TR-587

A Study of Mapping of Locally Message Exchanging Algorithms on a Loosely-coupled Multiprocessor

by
K. Wada & N. Ichiyoshi

August, 1990

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

(03)3456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

A Study of Mapping of Locally Message Exchanging Algorithms on a Loosely-coupled Multiprocessor

Kumiko Wada
Nobuyuki Ichiyoshi
Institute for New Generation Computer Technology (ICOT)
1-4-28 Mita, Minato-ku, Tokyo 108, Japan
phone: +81-3-456-3193
email: wada@icot.or.jp, ichiyoshi@icot.or.jp

keywords: parallel computation, loosely-coupled multiprocessors, load distribution, communication overhead, distributed shortest path algorithm.

Abstract

Good load balance is the key to get the most from the processing power of parallel computers. Dynamic load balancing techniques have been developed and proved successful for tightly-coupled multiprocessors, and for loosely-coupled multiprocessors when the problem is composed of independent tasks with large granularity. But the problem of mapping intercommunicating processes on loosely-coupled multiprocessors has not been well explored. In this paper, we consider the problem when there is locality in message communication (*locally message exchanging algorithms*). As an example, we developed a distributed algorithm for a single-source shortest path problem and tested three different static mappings of a large grid graph onto a mesh-connected multiprocessor, Multi-PSI. Two of them resulted in relatively good performance, and were shown to be a good compromise between communication localization and load balance.

1 Introduction

Good load balance is the key to get the most from the processing power of parallel computers.

For tightly-coupled multiprocessors, fairly good load balance can be achieved by employing the task stealing technique, in which an idle processor *steals* a task from a common process pool or from the task queues of busy processors.

For loosely-coupled multiprocessors, task stealing is not usually realistic, because remote task searching would involve too great a communication overhead. When the whole problem can be divided into a large number of tasks whose average size is much larger than task migration cost, dynamic load balancing schemes that distribute tasks to idle processors can achieve much the same

effect as task stealing, *provided* inter-task communication is not frequent [9, 7]. When tasks communicate with one another frequently, random task distribution will increase communication overhead, and could lead to performance degradation. But the problem of mapping inter-communicating processes on loosely-coupled multiprocessors has not been well explored.

This paper studies a simple case of the above problem, namely mapping of a *locally message exchanging algorithm* onto loosely-coupled multiprocessors. By a *locally message exchanging algorithm*, we mean an algorithm that solves a problem using a large number of processes, each of which communicates with its close neighbors. Distributed graph algorithms (e.g., distributed algorithms for minimum cost spanning trees, maximum connected components, etc.) generally fall into this category. As an example, we chose to solve a single-source shortest path problem for a large grid graph on a mesh-connected multiprocessor, the Multi-PSI/V2 [10].

Three mapping schemes, *two-dimensional simple*, *two-dimensional multiple*, and *one-dimensional simple* mappings were examined. The latter two resulted in better overall performance than the first, and were shown to be a good compromise between communication localization and load balance.

2 Locally Message Exchanging Algorithms

We define a *locally message exchanging algorithm* as one which solves a problem using a large number of processes, each of which communicates with its close neighbors. Distributed graph algorithms (e.g., distributed algorithms for minimum cost spanning trees, maximum connected components, etc.) in which processes corresponding to the graph vertices communicate with their neighbors along the edges generally fall into this category.[1]

We considered mapping a locally message exchanging algorithm for a large-scale graph (with $> 1,000$ nodes) onto a medium-to-large scale loosely-coupled multiprocessor. We assumed that the ratio of inter-processor data access cost to intra-processor data access cost is fairly big, say 100. In this situation, suppression of inter-processor communication is of great concern for achieving net performance.

In a locally message exchanging algorithm, inter-*process* communication translates to local data access when the two communicating processes are mapped onto the same processor, whereas it translates to inter-*processor* communication when the two processes are mapped onto different processors. Moreover, if they are mapped onto two processors that are far apart, the messages between them go through a number of routing switches, making the network busier. This means that the mapping should preserve the locality of the processes as much as possible, in order to

[1] There are many graph algorithms that are not of this category. The Warshall-Floyd algorithm [14, 6] to determine all-to-all shortest path is an example.

suppress communication. This is called *communication localization*.

At the other extreme, if all vertices of the graph are mapped onto one processor (if possible), there will be no inter-processor communication, but only one processor will be utilized and no speedup gained. Even when the communicating processes are evenly distributed over the processors, uneven distribution of active processes during computation could lead to poor processor utilization.

Thus, we have to find a reasonable compromise between communication localization and good load balance in mapping a locally message exchanging algorithm on a loosely-coupled multiprocessor.

Hereafter, we will consider an example of mapping a grid graph onto a mesh-connected multiprocessor. We will actually test three different mappings, using a distributed single-source shortest path algorithm for a 200×200 grid graph on the mesh-connected multiprocessor, Multi-PSI. We will discuss the Multi-PSI system first.

3 Machine and Language

3.1 The Multi-PSI

The Multi-PSI/V2 [12, 10] is a prototype parallel inference machine developed at ICOT to provide the software development environment required for parallel software research. It is a loosely-coupled multiprocessor whose element processors are connected in a mesh network. It can connect up to 64 element processors using an 8×8 mesh network. Fig. 1 shows the logical processor configuration of the Multi-PSI. (Sixteen processors connected by a 4×4 mesh network.) The performance per processor is 130K append LIPS[2].

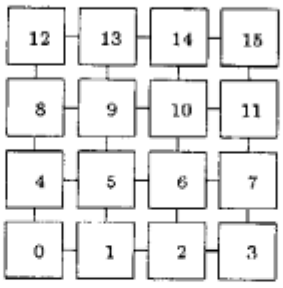


Figure 1: Logical processor configuration of the Multi-PSI with 16 processors

[2]LIPS stands for *Logical Inferences Per Second*. The performance of 130K append LIPS means the processor can append 130K cons cells per second.

3.2 KL1

On top of the hardware is an implementation of the concurrent logic language KL1 [3]. A KL1 program consists of a set of guarded Horn clauses [13] of the following form.

$$H \ :- \ G_1, \ \ldots, \ G_m \mid B_1, \ \ldots, \ B_n. \ (m, n \ > \ 0)$$

where H, G_i, and B_i are atomic formulas. H, G_i, and B_i are called the *head*, the *guard goals*, and the *body goals*, respectively. The vertical bar (|) is called the *commitment operator*. The logical reading of the above clause is, "If $G_1, \ldots, G_m$ and $B_1, \ldots, B_n$, then H." Operationally, when an active process matches H, and the tests $G_1, \ldots, G_m$ succeed, the process is reduced into child processes $B_1, \ldots, B_n$. The child processes can run concurrently. Synchronization is imposed solely by data-flow relations realized by guard part suspension mechanism. KL1 is suited for describing small-grain processes communicating with each other.

KL1 has a facility for specifying execution priority of processes and for specifying the processor in which the processes are to be executed. The priorities and processors can be specified by annotating the program with *pragmas*. Load distribution can be specified by attaching the load distribution pragma to the body goal (process creation) as follows:

$$Goal@\texttt{processor}(Proc).$$

The priority pragmas are used to specify the execution priority of the processes. A priority pragma is attached to the body goal as below:

$$Goal@\texttt{priority}(Prio).$$

Body goals without pragmas inherit the priority and processor of the parent goal.

The system provides a fine-grain priority scheduling: There are 4,096 priority levels in the current system. The priority scheduling is observed independently within each processor to avoid the large overhead incurred by centralized priority management.

Note that pragmas do not change the semantics of the program. Thus, semantics and mapping are clearly separated in KL1.

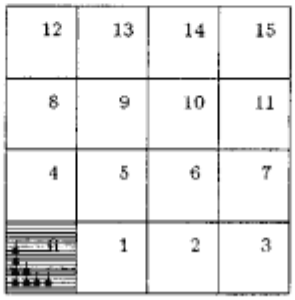
4 Mapping of Grid Graph on a Mesh-Connected Multiprocessor

In this section, we present three ways of mapping a large-scale grid graph on a mesh-connected multiprocessor. All three are static mappings — that is, a processor is assigned a fixed set of vertex processes during the whole computation. Dynamic allocation (or migration) of vertex processes seemed both difficult and unpromising, but its possibilities cannot be ruled out.

4.1 Two-Dimensional Simple Mapping

First, we give a very simple mapping which we call *two-dimensional simple mapping*. In this mapping, we employ $p = q^2$ processors, and divide the grid into $q \times q$ blocks and map each block onto the corresponding processor. For example, when the logical processor configuration of the Multi-PSI with 16 element processors is as shown in Fig. 1, the grid is divided into 4×4 blocks (Fig. 2). Each block has the same number of vertices. Processor P_0 is responsible for the shaded block.

This mapping preserves the locality of the grid very well, but unless the computation is carried out evenly on the whole grid during the whole execution, the load balance may not be good.



The shaded block is mapped onto processor 0.

Figure 2: Decomposition of a graph for two-dimensional simple mapping

4.2 Two-Dimensional Multiple Mapping

Since two-dimensional simple mapping can lead to poor load balance for an uneven distribution of active computation, putting each processor in charge of vertices from many different areas can be a good strategy. *Two-dimensional multiple mapping* is a simple way of doing this. Under this mapping, the grid is divided into k super-blocks, each of which is again divided into p blocks just as in two-dimensional simple mapping. Each processor is responsible for k blocks, each one from each superblock.

Fig. 3 shows the case of $k = 4 \times 4$, $p = 4 \times 4$. The graph is decomposed into 16 superblocks, and each superblock is decomposed again into 16 blocks in the same way as the two-dimensional simple mapping. The blocks for which processor P_0 is responsible are shaded in the figure.

Since the total length of the block boundaries becomes larger than the two-dimensional simple mapping, and since vertex-to-vertex communication translates to inter-processor communication when the edge between them crosses a block boundary, there tends to be more communication overhead in two-dimensional multiple mapping. Thus, this mapping is a way to achieve higher processor utilization, but at the cost of higher communication overhead.

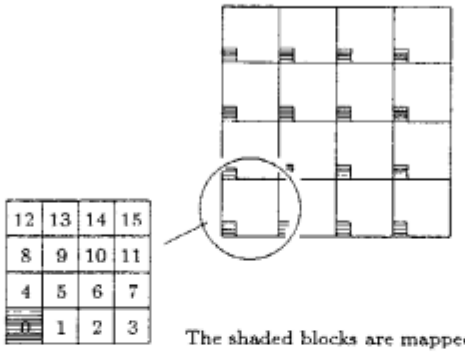


Figure 3: Decomposition of a graph for two-dimensional multiple mapping

4.3 One-Dimensional Simple Mapping

One-dimensional simple mapping is intended to get the same effect of two-dimensional multiple mapping by a simpler grid decomposition. We divide the grid simply as p narrow rectangular strips and map them onto the processors (see Fig. 4). It can be thought of as one particular way of two-level (block/superblock) decomposition of the grid: The grid is divided into p narrow vertical superblocks, each of which is divided into p blocks, and a processor is assigned one block from each superblock (The blocks mapped onto one processor are vertically aligned).

Compared to the two-dimensional mapping with $k = p$, this mapping has less randomness, but its total block boundary length is half of that in the former. If L is the length of the side of the grid, the block boundary length of the one-dimensional simple mapping is $(p-1) \cdot L$, while that of the two-dimensional multiple mapping is $2(p-1) \cdot L$. Fig. 4 shows an example with $p = 16$.

Figure 4: Decomposition of a graph for one-dimensional simple mapping

5 Shortest Path Problem

5.1 Problem Definition

The shortest path problem is described in terms of graph theory as follows. A directed graph $G = (V, E)$ is defined by a set of n vertices V and a set of e edges E. An edge is an ordered pair of vertices. The function c is the cost function, such that it maps from pairs of vertices to nonnegative reals. For an edge from vertex v_i to v_j, $c(v_i, v_j)$ is called the edge cost or the edge length. Technically, if there is no edge from v_i to v_j, then we assume $c(v_i, v_j) = +\infty$, some value much larger than any actual cost. Moreover, we set $c(v_i, v_i) = 0$. For $l+1$ vertices $v_0, v_1, \ldots, v_l$, if an edge e_i from v_i to v_{i+1} exists for $i = 0, 1, \ldots, l-1$, then a sequence of edges $P = (e_0, e_1, \ldots, e_{l-1})$ is called a path from v_0 to v_l. The cost of a path is just the sum of the costs of the edges on the path. P is a shortest path from v_i to v_j if its cost is minimum over the cost of all paths from v_i to v_j. The single-source shortest path problem is the problem of finding the shortest paths from a given vertex v_0, called the source, to every other vertex in V.

5.2 A Distributed Shortest Path Algorithm

We present a distributed algorithm for solving the single-source shortest path problem. Assume there are p processors, $P_0, \ldots, P_{p-1}$, which can communicate with each other over some communication medium. We decompose the set of vertices V into the direct sum of its subsets, i.e., subsets $V_0, \ldots, V_{p-1}$, such that $V = V_0 \cup V_1 \cup \cdots \cup V_{p-1}, V_i \cap V_j = \emptyset\ (i \neq j)$, and map each V_i onto processor P_i.

Each processor maintains the shortest known cost and path from the source vertex to each of the vertices mapped onto it. The cost and path associated with the vertex are updated by the *cost-path information*.

Cost-path information is of the form $cp(cost, v_j, v_i)$. It means that a path from the source to the vertex v_j exists and its cost is $cost$, and the predecessor vertex of v_j is v_i on the path. During the execution of the algorithm, each vertex v_j keeps the minimum known cost to that point and its predecessor vertex on the path in two variables $cost_j$, $path_j$. Given a cost-path information $cp(cost, v_j, v_i)$, $cost_j$, $path_j$ is updated to $cost$, v_i respectively when $cost < cost_j$. Then the processor creates new cost-path information $cp(cost + c(v_j, v_k), v_k, v_j)$ for each neighboring vertex v_k of v_j. At each processor, the cost-path information is kept in a priority queue and is dequeued in order by lower cost. When processor P_i creates the cost-path information for each neighboring vertex u of v, if vertex u belongs to the subset of vertices V_i, the information is enqueued in processor P_i's own priority queue. Otherwise, processor P_i sends the information by inter-processor message to processor P_j, which is responsible for $u \in V_j$. When processor P_j receives the cost-path

```
Initialization of processor P_0:
    begin
        for every vertex v_i which belongs to processor P_0 do cost_i := ∞, path_i := unknown ;
        initialize the local priority queue ;
        cost_0 := 0 ;
        for every neighboring vertex v_j of v_0 do
            send cp(c(v_0, v_j), v_j, v_0) to processor P_b that v_j belongs to
    end

Initialization of processor P_a (a ≠ 0):
    begin
        for every vertex v_i that belongs to processor P_a do
            cost_i := ∞ ;
        initialize the local priority queue ;
    end

Search for shortest paths at processor P_a:

    If the priority queue is not empty,
        begin
            dequeue cp(cost, v_j, v_i) from the priority queue ;
            if cost_j > cost then
                begin
                    cost_j := cost ;
                    path_j := v_i ;
                    for every neighboring vertex v_k of v_j do
                        send cp(cost + c(v_j, v_k), v_k, v_j) to processor P_b which v_k belongs to
                end
        end

    On receiving the message cp(cost, v_j, v_i),
        begin
            enqueue cp(cost, v_j, v_i) to the priority queue
        end
```

Figure 5: The distributed algorithm

information, it enqueues it in its own priority queue according to the cost.

Initial cost-path information for every vertex is $cost = +\infty, path =$ unknown, and all priority queues are empty. The processor to which the source vertex v_0 is mapped creates the cost-path information $cp(c(v_0, v_j), v_j, v_0)$ for each neighboring vertex v_j of the source v_0 and enqueues it in each priority queue of the processor to which v_j belongs. The algorithm terminates when all of p priority queues are empty. (This can be detected by using any distributed termination detection technique.) At this time, the cost of the shortest path and the predecessor vertex on the path are kept in $cost_j$ and $path_j$ for each vertex v_j. The distributed algorithm is given in Fig. 5.

The distributed algorithm reduces to Dijkstra's sequential algorithm when all vertices are mapped onto one processor (except difference in the processing of the cost-path information which is ignored in the later computation). At the other extreme, when the n vertices are mapped onto the same number of processors (i.e., $p = n$), the algorithm closely resembles Chandy and Misra's

distributed algorithm with overwritable message buffers [2]. Overwriting of a previous message in their algorithm corresponds to enqueuing of a cost-path information that is better than previous ones. This amounts to overtaking of intervertex messages, and reduces the computational complexity that would otherwise be exponential. Chandy and Misra's algorithm is more complex in that it can handle a graph with negative cycles.

5.3 Implementation in KL1

In our shortest path program, message exchanging vertex processes are represented by KL1 processes. The priority queue management was simply and efficiently realized by the priority pragma facility in the language in the following way. When a vertex process sends cost-path information to a neighbor vertex process, it spawns a temporary process to deliver it. A temporary process corresponding to a message with a lower cost is given a higher priority than one corresponding to a message with a higher cost. Thus, messages with lower costs are delivered earlier than those with higher costs, realizing the effect of a priority queue.

The mapping of vertices on the processors were specified by the load distribution pragmas. We were able to test different mapping strategies fairly easily, just by changing how the arguments to the pragmas are calculated, without affecting the logical specification of the program.

6 Measurements and Analysis

We tested the above-mentioned strategies of mapping grid graphs on the mesh-connected multiprocessor, Multi-PSI. The following presents the performance results and provides some analysis.

6.1 Graph Used in the Experiments

We used the following grid graph for our experiments.

Graph shape: The graph is a directed grid graph with 40,000 (200×200) vertices. The edges exist in both directions for all pairs of neighboring vertices. We placed the source vertex at one corner of the grid.

Costs for the graph edges: The costs for the edges are given by a pseudo-random number generator that generates nonnegative integers from 1 to 99.

6.2 Measurement Results and Analysis of Performance

We ran the distributed algorithm to solve the shortest path problem for the grid graph on the Multi-PSI, using the three mappings discussed, and with varying number of processors.

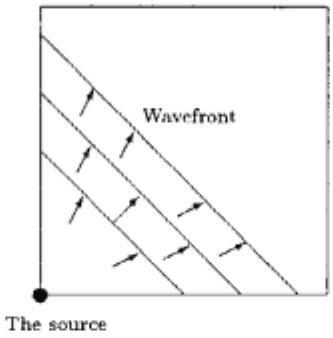


Figure 6: Wavefront

The execution times are shown in Fig. 9 and the speedups in Fig. 10. The vertical axis in Fig. 9 represents the execution time in seconds, and the horizontal axis represents the number of processors used. In Fig. 10, the vertical axis represents the speedup, and the horizontal axis represents the number of processors used.

6.2.1 Two-Dimensional Simple Mapping

The performance of the two-dimensional simple mapping was the worst among the three. A four-fold increase in the number of processors translates to a speedup of a little less than two-fold.

Let us consider the reason why we could not obtain good performance with this mapping. During the execution, the cost-path information is initially created at the processor in which the source vertex resides, after that, the information is propagated to other processors gradually like a wave. A processor is idle before the wave comes and becomes idle again after the wave has gone.

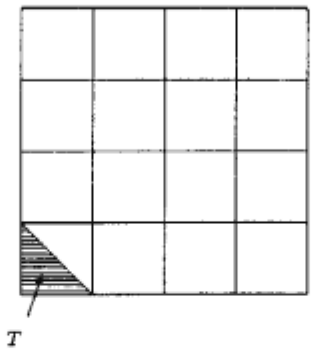


Figure 7: The estimation for the two-dimensional simple mapping

The speedup for this mapping can be estimated as follows, first ignoring interprocessor communication overhead and speculative computation[3] that may arise. We assume that the edge cost

[3] Computation that may turn out unnecessary afterwards. Heuristic search of subtrees to find a solution is a

distribution is more or less uniform, so that shortest paths do not deviate very much from Manhattan paths.[4] In such a case, cost-path information is expected to spread from the source vertex like a wavefront. At any point in computation, the wavefront consists of the set of vertices that are of equal distance from the source vertex, and those vertices are currently active in the graph. The shortest paths have already been determined for the vertices that the wave has passed, and cost-path information has not yet activated those vertices that are more distant from the source vertex than the vertices on the wavefront are. When the source is placed at one corner of the grid, the wavefront is expected to advance as in Fig. 6.

Under these assumptions, the execution time and speedup for solving the problem using $p = q^2$ processors can be estimated as follows. As we divide the problem into the same number of subproblems as the number of processors, the number of subproblems becomes also q^2 in all. Let T be the time required for one processor to create and consume the cost-path information for all the vertices in one shaded triangle in Fig. 7. Then, as the time required for one processor to solve the entire problem is proportional to the total number of the vertices in the graph, it takes $2Tp$ time when only one processor is used. The time required to solve the problem with p processors is $2Tq$. Therefore, the speedup is

$$\frac{2Tp}{2Tq} = \frac{q^2}{q} = q = \sqrt{p}.$$

This means processor utilization rate is $1/\sqrt{p}$, which is poor and becomes poorer as the number of processors increases. In the real performance figures, the speedup increases a little less than two fold as the number of processor increases four-fold.

We took measurements of the processor utilization rate and communication overhead for each mapping strategy when the number of the processors is 16 (Fig. 11). The height of each vertical bar represents the processor utilization rate ($(busy\_time/total\_execution\_time) \times 100$), the black part represents the communication overhead (time spent in message handling routines[5]). The average processor utilization rate of 24% for this mapping is close to the expected work rate of 25% ($= 1/\sqrt{16}$).

6.2.2 Two-Dimensional Multiple Mapping

In the two-dimensional multiple mapping, each processor works when the wave of the creation of cost-path information passes through on the k subsets of vertices dispersed on the graph. As a result, the idle time of each processor was reduced as we had expected.

---

typical case of speculative computation.

[4] On the two-dimensional plane, the Manhattan distance is defined by $d(x, y) = |y_1 - x_1| + |y_2 - x_2|$, for $x = (x_1, x_2)$, $y = (y_1, y_2)$. A Manhattan path is a path whose length is equal to the Manhattan distance of the two vertices at both ends. Note there can be many Manhattan paths that connect two given vertices.

[5] Precisely speaking, message handling represents a significant part (expected to be more than 90%) of but is not equal to the total communication overhead.

In Fig. 9 and 10, we can see that much better processor utilization rate is obtained with the two-dimensional multiple mapping than with the two-dimensional simple, in all cases of the number of processors used at the execution. In fact, the processor utilization rate was 80% with $k = 4 \times 4$ and 94.4% with $k = 8 \times 8$ in the case of 16 processors (Fig. 11). However, the execution time does not improve as dramatically. This is due to much increased speculative computation and communication overhead as discussed later.

6.2.3 One-Dimensional Simple Mapping

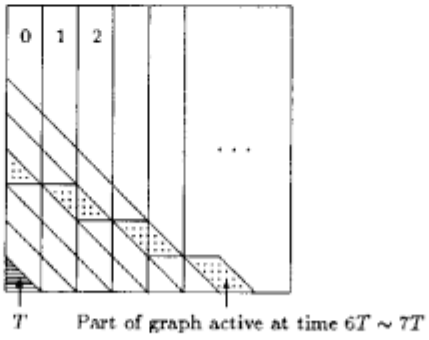


Figure 8: Expected Advance of Wavefront in One-Dimensional Simple Mapping

In Fig. 9 and 10, we can see that with this mapping, we could obtain similar same performances to the two-dimensional multiple mapping.

We will discuss the speedup for this mapping under the same assumptions as we discussed in the two-dimensional simple mapping. Let T be the time required for one processor to create and consume the cost-path information for all the vertices in one shaded triangle in Fig. 8. Then it takes $2T$ time to create and consume the additional information to the line in the figure. It follows that the time required to solve the problem with p processors is $T(3p-1)$. As it takes $2Tp^2$ time for one processor to solve the entire problem, the speedup is

$$\frac{2Tp^2}{T(3p-1)} = \frac{2p^2}{3p-1} \approx \frac{2}{3} \cdot p \text{ (when } p >> 1).$$

It says that the expected speedup is about two thirds the number of processors, in particular, the speedup is proportional to p, not the $\sqrt{p}$ as in the two-dimensional simple mapping.

The actual speedups are worse than we discussed above. It was 1.97 when 4 processors were used. But 5.7 with 16 processors, 11.85 with 36 processors, 17.34 with 64 processors, they become worse as the number of processors increases.

The average processor utilization rate of 45% for this mapping with 16 processors is also worse than the expected work rate of 68% ($= 2 \cdot 16/(3 \cdot 16 - 1)$) (Fig. 11).

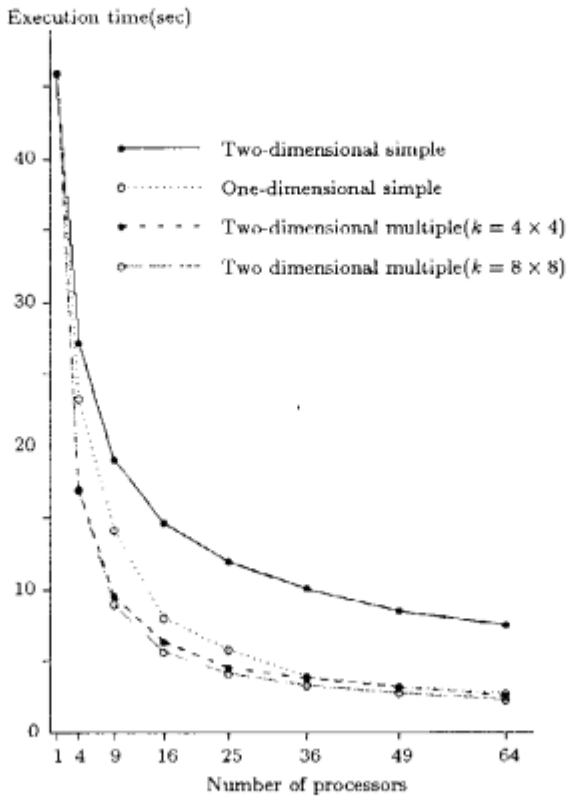


Figure 9: The execution time for various mappings and number of processors

7 Discussion

We would like to consider communication overhead and speculative computation that are ignored in the analysis above.

When the grid is divided into blocks for mapping, inter-processor communication arises at the boundaries of the blocks. As the grid is divided into smaller and smaller blocks, the total length of the block boundaries becomes longer and longer, incurring more and more inter-processor communication. The actual percentage of communication overhead depends on the size of the grid, the program, the underlying language implementation, and the particular multiprocessor used. But it is expected to be proportional to the total length of the block boundaries.

In Fig. 11, the ratio of communication time to non-communication time in the busy time is 5.3%, 19.2%, 34.1%, and 10.2% for two-dimensional simple, two-dimensional multiple (with 4×4 super blocks and 8×8 super blocks), and one-dimensional simple mappings. The results of dividing those numbers by the total boundary length of $6L$, $30L$, $62L$ and $15L$ respectively (L is the length of the side of the grid) are 8.8%, 6.4%, 5.5%, and 6.8% per L, which is fairly constant[6] as expected.

The amount of speculative computation was very difficult to predict. While all computation in

[6] The measurement error is 5% ~ 10%.

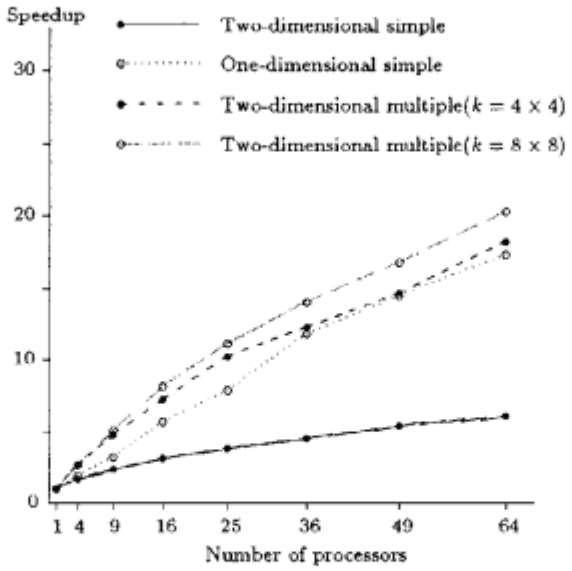


Figure 10: The speedup for various mappings and number of processors

Dijkstra's sequential algorithm is mandatory, all cost-path information processing in the distributed algorithm is speculative because one cost-path information can turn out to be useless if a better information later arrives. We compared the number of cost-path information packets generated in executions using various numbers of processors and mappings against that in a single processor execution. Fig. 12 shows them. The excess part represents the amount of cost-path information that have turned out useless. It was 20.2% in two-dimensional simple, 39.1% in two-dimensional multiple with 4×4 superblocks, 20.2% in two-dimensional multiple with 8×8 superblocks, 17.2% in one-dimensional simple. The excess is less in the 8×8 division than that in the 4×4 division. This apparent anomaly seems to come from the regular repeating patterns in the edge cost distribution, due to the nature of the pseudo-random number generator used, but this has not yet been confirmed.

To extend our experience to more general cases, we have to consider (1) different types of locally message exchanging algorithms, (2) different types of graphs, and (3) different types of networks of the multiprocessor.

The reason why one-dimensional simple mapping worked fairly well was due to the pattern of computation in the shortest path algorithm. If the computational load were uneven along the horizontal direction in the graph, the performance would have been worse. Two-dimensional multiple is expected to perform better in general cases. A more random mapping can be also considered, in which the graph is divided into $k \times p$ blocks, and each one of p processors is in charge of randomly chosen k blocks spatially scattered in the graph.

The topology of the graph and that of the multiprocessor network must be considered as a pair:

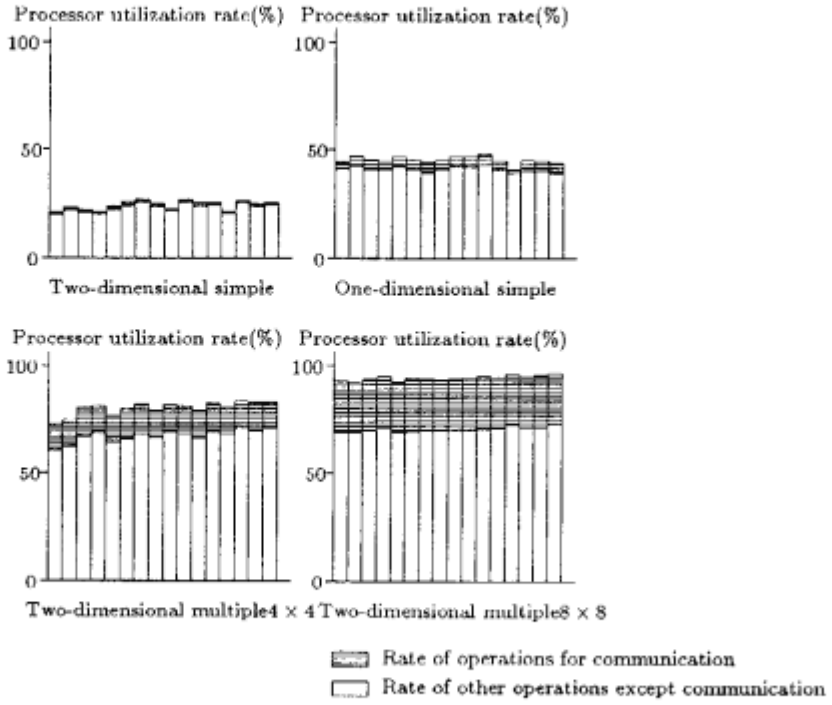


Figure 11: The processor utilization rate for various mappings with 16 processors

domain and range of mapping. In our experiment, the network traffic was very small compared with the bandwidth of network hardware. Thus, the topology of the target network was not so important. The two reasons were: (1) the overhead in interprocessor communication was mainly in encoding and decoding of message packets, and (2) most of the interprocessor message sending was from one processor to an adjacent processor (average traveling distance was close to one edge of the network).

Locality is a relative concept, relative to the target machine. In a sense, a tightly-coupled multiprocessor is one in which *everything* is local. Load balancing is easy, since communication suppression need not be considered. This ideal situation, however, does not scale. Loosely-coupled multiprocessors buy scalability at the cost of lost locality. As larger and larger scale multiprocessors are built, the program mapping will increasingly have to take locality preservation in consideration.

8 Conclusions

Good load balance is the key to get the most from the processing power of parallel computers.

Dynamic load balancing techniques have been developed and proved successful for tightly-coupled multiprocessors, and for loosely-coupled multiprocessors when the problem is composed of independent tasks with large granularity. But the problem of mapping inter-communicating processes on loosely-coupled multiprocessors has not been well explored. In this paper, we con-

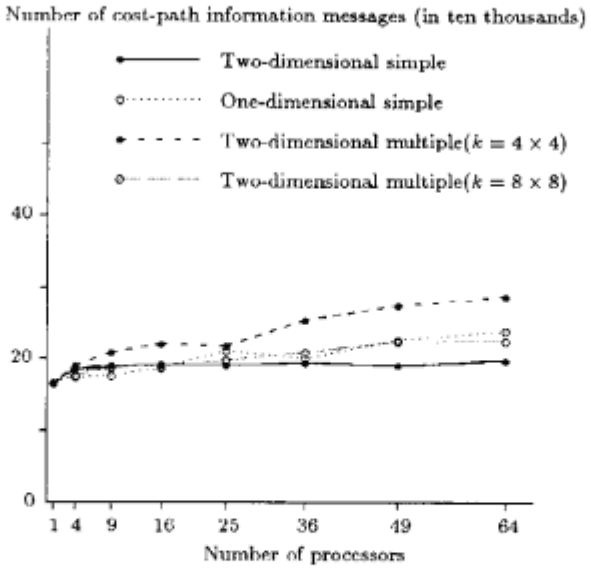


Figure 12: Number of cost-path information messages for various mappings and number of processors

sidered the problem when there is locality in message communication (*locally message exchanging algorithms*). Distributed graph algorithms generally fall into this category. As an example, we developed a distributed algorithm for a single-source shortest path problem for a large grid graph on a mesh-connected multiprocessor, Multi-PSI.

We experimented with three different static mappings: two-dimensional simple mapping which has the least communication overhead of the three. two-dimensional multiple mapping which has higher processor utilization rate at the cost of higher communication overhead, and one-dimensional simple mapping which has less communication overhead than the two-dimensional multiple but with less even load distribution than the two-dimensional multiple mapping. Measurements were taken of a shortest path program with the three mapping, with varying number of processors, and the actual speedups and communication overhead were discussed.

The experiments showed that two-dimensional multiple mapping and one-dimensional simple mapping attain a good compromise between communication localization and load balance. Unfortunately, the impact of ways of mapping on the amount of speculative computation is left as an open question.

The Multi-PSI proved a useful tool for doing parallel programming research — we could run a medium-scale program and obtain results that scale. The separation of semantics and mapping in KL1 greatly facilitated the process of experimenting load distribution and priority scheduling.

We would like to conduct further research on this topic, using different types of graphs and other types of locally message exchanging algorithms.

9 Acknowledgments

We would like to express our gratitude to Takashi Chikayama for giving us the idea of using the priority mechanism of KL1 to schedule messages, and Kazuo Taki for frequent, stimulating, and helpful suggestions and guidance. We thank Katsuto Nakajima, a former member of ICOT and now back in Mitsubishi Electric, who helped us gather measurements of processor work rate and communication overhead, and Kazuaki Rokusawa, now back in Oki Electric, for helpful suggestions and observations, and other members of ICOT who made valuable suggestions.

We would also like to thank Kazuhiro Fuchi, the director of ICOT and Shun'ichi Uchida, the manager of ICOT's Research Department, for giving us the opportunity to participate in this research.

References

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman. The Design and Analysis of Computer Algorithms, Addison-Wesley (1974).

[2] K. M. Chandy and J. Misra. Distributed Computation on Graphs: Shortest Path Algorithms. *Comm. ACM*, Vol.25, No.11 (Nov.1982), pp. 833-837.

[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* (1988), pp. 230-251.

[4] N. Deo, C. Y. Pang and P. E. Lord. Two Parallel Algorithms for Shortest Path Problems. In *Proceedings of the 1980 International Conference on Parallel Processing*. IEEE, New York, 244-153, 1980.

[5] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. Numerische Mathematik 1 (1959), 269-271.

[6] R. W. Floyd. Algorithm 97: Shortest Path. *Comm.ACM*, Vol.5, No.6 (1962), p. 345.

[7] M. Furuichi, K. Taki and N. Ichiyoshi. A Multi-Level Load Balancing Scheme for OR-Parallel Exhaustive Search Programs on the Multi-PSI. To appear in *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 1990.

[8] D. B. Johnson. Efficient Algorithms for Shortest Paths in Sparse Networks. *J. ACM*, Vol.24, No.1, pp. 1-13.

[9] V. Kumar and V. Nageshwara Rao. Load Balancing on the Hypercube Architecture. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, 1989.

[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* (1989), pp. 436–451.

[11] M. J. Quinn and N. Deo. Parallel Graph Algorithms. *ACM Computing Surveys*, Vol.16, No.3 (Sept.1984), pp. 319-348.

[12] K. Taki. The Parallel Software Research and Development Tool: Multi-PSI system. Programming of Future Generation Computers, Elsevier Science Publishers B.V. (North-Holland), 1988.

[13] K. Ueda. Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard. Technical Report TR-208, ICOT, 1986.

[14] S. Warshall. A Theorem on Boolean Matrices. *J.ACM*, Vol.9, No.1 (1962), pp. 11-12.