes are pruned when there are no more possible placements,the number of OR-nodes decreases below a certain tree depth.

4.2 Multi-PSI and the Parallel Language KL1

The performance is measured on the Multi-PSI [10] system developed in the Japanese fifth generation computer system (FGCS) project. The Multi-PSI system is a loosely-coupled multiprocessor running the concurrent logic programming language KL1 [3]. The processing element is the CPU of the PSI-II, and up to 64 processors are connected by an 8×8 mesh network. The parallel implementation is written in the microcode, and it attains 130K append RPS[1] per processor.

The concurrent logic programming language KL1 is based on Flat GHC, and is augmented by metaprogramming and pragma facilities. Those capabilities are used for writing an operating system and for load distribution. The operating system of Multi-PSI system is PIMOS [3] which is wholly written in KL1, and it provides basic OS functions such as resource management, I/O management, programming environment, and measurement capabilities. The performance reported in this paper is measured with these measurement capabilities supported by PIMOS and KL1 implementation.

4.3 Detection of Idle PEs

For the detection of idle PEs, the priority pragma capability of KL1 is used. Priority pragma is used for efficient program execution by scheduling tasks with broad range (0 to 4095) priority. Subtasks to solve a problem are scheduled with high priority, and tasks to send a demand message to master PE are scheduled with low priority. Low priority tasks are scheduled only when PEs become idle. This detection mechanism is very simple and the implementation overhead is quite small, and no operating system support is required.

[1] Reductions Per Second: A reduction is a logical inference step, and roughly corresponds to the execution of a simple statement in a procedural language.

	Number of Subtasks (N)	Total Reduction(R) ($\times 1,000$)	Total Reduction of Subtasks (S) ($\times 1,000$)	Total Reduction of Generator(G) ($\times 1,000$)	$\frac{G}{R}$ (%)	Average Reduction of Subtasks ($\times 1,000$)
L1	13	8,269	8,267	1.7	0.0	636.0
L2	118	8,273	8,267	5.4	0.0	70.1
L3	485	8,289	8,253	35.9	0.4	17.0
L4	1,583	8,321	8,201	120.6	1.4	5.2
L5	5,625	8,446	8,049	397.5	4.8	1.4
L6	16,124	8,854	7,774	1,080.4	12.2	0.5
L7	38,105	-	-	-	-	-
L8	64,980	-	-	-	-	-
L9	44,560	-	-	-	-	-
L10	3,106	-	-	-	-	-

Table 1: Number of Subtasks and Granularity

4.4 Granularity Measurements

Table 1 gives the measurement results of number of subtasks and their granularity. They are measured by executing the program on one processor. In the table, L1 through L10 in the row describe the tree depth corresponding to the distribution levels.

Number of Subtasks (N) in the row corresponding to Ln is the number of subtasks (OR-nodes) generated at each distribution level. Total Reduction (R) in the row corresponding to Ln (n$\geq$ 2) is the total number of reductions when two-level load balancing L1-Ln is done. Total Reduction (R) in the row corresponding to $L1$ is the total number of reductions when one-level load balancing $L1$ is done. Total Reduction of Subtasks (S) is the total number of reductions for all generated subtasks. Total Reduction of Generator (G) is the total reductions of super-subtask generator and subtask generators, and it is calculated by $R - S$. Rate of Generator Reduction ($\frac{G}{R} \times 100$) is the ratio of Total Reduction of Generator (G) to the Total Reduction (R). Average Reduction of Subtasks is the average granularity of subtasks, and it is calculated by $\frac{S}{N}$. Unit of Number of Subtasks is number, Rate of Generator Reduction is percent, and others are kilo ($\times 1,000$) reductions.

Figure 7 shows the number of subtasks against their granularity. X-axis is the number of reductions of subtasks, and Y-axis is the number of subtasks which corresponds to the number of reductions.

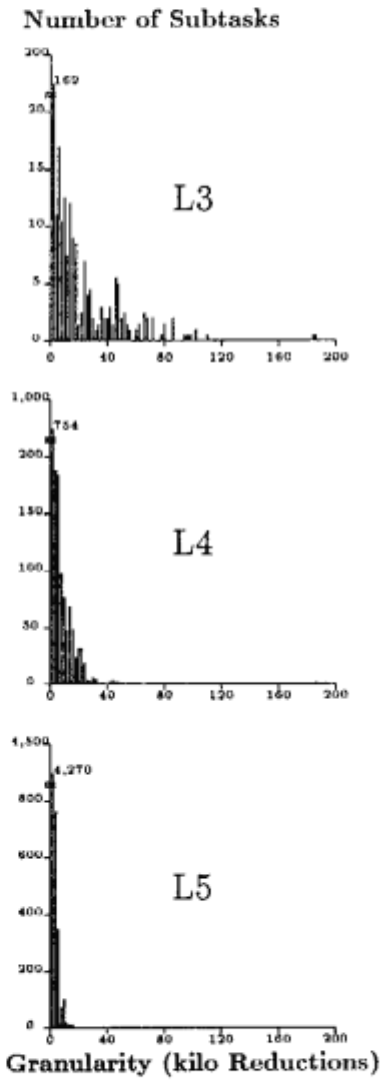


Figure 7: Granularity Distribution Map

No. of PEs		1	8	16	32	64
Exec.Time(sec)	L3	260.4	36.7	22.4	16.8	13.2
Speedup	L3	1	7.1	11.6	15.5	19.7
Speedup Rate(%)	L3	100	88.8	72.5	48.4	30.8

Table 2: Performance of One Level Distribution

No. of PEs	1	8	16	32	64
Execution Time (seconds)					
L1-L3	259.2	33.7	18.8	10.4	7.8
L1-L4	261.2	34.2	17.5	9.2	5.3
L1-L5	262.4	37.8	18.7	9.8	5.8
L1-L6	278.4	46.9	26.5	17.8	14.2
L2-L3	260.3	34.5	19.7	10.9	6.6
L2-L4	264.8	34.4	17.6	9.4	5.3
L2-L5	265.1	37.3	18.8	9.7	5.3
L2-L6	273.7	50.8	26.7	15.5	11.2
Speedup					
L1-L3	1	7.7	13.8	24.9	33.2
L1-L4	1	7.6	14.9	28.4	49.3
L1-L5	1	6.9	14.0	26.8	45.2
L1-L6	1	5.9	10.5	15.6	19.6
L2-L3	1	7.5	13.2	23.9	39.4
L2-L4	1	7.7	15.0	28.2	50.0
L2-L5	1	7.0	14.1	27.3	50.0
L2 L6	1	5.4	10.3	17.7	24.4
Speedup Rate(%)					
L1-L3	100	96.3	86.3	77.8	51.8
L1-L4	100	95.0	93.1	88.8	77.0
L1 L5	100	86.3	87.5	83.8	70.6
L1-L6	100	73.8	65.6	48.8	30.6
L2-L3	100	93.8	82.5	74.7	61.6
L2-L4	100	96.3	93.8	88.1	78.1
L2-L5	100	87.5	88.1	85.3	78.1
L2-L6	100	67.5	64.3	55.3	38.1

Table 3: Performance of Two Level Distribution

4.5 Performance Measurements

Execution times are measured on a 64-processor Multi-PSI system. For one-level load distribution, they are measured for various numbers of processors (1, 8, 16, 32, 64 PEs), for distribution level L3, which obtained the best performance; it is shown in Table 2. For two-level load distribution, they are measured for various numbers of processors, for the various pairs of the first distribution level (L1, L2) and second distribution level (L3, L4, L5, L6). Notation Li-Lj stands for the two-level load balancing with Li the first balancing level, and Lj the second.

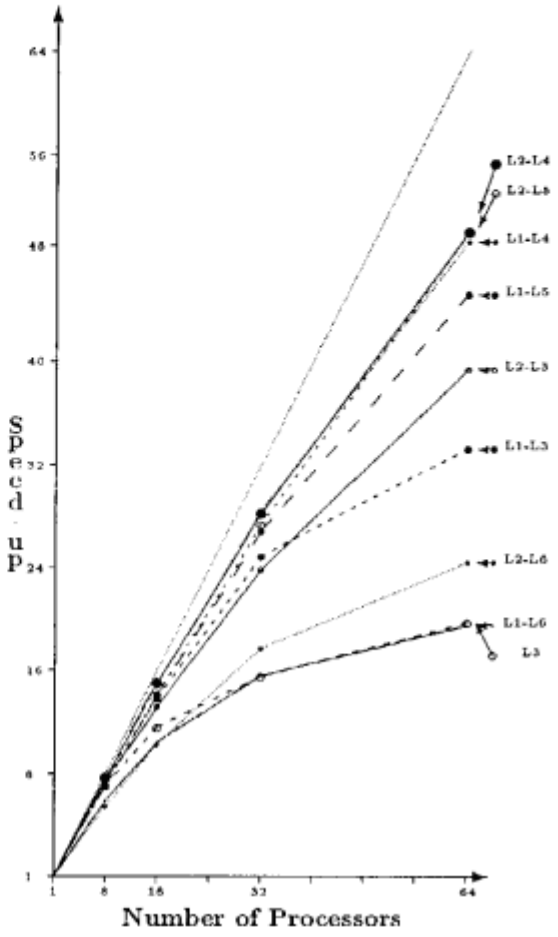


Figure 8: Speedups

One processor group (PG) contains 4 PEs in all of the two-level load balancing. Therefore, 64-PE is formed by 16 PGs, 32-PE by 8 PGs, 16-PE by 4 PGs and 8-PE by 2 PGs. As for the measurements for 1 PE, every subtasks are distributed to the same PG and same PE. Measured and calculated items are as follows. They are given in Table 3. Speedups are shown in Figure 8.

Execution Time (T_N) is measured by system clock of Multi-PSI system. Speedup (S_N) is defined as the ratio of execution time on 1 PE (T_1) to N PEs (T_N), and calculated by $\frac{T_1}{T_N}$. Speedup Rate (SR_N) is calculated by $\frac{S_N}{N} \times 100$.

4.6 Observation of Measurements Results

In Table 1, Number of Subtasks indicates that 13 subtasks are generated at distribution level L1, and it increases for up to level L8, it decreases because a lot of OR-nodes are pruned, and 3,106 at L10 is the total number of solutions.

Total Reduction (R) increases as the distribution level gets deeper. Here, the cost required for distributing one subtask is uniform, and it is about 35 reductions, therefore, as number of subtasks increases, the number of total reductions increases. It is also indicated by the increase of Total Reduction of Generator.

In Table 2, Execution Time with one-level load balancing decreases as the number of processors increases, but at 32 PEs or 64 PEs, Speedup Rates are saturated at less than 50%.

In Table 3, Execution Time of two-level load balancing decreases as the number of processors increases in any pair of Li-Lj, and near-linear speedups are obtained. The best performance on 64 PEs are L2-L4 and L2-L5, and 50 times speedups are obtained. Comparing the speedups of two-level load balancing with those of one-level load balancing, the performance of 32 PEs and 64 PEs was much improved. In this table, for each number of processors, the best and the second best speedups are marked with a rectangular box.

5 Discussion

As we have described in Section 2, there are conflicting requirements on the granularity of subtasks: a large number of subtasks for a good load balance and large granularity to avoid the subtask supply bottleneck. In this section, we will clarify these requirements using equations and a graph, and give another view to the multi-level load distribution. We also try to analyze the measurements result. A tentative plan, which describes how to apply this scheme to a given problem, is also presented.

5.1 Requirement on the Number of Subtasks

The number of subtasks has to be large enough to make lots of processors busy. This can be controlled by increasing the load distribution level. We will derive an equation which gives an approximate lower bound of the number of subtasks required to guarantee the processors' average work rate above a certain given value.

In the following, we fix an OR-parallel problem and the number of processors. We assume that response time from subtask request to subtask delivery is negligible compared to the subtask grain size. Subtask generators have enough throughput not to become a bottleneck.

Suppose there are N_{PEs} processors, each of $N_{PEs}-1$ processors processes K subtasks with uniform grain size equal to *Average_granularity*(Ave_gran), and the last processor processes K subtasks of the same size first and processes a task with *Max_granularity*(Max_gran) at last. The $N_{PEs}-1$ processors will be idle while the last processor processes the last subtask. This results in the lowest processor work rate for the given average size of subtasks. To guarantee a certain expected average work rate of processors ($Work\_rate$) in this worst case, the number of subtasks must be large enough. We determine below precisely how many subtasks are needed.

There are $K \times N_{PEs}$ subtasks (excluding the biggest subtask.)

$$N_{Subtasks} = K \times N_{PEs} \tag{1}$$

The actual work rate in the worst situation is given by the following formula.

$$Actual\_work\_rate = \frac{K \times Ave\_gran}{K \times Ave\_gran + Max\_gran} \tag{2}$$

The condition $Actual\_work\_rate \geq Work\_rate$ is equivalent to the following, using the above two equations.

$$N_{Subtasks} \geq N_{PEs} \times \frac{Work\_rate}{1 - Work\_rate} \times \frac{Max\_gran}{Ave\_gran} \tag{3}$$

Here, $Work\_rate$ is given by the user as a desired value; $Max\_granularity$ and $Average\_granularity$ depend on the program.

Suppose a system has 64 PEs, $Max\_granularity$ of a program is estimated ten times larger than $Average\_granularity$, and the user wishes an average work rate of at least 80%. Then, the condition

$$N_{Subtasks} \geq 64 \times \frac{0.8}{1 - 0.8} \times 10 = 2,560 \tag{4}$$

must be satisfied to guarantee the 80% work rate. This gives a guideline to choose the load distribution level which determines $N_{Subtasks}$ of a program. Condition (3) guarantees the expected work rate in the worst case. In average cases, however, a smaller $N_{Subtasks}$ may suffice to attain the expected work rate.

5.2 Requirement on the Subtask Granularity

Let us now consider a simple case where both subtask generation cost (C_{Gen}) and distribution cost (C_{Dist}) are paid only by the master processor. When there are N_{PEs} processors and the subtask granularity is just $(N_{PEs}-1)$ times larger than $(C_{Gen}+C_{Dist})$, a master processor always generates and distributes subtasks, and others execute them. In this situation, subtask supply does not become bottleneck, and $(N_{PEs}-1)$ times speedup out of N_{PEs} processors can be attained. When the granularity is smaller, subtask supply becomes a bottleneck, and it cannot make all processors busy. Thus, the lower bound of average granularity (Ave_gran), which keeps the subtask generation from becoming a bottleneck, can be expressed by the following inequality.

$$Ave\_gran \geq (C_{Gen} + C_{Dist}) \times (N_{PEs} - 1) \quad (5)$$

This condition comes from the requirement on the average throughput of subtask generation and supply, and does not guarantee that the processors are always kept busy. In a bad situation, subtasks of small grain size are generated in the early stage of execution, resulting in a subtask supply bottleneck. Therefore, *Average_granularity* should be sufficiently larger than the value determined by condition (5).

5.3 Problem Size and Speedup

Condition (3) gives a lower bound of number of subtasks, or equivalently, an upper bound of average granularity, to keep the system work rate above some expected value. Condition (5) gives a lower bound of average granularity, or equivalently, an upper bound of number of subtasks to prevent subtask supply bottleneck. Let us now consider the relationship between these upper and lower bounds of granularity, the problem size, and speedup.

Figure 9 illustrates the relation between speedup versus subtask granularity and number of subtasks intuitively, and (a) for a large problem and (b) for a small problem. Solid curved lines correspond to speedup when one-level load balancing is done.

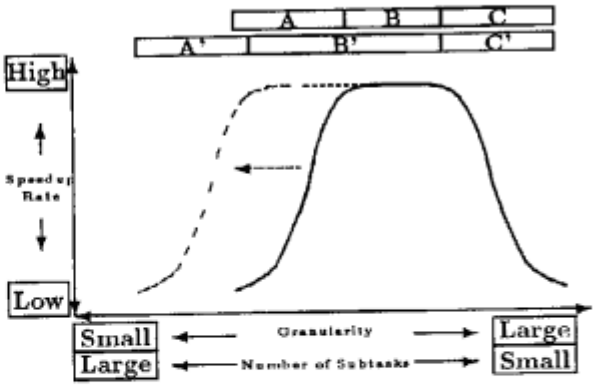
In region **A**, the speedup curves decline to the left because of the subtask supply bottleneck (condition (5) is not satisfied). In region **C**, the speedup curves decline to the right because of load imbalance (condition (3) is not satisfied).

When Figure 9(b) is put upon 9(b), the two curves for different problem sizes are close to each other in region **A**, because the performance decrease is affected by small granularity itself described in condition (5). On the other hand, the two curves lie far apart in region **C**, because the performance decrease in region **C** is caused by the small number of subtasks which have a relation with *Problem_size* and *Average_granularity* as below:

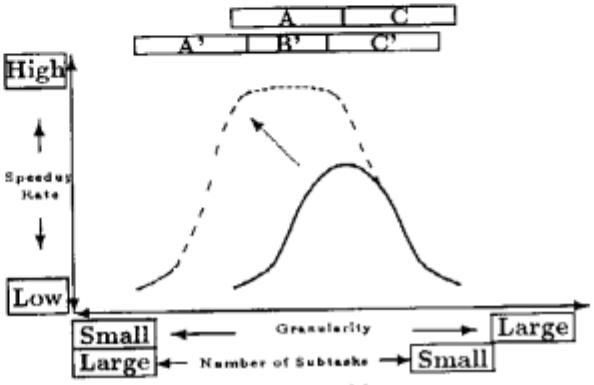
$$N_{Subtasks} = \frac{Problem\_size}{Average\_granularity} \quad (6)$$

For the large problem size, there is a wide plateau (region **B**) between **A** and **C** in which the highest speedup is obtained. The larger the problem is, the wider the plateau. It indicates that load balancing of large problems is much easier than that of small one, since the optimal point of the granularity is widely spread.

For the small problem size, region **A** and **C** may overlap, the curve shows a peak, and there is no plateau (region **B**). So it is more difficult to find a suitable granularity for the best speedup. This is caused by both subtask supply bottleneck and load imbalance.



(a) Large Problem



(b) Small Problem

Figure 9: Granularity and Speedup

Let us look at our multi-level load balancing scheme again using Figure 9. The scheme solves the subtask supply bottleneck which is caused by small granularity in region **A**. The scheme replaces the factor N_{PEs} in the condition (5) with the number of PEs in the processor group (N_{PEs_inPG}) where $N_{PEs\_inPG} \leq N_{PEs}$. Hence, the *Average_granularity* can be reduced. That is, the multi-levelling widens the region **B** in the speedup curve for the large problem, and moves the peak up for the small problem. These are shown as dotted curved lines in Figure 9(a) and (b).

Applying this scheme to problems showing speedup saturation, peak speedup will be improved. Thus, the multi-level distribution scheme has wider applicability to various sizes of problems than a single-level scheme does.

5.4 Analysis of the Measurement Results

We now try to derive the required number of subtasks for the Pentomino program from condition (3). Let *Work_rate* be 80%, *Max_granularity* be 180 from Figure 7 (granularity data of L3 is used),

Average_granularity be 17 from Table 1, and N_{PEs} be 64. Then the required Number of Subtasks ($N_{Subtasks}$) is 2,710.

Next, we try to derive the required average granularity of subtasks from condition (5). In this condition, C_{Gen} depends on the application programs which is about 40 reductions for this Pentomino program, while C_{Dist} does not change for all programs which is about 35 reductions. In our measurements, we have fixed the number of PEs in a PG (N_{PEs_inPG}) at 4. *Average_granularity* must be at least 225 reductions by condition (5).

Table 1 gives the number of subtasks and granularity for each level. The required number of subtasks 2,710 is satisfied by L5 and L6, and the required granularity is satisfied by L1 to L6. Hence, L5 and L6 seem to be the optimal distribution level.

Table 3 gives the performance of two level distribution. The number of subtasks at L3 is too small, which causes the load imbalance, therefore, good speedups are not obtained for L1-L3 and L2-L3. The number of subtasks at L4 is rather small, but the load imbalance problem does not show itself (recall condition (3) was not a must), and good speedups are obtained for L1-L4 and L2-L4.

Both requirements are satisfied for L5 and L6, and good speedups are obtained for L1-L5 and L2-L5, but not for L1-L6 and L2-L6. This is caused by the following problem of current group merging mechanism.

When the PEs in a PG are merged to other PG, becoming larger group, N_{PEs} in condition (5) also becomes larger. For example, N_{PEs} is 4 at the beginning, but as it becomes 64, then the required average granularity becomes 4,625 reductions.

Then L5 and L6 do not satisfy this requirement. Observing Table 3, subtask supply bottleneck limits the performance at the distribution level of L1-L6 and L2-L6. However, good speedup is obtained at L1-L5 and L2-L5, the group merging seems not to make the subtask supply bottleneck.

Lastly, let us consider where L3, L4, L5, and L6 are located in the graph of Figure 9(b). Looking at both Table 3 and the graph, it is observed that L3 corresponds to the region **C'**, since speedup is saturated because of load imbalance. L6 corresponds to the region **A'** since it is saturated because of the subtask generation bottleneck. L4 and L5 correspond to the region **B'** and the best speedups are obtained.

5.5 Determination of Optimal Load Balancing Levels

Here, let us present how to apply the multi-level load balancing scheme to a given OR-parallel problem. Optimal number of subtasks, granularity of subtasks, and load balancing level can be determined by the following procedure.

(a) Find the size of the problem by estimation or measurement.

(b) Give a certain expected work rate, and derive the required number of subtasks from condition (3).

(c) Derive the average granularity from (a) and (b).

(d) Find the cost of subtask generation by estimation or measurements. Determine the cost of distributing a subtask which is constant. Then derive the number of PEs in a processor group from condition (5).

(e) When the size of the problem is large, the number of PEs in the condition (5) becomes equal to or larger than that in the condition (3), then one level distribution suffices.

(f) When $2 \leq N_{PEs}$ (in the condition (5)) $\leq N_{PEs}$ (in the condition (3)), a multi-level load balancing is suitable.

(g) When N_{PEs} (in the condition (5)) ≤ 2, the problem size is too small and efficient speedup may not be expected. So, decrease the number of subtasks with decreasing the number of processors to be used, and try one-level distribution.

Programmers may derive near-maximum speedup by applying the described procedures to the given problem. They can get the best speedup by tuning the distribution level with a few trials-and-errors.

6 Conclusions and Future Works

A multi-level dynamic load balancing scheme for OR-parallel problems on loosely-coupled multiprocessors is proposed. Processors are grouped in our scheme, and work loads of groups (PGs) and processors (PEs) are balanced dynamically in each hierarchy level. This scheme is scalable to any number of processors.

Loads of PGs and PEs can be balanced when enough number of super-subtasks and subtasks are supplied, however, when the number of super-subtasks is not large enough, load imbalance between processor groups is occurred. We introduced group merging to solve this problem partially.

These schemes are implemented and tested for the all-solution exhaustive search Pentomino program on the mesh-connected loosely-coupled multiprocessor Multi-PSI, and near-linear speedups are obtained: 7.7 times with 8 PEs, 15 with 16 PEs, 28.4 with 32 PEs, 50 with 64 PEs. The relations between granularity, number of processors and speedups are discussed, and this

would be an essential guide to think about the load balancing for OR-parallel problems on loosely-coupled multiprocessor in general. This scheme is efficient not only for OR-parallel search problems, but also applicable to some types of search problems such as alpha-beta pruning problems, which do not involve frequent interprocessor communication. Applying the multi-level load balancing scheme to such programs is our future work. Improvement of the group merging scheme is also the future work.

7 Acknowledgments

We would like to thank the ICOT Director, Dr. K.Fuchi, and the chief of the fourth research laboratory, Dr. S.Uchida, for giving us the opportunity to conduct this research. We would also like to thank K.Nakajima, a former member of ICOT, who has returned to Mitsubishi Electric, who helped us to prepare the measurements tools, and all researchers of ICOT and the cooperating companies who gave us valuable suggestions.

References

[1] K. M. Baumgartner, and B. W. Wah. "GAMMON: A Load Balancing Strategy for Local Computer Systems with Multiaccess Networks". In *IEEE Transactions of Computers*, Vol. 38, No.8, pages 1,098–1,109, Aug. 1989.

[2] T. Chikayama. "Load balancing in a very large scale multi-processor system". In *Proceedings of Fourth Japanese-Swedish Workshop on Fifth Generation Computer Systems*. SICS, 1986.

[3] T. Chikayama, H. Sato, and T. Miyazaki. "Overview of the parallel inference machine operating system (PIMOS)". In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*, pages 230–251, 1988.

[4] T. C. K. Chou, and J. A. Abraham. "Load Balancing in Distributed Systems". In *IEEE Transactions of Computers*, Vol. SE-8, No.4, pages 401–412, Jul. 1982.

[5] E. W. Felten and S. W. Otto. "A highly parallel chess program". In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*, pages 1001–1009, Dec. 1988.

[6] C. Ferguson, and R. E. Korf. "Distributed Tree Search and its Application to Alpha-Beta Pruning". In *Proceedings of the Seventh National Conference on Artificial Intelligence 1988*, Vol. 1, pages 128–132, Aug. 1988

[7] A. George, M. T. Heath, J. Jiu, and E. Ng. "Solution of Sparse Positive Definite Systems on a Shared-Memory Multiprocessor". In *International Journal of Parallel Programming*, Vol. 15, No. 4, pages 309–325, 1986.

[8] S. Hiroguchi, and Y. Shigeki. "Optimal Number of Processors for Finding the Maximum Value on Multiprocessor Systems". In *Proceedings of The Twelfth Annual International Computer Software and Applications Conference 1988*, pages 308–315, Oct. 1988

[9] V. Kumar and V. N. Rao. "Parallel Depth-First Search, Part I: Implementation. In *International Journal of Parallel Programming*, 16(6), 479–499, 1988

[10] K. Nakajima, Y. Inamura, N. Ichiyoshi, K. Rokusawa, and T. Chikayama. "Distributed implementation of KL1 on the Multi-PSI/V2". In *Proceedings of the Sixth International Conference on Logic Programming*, pages 436–451, 1989.

[11] L. M. Ni, and Kai Hwang. "Optimal Load Balancing Strategies for A Multiple Processor System". In *Proceedings of 10th International Conference of Parallel Processing 1981*, pages 352–357, Aug. 1981

[12] D. M. Nicol, and F. H. Willard. "Problem Size, Parallel Architecture, and Optimal Speedup". In *Journal of Parallel And Distributed Computing*, 6, pages 404-420, 1988

[13] S. Pulidas. "Imbedding Gradient Estimators in Load Balancing Algorithms". In *Proceedings of 8th International Conference on Distributed Computing Systems 1988*, pages 482-490, Jun. 1988

[14] A. N. Tantawi. "Optimal Static Load Balancing in Distributed Computer Systems". In *Journal of the Association for Computing Machinery*, Vol. 32. No. 2, pages 445-465, April 1985.

[15] E. Tick. "Compile-Time Granularity Analysis for Parallel Logic Programming Languages". In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*, ICOT, 1988.

[16] J. Schaeffer. "Distributed Game-Tree Searching". In *Journal of Parallel And Distributed Computing*, 6, pages 90-114, 1989