Chapter 7

The Implementation Scheme and Example Programs

The accuracy of the performance predictions made by analyser for a hypercube topology is studied in this chapter. Three programs are expressed in terms of the recognised functions and input to the analyser. For each program, the analyser determines a cost-effective parallel implementation, with the corresponding code being generated and executed on a hypercube network. The computation times are measured and compared with the predictions of the analyser. As mentioned in Section 3.5.4, a fully-fledged parallel code generator has not been implemented. However, support is available in the form of a library of functions for developing and testing programs. This chapter describes the environment for program development and discusses the results of implementing the three example programs.

The implementations are performed on the Meiko Computing Surface, which is a transputer-based distributed-memory parallel machine. Each transputer has four communication links and represents a single node on the binary hypercube. It is ensured that *logical* neighbours on the hypercube are also *physical* neighbours on the transputer network, which is assumed to be the case in the analyser's cost model. Therefore with only four physical links, it is not possible to simulate a hypercube of dimension greater than four. The communication harness, CSTools

[Mei], does allow for more than four neighbours per processor, but some *logical* neighbours will no longer be *physically* only a hop away. Worse still, it is not possible to predict the number of hops involved in the routing of a message, and indeed this value may vary for different executions of the program. This in turn would make it almost impossible to accurately predict the cost of such communications. Consequently, all the experiments have been performed on a hypercube with a maximum of 16 processors.

7.1 The Analyser

The algorithm for the analyser, as given in Section 5.2.1, has been implemented as a *C* program. The input is an 8-tuple, as described in Section 5.1. The analyser in its current form lacks type-inference and profiling capabilities; therefore, the types of functions and data, and estimates of data sizes have been explicitly provided. The cost expressions for the recognised functions have been coded into the analyser, which constructs the program tree for the application and computes the costs for the different possible implementations. The cost computation builds a search tree from which the least-cost path is selected. The path comprises of a set of nodes, one for each phase in the program. Each node contains the necessary information for its implementation: the number of processors for a particular phase, the nature of the current list distribution (as a result of the previous phase), the type of list distribution required, and which of the recognised functions in the phase are to be implemented in parallel and on how many processors. A communication routine computes the costs associated with inter-phase data rearrangements. It also keeps track of the nature of the input list distribution at any point in the execution of the program. The information contained in the set of nodes returned by the analyser should enable the automatic generation of code for a target parallel machine.

7.2 The Parallel Library

In the prototype implementation, the code for each call to a recognised function is generated manually. Some support is available in the form of a library of functions, which contains the code for the recognised functions on the hypercube topology. This enables the performance figures for the different test examples to be compared on a uniform basis. The code for the parallel implementation of the recognised functions is written in *C*, with CSTools [Mei] being used as the communication harness.

The support available to develop and test programs on the Meiko Computing Surface comprises of the following:

- A *definitions* guide which contains the *C* prototypes for the recognised functions and the various communication routines, and the list data structure on which the recognised functions operate.
- A library of routines containing the code for the implementation of the recognised functions on the hypercube.
- A library of routines which implements the definitions for the communication operations on the hypercube topology.
- A library of utilities containing list manipulation functions such as the creation and copying of lists.

Several optimisations have been adopted in order to reduce the cost of the parallel implementations. These are not program-specific, which ensures uniformity in performance comparisons. The analyser assumes that the following optimisations will be performed, which, in turn, is reflected in the cost estimates.

7.2.1 Memory Allocation

By its very nature, the list structure is dynamic. An implementation based on lists therefore involves a number of list creation and destruction operations, making it important to handle memory allocation efficiently. The implementation scheme allocates space for a list in a single operation, as opposed to allocating space individually for each list element, resulting in a significant reduction in memory allocation overheads. Also, since these costs are not accounted for by the analyser, it helps to bridge any disparity between the predicted and practical results. The current implementation does not incorporate a garbage collector.

This *one-step* memory allocation is possible because of the predefined behaviour of the recognised functions. For every recognised function applied to a list of a particular size, it is possible to compute the size of the resulting list (with the exception of the **filter** function, in which case the worst-case size of output list, i.e. the size of input list, can be assumed).

7.2.2 In-line Expansion of Function Calls

Function calls are expensive on the Computing Surface. Quite often, the cost of calling the function may be much larger than executing it. The model does not account for this cost which could lead to a disparity between the predicted and actual values. This, in turn, could lead to the selection of a less efficient implementation as the optimal one.

The problem could be resolved by designing the analyser such that it accounts for the costs of function calls. However, this introduces a machine-specific detail in the analyser. The problem is overcome by adopting an in-line expansion of the called function. A language such as *C++* would handle such expansions automatically. In its absence, these expansions are done manually.

A potential problem with in-line expansion would arise in the case of recursive functions. If a sequential function is recursively defined, then it has to be first converted into an equivalent iterative one before expansion.

7.2.3 Distribution of Lists among Processors

This is an optimisation that is made with a view to reducing the communication overhead. It involves the case where a list whose D-value ≥ 1 (i.e. a list of lists), is to be scattered across the processor network such that each sublist is distributed across p processors. Such a situation would arise when, in a branch with two nested recognised functions, the innermost one is to be implemented in parallel on p processors.

Let Xss represent a list of lists as follows.

$$\begin{aligned} Xss &= [Xs_1, Xs_2, \ldots, Xs_m] \\ Xs_i &= [x^i_1, x^i_2, \ldots, x^i_n] \\ & \quad 1 \leq i \leq m \end{aligned} \tag{7.1}$$

The size of Xss is $(m \times n)$. For the sake of simplicity in cost computations, it is assumed that $(n \bmod p = 0)$. The recognised function **split** can be used to divide the list, Xs_i, in Equation 7.1, into p parts which results in the following:

$$\begin{aligned} Xs_i &= [[x^i_1, x^i_2, \ldots, x^i_{\frac{n}{p}}] \\ & \quad [x^i_{\frac{n}{p}+1}, \ldots, x^i_{2\frac{n}{p}}] \\ & \quad \vdots \\ & \quad [x^i_{(n-1)\frac{n}{p}+1}, \ldots, x^i_n]] \\ &\equiv [Y^i_1, Y^i_2, \ldots, Y^i_p] \\ & \quad 1 \leq i \leq m \end{aligned}$$

In order to scatter each Xs_i across the p processors, the operation that needs to be performed on Xss is: **map** *scatter* Xss.

If d represents the dimension of the hypercube, from Equation 6.8 it can be deduced that the cost of distributing each sublist of the list of lists Xss is given by:

$$C_1 = m(dK_0 + \frac{1}{K_1}\frac{n}{p}(p-1)) \tag{7.2}$$

However, a more cost-effective distribution is possible and is discussed below. Consider the following definitions.

re_order p xss = **g_fold** *append* [ ] ∘ *shuffle* ∘ **map** (**split** p) xss

where,

shuffle ([ ]::ys) = [ ]

shuffle yss = (**map** *head* yss) :: *shuffle* (**map** *tail* yss)

The functions *head* and *tail* obtain the head and tail of a list, respectively. Now consider the function *re_order*, as applied to the list Xss defined in Equation 7.1.

If the result of the first stage of the **composition**, viz. (**map** (**split** p) Xss), is represented by Yss, then

$$\begin{aligned}
Yss &= [[Y_1^1, Y_2^1, \ldots, Y_p^1], \\
&\quad [Y_1^2, Y_2^2, \ldots, Y_p^2], \\
&\quad \vdots \\
&\quad [Y_1^m, Y_2^m, \ldots, Y_p^m]] \\
Y_i^j &= [x^j_{(i-1)\frac{n}{p}+1}, x^j_{(i-1)\frac{n}{p}+2}, \ldots, x^j_{i\frac{n}{p}}] \\
&\quad 1 \le j \le m \\
&\quad 1 \le i \le p
\end{aligned} \tag{7.3}$$

The size of Yss is given by $(m \times p \times \frac{n}{p})$. Next, the second stage of the composition is applied to the result of the first stage. Let Zss = shuffle Yss.

$$\begin{aligned} Zss &= [[Y_1^1, Y_1^2, \ldots, Y_1^m], \\ &\quad [Y_2^1, Y_2^2, \ldots, Y_2^m], \\ &\quad \vdots \\ &\quad [Y_p^1, Y_p^2, \ldots, Y_p^m]] \end{aligned} \tag{7.4}$$

The function *shuffle* effectively produces the transpose of its input list. The size of Zss is given by $(p \times m \times \frac{n}{p})$. Applying the final stage of the composition, viz. **g_fold** *append* [ ], to flatten the list, the result of *re_order* is as follows:

$$Rss = [Y_1^1, Y_1^2, \ldots Y_1^m, \ldots, Y_p^1, Y_p^2, \ldots, Y_p^m]. \tag{7.5}$$

The size of Rss is given by $(mp \times \frac{n}{p})$. Once the list Xss is transformed into a form as given by Rss in Equation 7.5, a *scatter* operation on Rss would place the elements on the correct processors. This method of distributing the list amounts to performing the following operation:

$scatter \circ (re\_order\ p)\ Xss.$

A *scatter* operation on a list of size $(a \times b)$ would involve the distribution of $a(b+1)$ list nodes. (For each sublist, there are b nodes representing each element of the sublist and one node that represents the entire sublist. There are a such sublists, hence the total number of nodes is $a(b+1)$). Since the size of Rss is $(mp \times \frac{n}{p})$, the total number of list nodes to be scattered is $(mp(\frac{n}{p}+1))$.

Again, referring to Equation 6.8, the cost of the entire list distribution operation is deduced to be:

$$C_2 = dK_0 + \frac{1}{K_1}\frac{mp(\frac{n}{p}+1)}{p}(p-1) + C_{re\_order}. \tag{7.6}$$

C_{re_order} represents the cost of the function *re_order* and essentially comprises of list processing costs.

Consider the following equation (Equation 7.2 - Equation 7.6):

$$C = dK_0(m-1) - \frac{1}{K_1}(p-1)m - C_{re\_order}.$$

$C > 0$ implies that:

$$1 + dK_0K_1\frac{m-1}{m} - \frac{K_1}{m}C_{re\_order} > p. \tag{7.7}$$

The cost of the *re_order* function could be computed by the analyser from its estimates of the list processing costs and the sizes of the input and output lists. If the left-hand-side of Equation 7.7 is greater than p, then the second option could be selected for distributing the input list across the p processors.

In practice, the costs of list processing will probably be much less than the communication costs and the second option can be safely selected as the more cost-effective one. As an example, the cost of processing a single list element is approximately $\frac{1}{100}K_0$ on the Computing Surface (by experiment). In the current implementation of the analyser, the second method of list distribution is assumed by the analyser in computing data distribution costs.

7.2.4 Parallel Implementations of Recognised Functions Involving Communication

This optimisation is used in the implementation of those recognised functions that occur in situations satisfying the following conditions:

1. The function is an argument of the recognised function **map**
2. The function is being implemented in parallel on p processors
3. The computation of the function in parallel on the p processors involves communication.

Some possible examples of such functions are **fold**, **scan** and **copy**.

Let $\mathcal{R}$ be a recognised function that occurs in such a situation. Typically, the distributed implementation for $\mathcal{R}$, represented by $\mathcal{R}_{dis}$, can be defined in the following manner.

$\mathcal{R}_{dis} = \mathcal{U} \circ \mathcal{C} \circ \mathcal{R}_{seq}$
where,

$\mathcal{R}_{seq}$ is the function describing the sequential implementation for $\mathcal{R}$

$\mathcal{C}$ is a communication function that involves the communication of partial results to neighbouring processors

$\mathcal{U}$ is a function that updates a processor's local partial result with the one that is received from a neighbour

According to condition 1, $\mathcal{R}$ must be an argument of **map**. Consider the following equation:

$$\text{map } \mathcal{R}_{dis} \equiv \text{map } (\mathcal{U} \circ \mathcal{C} \circ \mathcal{R}_{seq}). \tag{7.8}$$

Using the **map**-promotion rule, 7.8 can be expanded to:

$$\text{map } \mathcal{R}_{dis} \equiv \text{map } \mathcal{U} \circ \text{map } \mathcal{C} \circ \text{map } \mathcal{R}_{seq}. \tag{7.9}$$

The expression in Equation 7.9 has less communication overhead associated with it, in comparison to the one in Equation7.8. The implementation corresponding to Equation 7.8 involves the application of the sequential definition of the function, followed by the communication of the partial result for *each* element of the list in sequence. This implies that the communication start-up cost is multiplied by the length of the list. Also, the entire bandwidth of the channel will probably not be utilised, since only one list element is communicated at each step. In Equation 7.9, the sequential definition of the function is first applied to all the elements of the list resident on the particular processor. The resulting partial values are then communicated in one step. The communication start-up cost is incurred only once and the bandwidth of the communication channel is used more effectively. Both of these factors lead to a significant reduction in the communication cost if the implementation corresponding to Equation 7.9 is chosen.

7.3 Communication on the Hypercube

The algorithms for the different types of communication on the hypercube topology, as described in [SS89], have been implemented on the Meiko Computing Surface. In the communication of list data structures between processors, the one that *sends* the list must convert pointer addresses to relative ones and the one that *receives* the list must compute the real addresses from the offsets. These operations are necessary to obtain correct pointer references across processor boundaries. This effectively increases the communication overhead and is incurred in the form of list processing costs. The analyser accounts for these costs in the same way as it does for other list processing overheads, since the data to be communicated at any step in the program is predetermined.

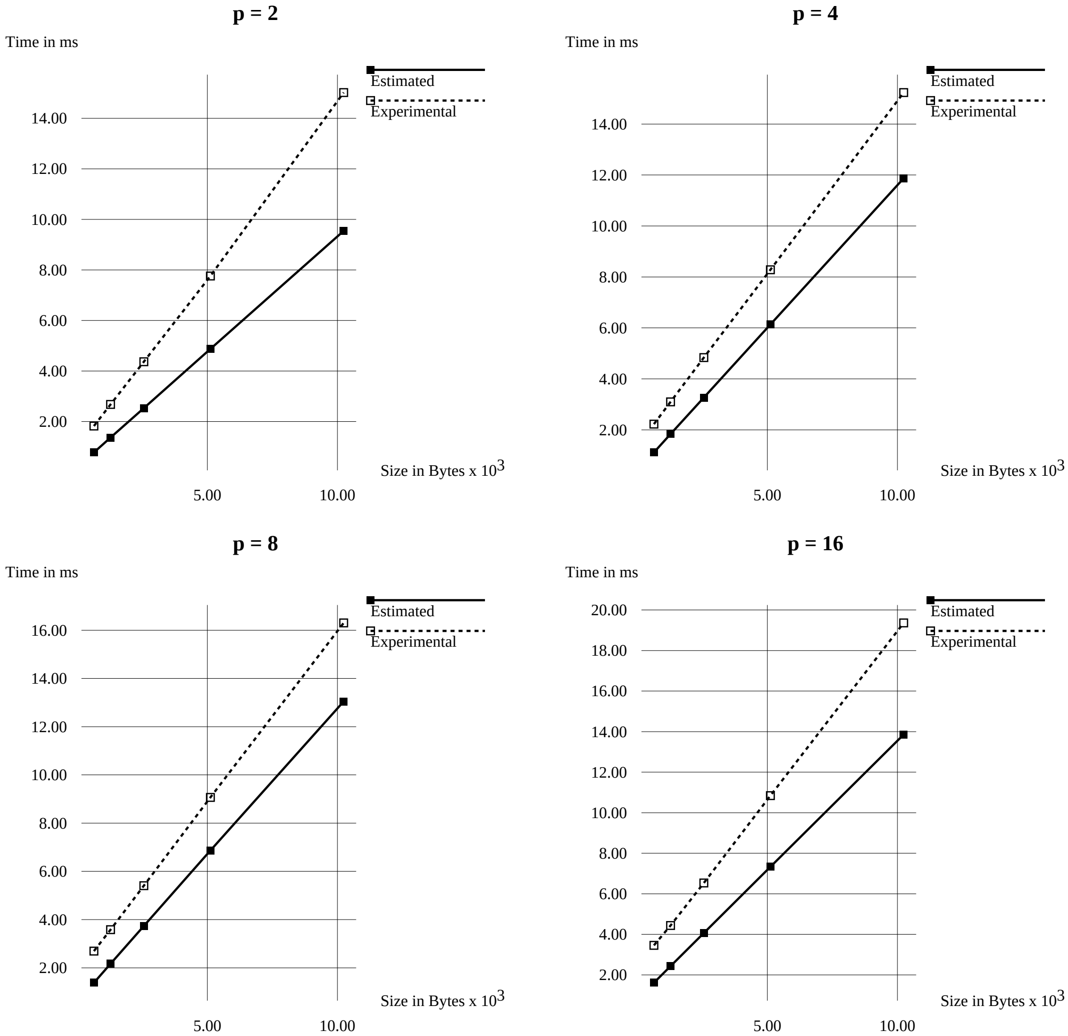
Graphs depicting the predicted and experimental results for some of the communication routines (ones which are used in the examples) on the hypercube topology are shown in Figures 7.1 and 7.2, respectively. The following points regarding the method used to obtain the graphs may be noted.

- Each processor executes an *initialisation* routine in which it opens its communication links and identifies its neighbours. Due to the lack of global synchronisation on a distributed-memory machine like the Computing Surface, it is unlikely that all the processors would operate in lock-step. This means that all the processors may not complete their initialisation and start on their communication routines at the same time. This implies that data transfers which are assumed to overlap, may not do so in a practical implementation. Consequently, the predicted communication costs may not always be obtained experimentally. In order to minimise this discrepancy, the experimental costs of communication routines are averaged over a large number of runs.

- The *scatter* operation is the dual of the *gather* operation. i.e. the *gather* algorithm is obtained by reversing the steps of the *scatter* algorithm. The costs of the two routines are, therefore, identical and have been plotted as a single graph. They are determined by performing a number of *scatter-gather* operations and halving the average cost. In the case of the *broadcast* operation, the processor that receives the data last, performs a broadcast in the reverse direction, so that the original broadcaster is the last one to receive it. This introduces a form of synchronisation and the cost of the *broadcast* operation is half the cost, averaged over a number of runs.

A study of the graphs reveals that the experimental curves follow closely with the theoretical predictions. However, as the size of the data to be communicated increases, the experimental curves seem to diverge from the predicted ones. A possible explanation for this feature may lie in the manner in which the Computing Surface handles communication. Beyond a particular size, the data may have to be divided into smaller-sized chunks and transferred in more than one step. This would increase the communication cost, which is not predicted by the model. Also, with increase in the number of processors, the match between the predicted and observed behaviour appears to improve, more so in the broadcast routine. A similar feature is to be expected in programs using these routines.

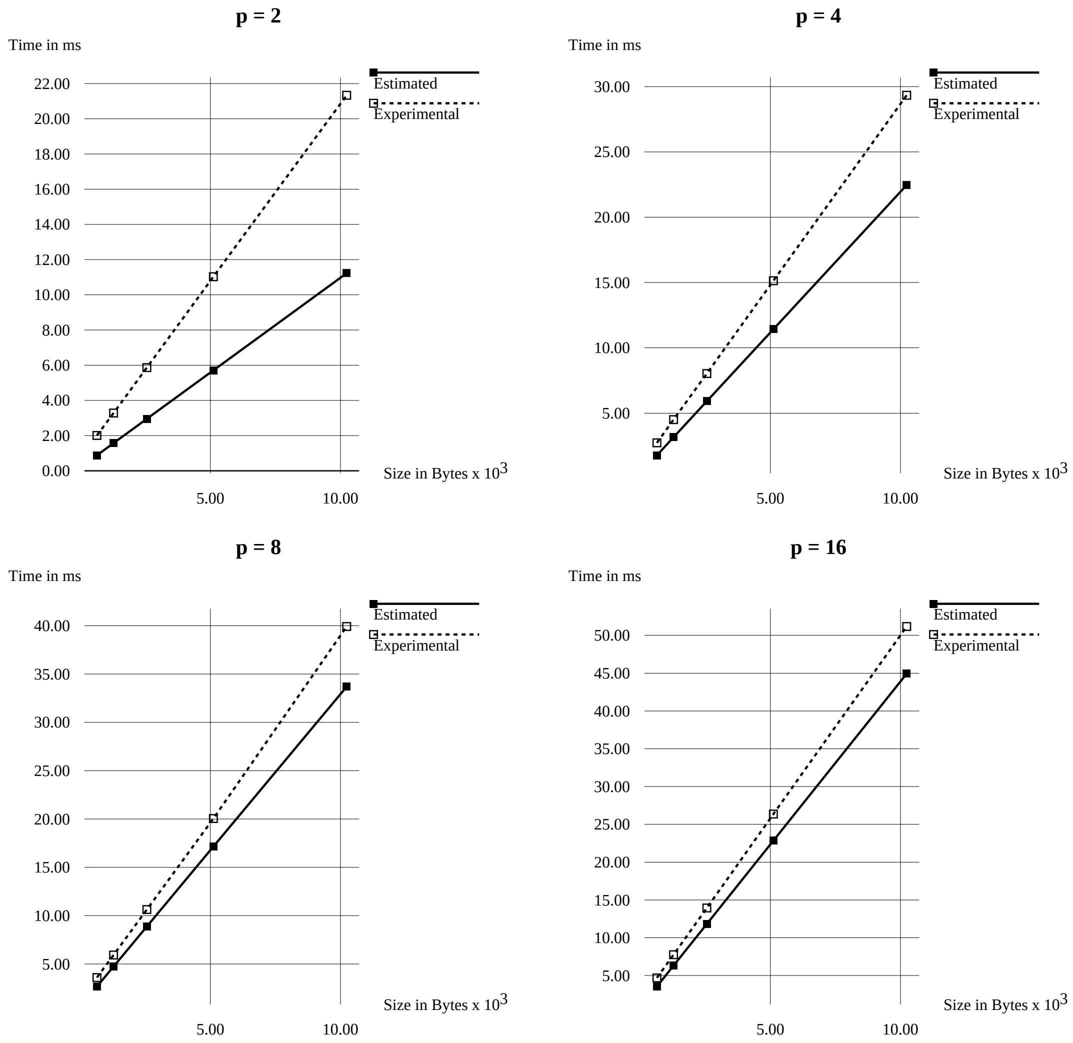
Figure 7.1: Scatter/Gather on a Hypercube



7.4 Example Programs

The three programs chosen for implementation are *Matrix Multiplication*, *Merge Sort*, and *Jacobi Iteration*, as discussed in Chapter 4. They are well-known problems in parallel programming and illustrate the use of different recognised functions in the HOPP model. The chosen programs are also intended to highlight different features of the analyser. The program for Matrix Multiplication is the straightforward case where the costs of all sequential functions are independent of input size, and where the analyser is capable of making correct inferences

Figure 7.2: Broadcast on a Hypercube



regarding problem size. The program for Merge Sort tests the performance of the analyser in the case where the cost of a sequential function (*merge*) is not a constant, but proportional to the sizes of its input lists. The program for Jacobi Iteration considers the performance of the analyser in a situation where it is incapable of making correct inferences regarding problem size. This, in turn, serves to demonstrate some of the limitations of the analyser in its current form.

For each of the three example programs, parallel implementations are considered and the corresponding costs as computed by the analyser are examined.

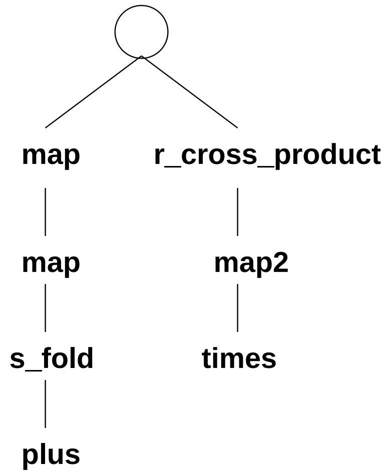


Figure 7.3: The Program Tree for Matrix Multiplication

The results of implementing the most efficient choice of the analyser on the Meiko Computing Surface are presented. The calls to the recognised functions and the communication routines are generated by hand, and the C-code in the parallel library is used. The *second-best* choice is also implemented, to add confidence in the performance of the analyser. The parallel implementations are considered on hypercubes with 2,4,8 and 16 processors, respectively.

The results of implementing the matrix multiplication and merge sort programs which are discussed next, can also be found in [Ran95].

7.4.1 Matrix Multiplication

This section discusses the well-known problem of multiplying two matrices $A_{m\times n}$ and $B_{n\times k}$, resulting in the matrix $C_{m\times k}$. The code for performing the multiplication is discussed in Section 4.3.1. Each matrix is represented as list of lists.

7.4.1.1 The Analysis

The program tree constructed by the analyser is shown in Figure 7.3. The root node represents **composition**. In *phase one*, the analyser computes the costs for a sequential implementation and three parallel implementations, corresponding to the two recognised functions in the phase, viz. **r_cross_product** and **map2**. On a p-processor hypercube, if **r_cross_product** is implemented on p_1^1 processors and **map2** is implemented on p_2^1 processors, ($p_1^1 \leq p$, $p_2^1 \leq p$ and $p_1^1 p_2^1 = p$), then the cost of *phase one* can be deduced as follows: (see Sections 6.2.6, 6.2.8)

$$C_{p1} = C_{com} + \lceil \frac{m}{p_1^1} \rceil \lceil \frac{n}{p_2^1} \rceil k C_{times}.$$

C_{com} represents the costs incurred in distributing the initial input lists across the p processors. C_{times} represents the cost of the sequential function *times*. It should be noted that the result of *phase one* is a list of lists of size $(m \times n \times k)$. *Phase two* contains three recognised functions. The analysis results in the costs for a sequential implementation and seven different parallel ones. If the outer **map** is implemented on p_1^2 processors, the inner **map** on p_2^2 processors and the **fold** is implemented on p_3^2 processors, respectively ($p_1^2 \leq p$, $p_2^2 \leq p$, $p_3^2 \leq p$ and $p_1^2 p_2^2 p_3^2 = p$), then the cost of *phase two* can be deduced to be as follows: (also refer to Sections 6.2.1, 6.2.2.2)

$$C_{p2} = R_{com} + \lceil \frac{m}{p_1^2} \rceil \lceil \frac{n}{p_2^2} \rceil C_{plus}(\lceil \frac{k}{p_3^2} \rceil - 1) + d(T_{com} + C_{plus}).$$

R_{com} represents the cost incurred in re-arranging the results of *phase one* to suit the implementation of *phase two*, ($d = \log_2 p_3^2$), T_{com} is the cost of communicating the partial result of **fold** on one processor to a neighbouring one.

Two sets of matrices, $A^1_{32\times32}$, $B^1_{32\times32}$ and $A^2_{64\times32}$, $B^2_{32\times16}$, are considered for analysis. In both the cases, the following implementations are selected as the *best* (i.e. least-cost) and *second-best* parallel implementations, respectively, for $p = 2, 4, 8, 16$.

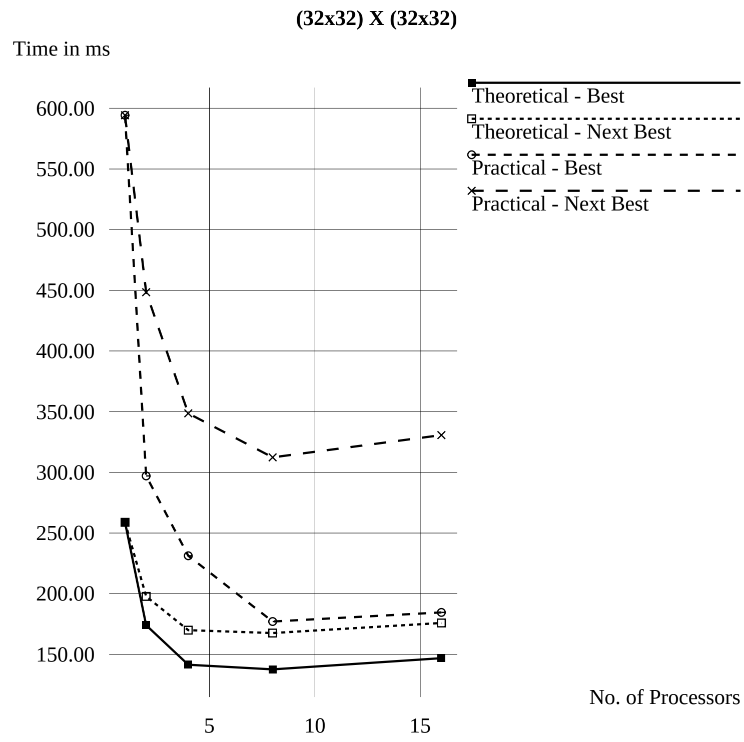
1. *best* - for the two phases, $p_1^1 = p$, $p_2^1 = 1$ and $p_1^2 = p$, $p_2^2 = 1$. This corresponds to parallel implementations for **r_cross_product** and the outer **map** in the first and second phases, respectively.

2. *second-best* - for $p = 2$, it is clear that only one function in a phase can be implemented in parallel. For the *second-best* implementation, **map2** and **fold** are implemented in parallel in the first and second phases, respectively. For $p = 4, 8, 16$, the following implementation was selected as the *second-best*. In *phase one*, $p_1^1 = 2$, $p_2^1 = \frac{p}{p_1^1}$. In *phase two*, $p_1^2 = 2$, $p_2^2 = 1$ and $p_3^2 = \frac{p}{p_1^2}$. This corresponds to a parallel implementation for **r_cross_product** on p_1^1 processors and **map2** on p_2^1 processors in the first phase. In the second phase, the outer **map** and the **fold** are implemented in parallel on p_1^2 and p_3^2 processors, respectively. The inner **map** is implemented sequentially.

7.4.1.2 <u>The Results</u>

The theoretically predicted values are plotted along with the experimental ones in Figure 7.4, for the two chosen implementations. In each of the graphs, the *trend* predicted by the analyser is also obtained experimentally. The experimental costs are greater than the predicted ones. This is because the analyser does not account for low-level costs which are incurred by a practical implementation. For both sets of matrices, the number of processors that is predicted to be the optimum, is confirmed experimentally. The experimental curves tend to have much steeper gradients compared to the predicted ones, especially in the case of smaller numbers of processors. This feature is less easy to explain. A probable explanation may lie in the observed behaviour of the communication routines (refer to Figure 7.1, Figure 7.2), where the match between predicted and observed behaviour improves with an increase in the number of processors.

It may be noted that the theoretical and experimental curves in implementation 1 are much closer than the corresponding ones in implementation 2. The sequential code that is executed in both the cases is identical. However, implementation 2 involves more communication than implementation 1, both for initial



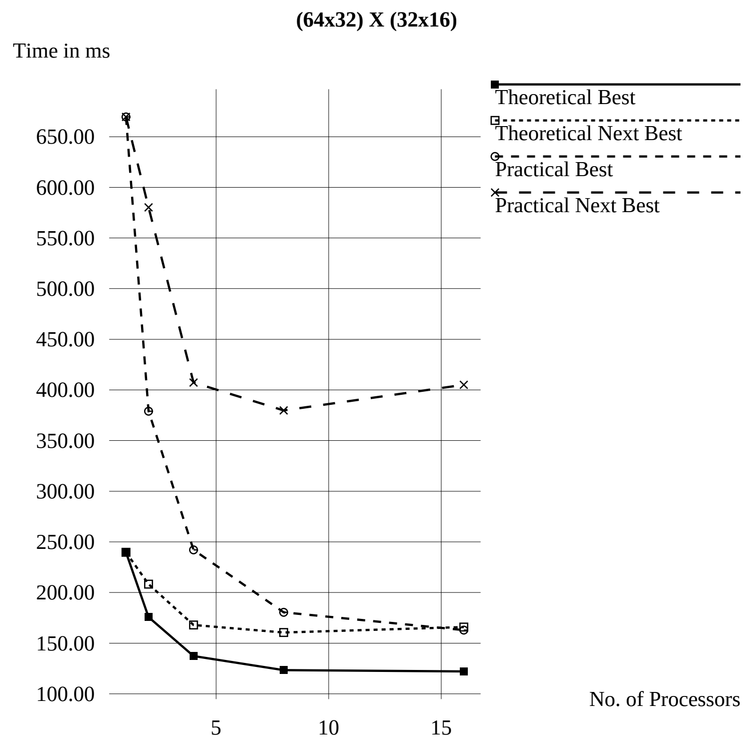


Figure 7.4: The Results for Matrix Multiplication

data distribution and for the parallel implementation of **s_fold** in *phase two*. Implementation 1 involves a *scatter* and a *broadcast* operation on matrices A and B, respectively, at the beginning of *phase one*. There is no communication in *phase two*. *Phase one* of implementation 2 involves a *scatter* followed by a *broadcast* operation on matrices A and B, respectively, to place them on p_1^1 processors for the parallel implementation of **r_cross_product**. Two further *scatter* operations are required to distribute each of these on p_2^1 processors, for the parallel implementation of **map2**. More communication is required in *phase two* for processors to communicate the partial results of **s_fold**, since it is implemented in parallel. There seems to be a greater discrepancy between the predicted and experimental results when an implementation involves more communication. This is in a large part due to the lack of global synchronisation between processors.

The costs for communication routines assumed overlapping data paths which may not always happen in practice. This effect gets more pronounced with increase in communication, with the processors getting more staggered and leading to a greater discrepancy between predicted and observed behaviour. Care was taken in the cases of the communication routines such as *scatter* and *broadcast*, to minimise these discrepancies by averaging the costs over several executions and forcing some synchronisation by performing the dual of the corresponding routine and considering only half the total cost. However, in the implementation of a program, the corresponding routine is executed only once and this leads to a staggered operation of processors which are otherwise assumed to operate in lock-step. The discrepancy observed in the performance of the individual communication routines is also probably a contributing factor.

7.4.2 Merge Sort

The code for the program is discussed in Section 4.3.2. For the sake of simplicity, the number of elements in the input list is assumed to be a power of 2 and is represented by n.

7.4.2.1 The Analysis

Each of the three phases have only one recognised function. Hence, only two implementations are possible for each phase - a parallel implementation and a sequential one. The cost of the function *msort* is a constant, but the cost of the function *merge* is proportional to the sizes of the two lists which are being merged. In this case, as discussed in Section 5.1.4, the scheme allows the user to specify a cost function for the sequential function, *merge*. The analyser uses this function to estimate implementation costs.

The analyser first constructs the program tree. For *phase one*, the cost of **split** is given by $S_{\frac{n}{p},\frac{n}{2p}}$, (refer to Section 6.2.10). The cost for the sequential implementation is obtained by setting p to 1. At the end of *phase one*, each processor contains a list of lists of size $(\frac{n}{2p} \times 2)$. *Phase two* only involves a **map** and its cost is given by $\frac{n}{2p}C_{ms}$, where C_{ms} represents the cost of the function *msort*. In *phase three*, after each *merge* operation, the size of the resulting list is the sum of the sizes of the two lists being merged. As the **fold** progresses, the size of the intermediate result increases. After each processor executes the **fold** locally, the number of elements on each processor is $\frac{n}{p}$. In the first step of the combination of partial results, two lists each of size $\frac{n}{p}$ are merged to produce a list of size $\frac{2n}{p}$. The second step involves the merging of two lists each of size $\frac{2n}{p}$ to produce a list of size $\frac{4n}{p}$, and so on. Let the cost of merging two lists of sizes m and n be represented by $f(m,n)$. Using the formula for the cost of **g_fold** in

Section 6.2.2.2, the cost of *phase three* is as follows :

$$\sum_{i=1}^{\frac{n}{2p}-1} f(2i,2) + \sum_{i=0}^{d-1}(T_{com}^{2^i \frac{n}{p}} + f(2^i \frac{n}{p}, 2^i \frac{n}{p})).$$

In order that the analyser accounts for the increasing size of the emerging result while computing communications costs, the user is required to specify the **fold** operation using **g_fold**.

Two lists of sizes 512 and 1024 elements are considered for analysis. In both the cases, the following implementations are selected as the *best* (i.e. least-cost) and *second-best* implementations respectively, for $p = 2, 4, 8, 16$.

1. *best* - implement each recognised function in parallel on p processors for all the phases.

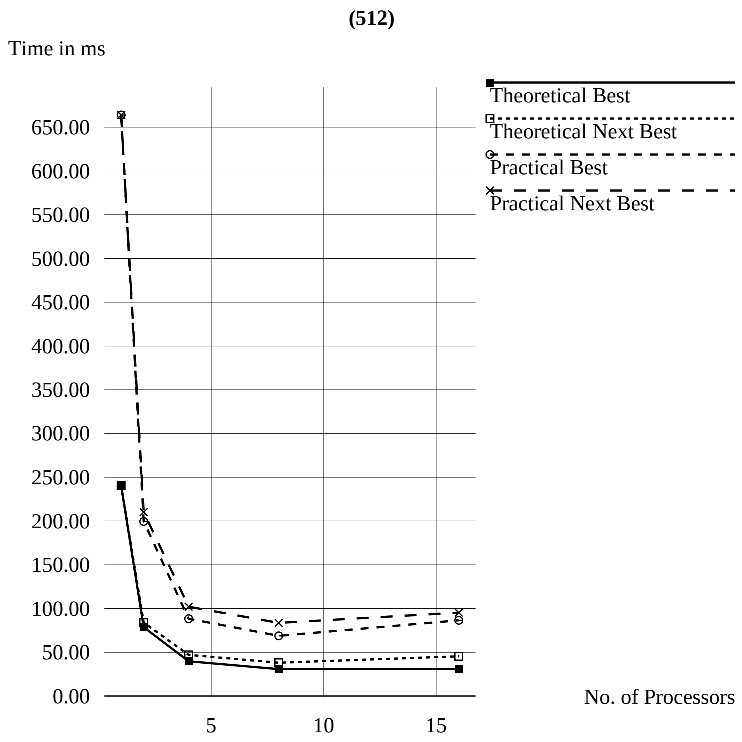
2. *second-best* - in *phase one* implement the function **split** sequentially. Then *scatter* the result to implement the two recognised functions in the next two phases in parallel on p processors.

7.4.2.2 The Results

The results for merge sort are plotted in Figure 7.5. Once again the *trend* predicted by the analyser is obtained experimentally.

For the input list of size 512, the optimal number of processors is predicted to be 8 for both implementations and this is confirmed experimentally. For the list of size 1024, the analyser predicted a more cost-effective implementation with 16 processors, which is again confirmed experimentally.

In the case of merge sort, fewer communications are involved in the initial distribution of data, as compared to both implementations of matrix multiplication. The initial data distribution comprises of a single *scatter* routine, either at the beginning of *phase one* or *phase two*, depending on whether the *best* or *second-best* implementations are considered. This probably causes a smaller stagger in the



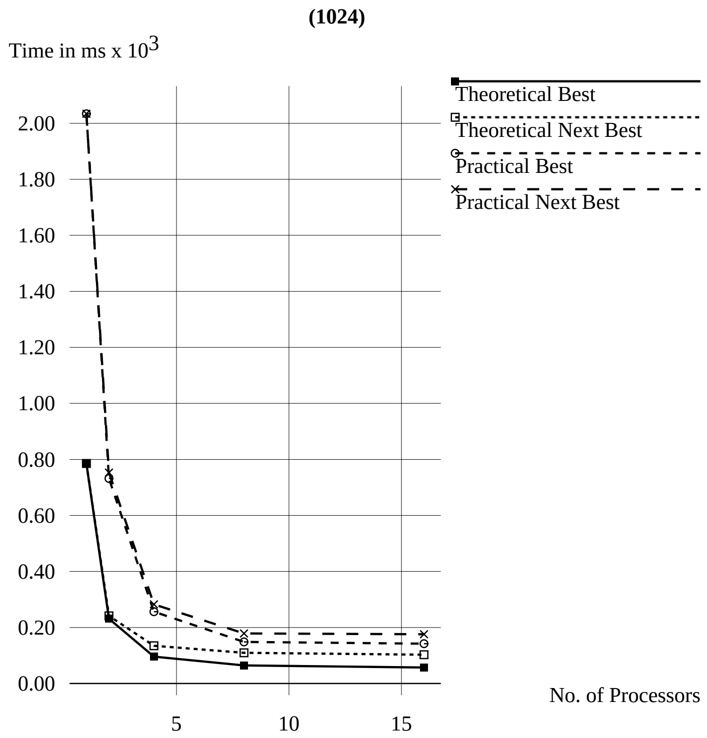


Figure 7.5: The Results for Merge Sort

processors. Also, the parallel implementation of **g_fold** in the last phase forces some form of synchronisation between the processors, since the partial results are communicated to processor$_0$, which computes the time of execution for the program. Both of these factors seem to produce a closer match between the predicted and experimental results in the case of merge sort. However, the gradients of the curves in a practical implementation are still steeper than those of the predicted curves and the same explanation as given in the case of matrix multiplication applies.

7.4.3 Jacobi Iteration

The code for the program has been discussed in Section 4.3.3. As explained in Section 5.1.4, the analyser in its current form is incapable of inferring that the function *rearrange*, which is the argument of **map** in *phase two* of the function *jac*, produces an output list whose size is different from the size of its input list. Consequently, the costs computed by the analyser for *phase three* will be inaccurate. Also, the function *update* in *phase three* changes the D-value of its argument list. This again, cannot be determined by the analyser. The change in D-value will not influence the cost of the function that tests for convergence. This is because the function *test* assumes that the output of the function that operates on the input list at each iteration, is of the same size as its input list. This assumption is justified because the analyser assumes that the cost of iteration$_i$ is the same as the cost of iteration$_{i-1}$. The inability of the analyser to detect the change in the D-value of the output of *update* will only influence the cost of the subsequent iteration. However, in the case of iterative functions, since the cost of a single iteration is considered for the purposes of determining a cost-effective parallel implementation, this will not lead to inaccuracies in cost predictions for this particular program.

7.4.3.1 The Analysis

The input matrix is represented by a list of lists of size $(m \times n)$. The analyser first constructs the program tree. The first and second phases have only one recognised function each, while *phase three* has two recognised functions. Consequently, there is only one possible parallel implementation for the first and second phases. For *phase three*, there are three possible parallel implementations. The costs for the first and second phases are straightforward, being the costs of **get_neigh** and **map**, respectively. Again, the cost of the function *rearrange* in *phase two*, is a function of the size of its input list and the analyser determines its cost from the specification of a cost function for *rearrange*. Consider the costs for *phase one* and *phase two*, represented by C_{p1} and C_{p2}, respectively. Referring to Equations 6.66 and 6.22, the costs are given by:

$$\begin{aligned} C_{p1} &= C_{com} + T^2_{com} 2d + G_{\frac{m}{p}} \\ C_{p2} &= R^1_{com} + \lceil \frac{m}{p} \rceil C_{rearr} \end{aligned} \qquad (7.10)$$

where, C_{com} is the communication cost incurred in scattering the input list across p processors, T^2_{com} represents the cost of communicating two sublists each of size n to a neighbouring processor, $d = \log_2 p$, C_{rearr} represents the cost function specifying the cost of the function *rearrange*, and R^1_{com} is the communication cost incurred in rearranging the output list of *phase one*, to suit the implementation of *phase two*. It should be noted that $R^1_{com} = 0$, if both *phase one* and *phase two* are implemented in parallel on p processors, since each of the phases has only one recognised function. The result of *phase one* is a list of size $(m \times 3 \times n)$. The function *rearrange* in *phase two*, transforms an input list of size $(3 \times n)$ into a list of size $(n \times 5)$. So, the output of *phase two* is a list of size $(m \times n \times 5)$. However, the analyser cannot deduce this change in the size of the output list and instead assumes it to still be $(m \times 3 \times n)$, since the recognised function in *phase two* is

map, which is not assumed to change the size of its input list. Hence, the cost computed by the analyser for *phase three* will be inaccurate. If the outer **map2** is implemented on p_1 processors and the inner **map2** on p_2 processors, $(p_1 p_2 = p)$, the cost of *phase three* is as follows: (refer to Section 6.2.8)

$$C_{p3} = R^2_{com} + \lceil \frac{m}{p_1} \rceil \lceil \frac{n}{p_2} \rceil C_{update} \tag{7.11}$$

where, R^2_{com} again represents the cost incurred in rearranging the output of *phase two* to suit the implementation of *phase three*. However, the analyser will compute a different cost (since the list size it deduces is different), and probably select an implementation that is not the most cost-effective one.

The function *test* (see Section 4.3.3) comprises of four phases, and its cost is the sum of the costs of these phases. The first three phases in the function *test* could be implemented in parallel, since each of them has instances of recognised functions. The cost of *test* has to be added to the costs C_{p1}, C_{p2} and C_{p3} to obtain the total cost of **iterate_cond**.

As mentioned in Section 6.2.9, the cost of only one iteration is considered for the purposes of determining a cost-effective parallel implementation. In an actual implementation, this is the average cost of an iteration.

7.4.3.2 The Results

Two lists of sizes (32×32) and (64×64), respectively, are considered for analysis. In each case, four costs are considered:

- The costs of the *best* and *second-best* implementations, as predicted by the analyser, which will not be accurate.

- The correct costs for the *best* and *second-best* implementations. These are computed manually, in order to determine the percentage error in the costs computed by the analyser.

In each case, the following implementations are selected as the *best* and *second-best* implementations, respectively.

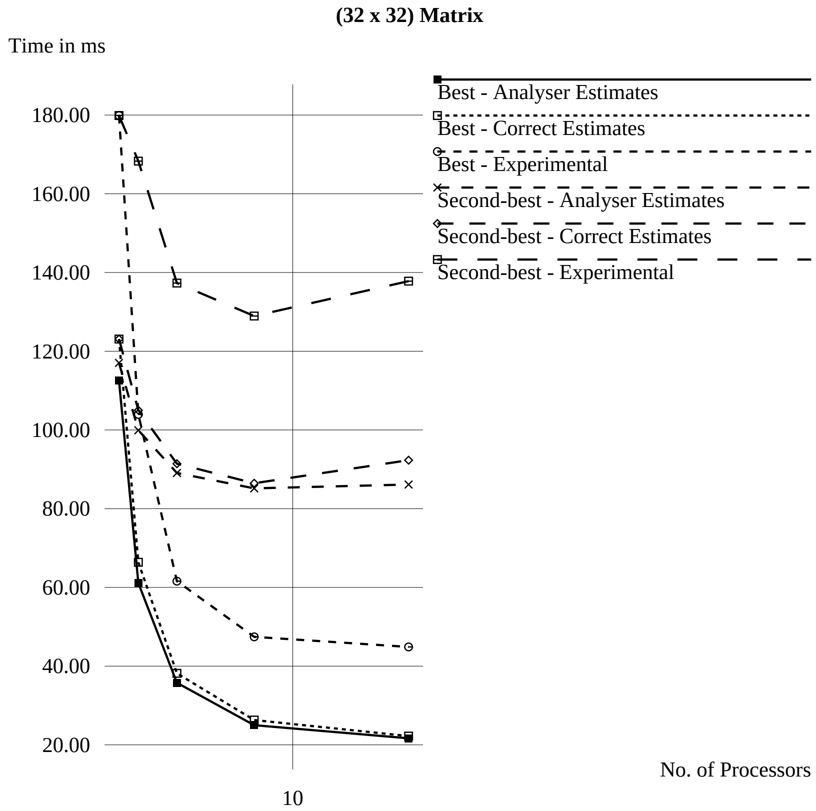
1. *best* - implement **get_neigh**, **map** and the outer **map2** in parallel on p processors for the three phases. Implement the first three phases of the function *test* in parallel, implementing the outer **map2** in *phase one*, and the **map** and **s_fold** in phases *two* and *three*, respectively, in parallel on p processors. The function *less_than* is implemented sequentially. Note that the result of **s_fold** in *phase three* of *test* would leave its result on processor$_0$ and the function *less_than* can then be implemented sequentially on processor$_0$.

2. *second-best* - implement **get_neigh**, **map** and the outer **map2** in parallel on p processors for the three phases. Then *gather* the results and implement the function *test* on a single processor (processor$_0$). As noted in Section 6.2.9, only a *copy* of the result is gathered back to processor$_0$ for evaluating the function *test*. Each processor still possesses its copy of the result to serve as the data for a possible next iteration.

It is to be noted here, that in both implementations the result of the function *test* is left on processor$_0$. Processor$_0$ must *broadcast* this result to all the other processors, so that they can determine whether the iteration is to be terminated. This *broadcast* is performed after every step of the iteration and the analyser includes its cost in the computation of the cost for the program. The cost of an iteration is, therefore, given by (refer to Equations 7.10, 7.11):

$$C_{jac} = C_{p1} + C_{p2} + C_{p3} + C_{test} + C_{broad} \tag{7.12}$$

where, C_{test} is the cost of the function *test*, and would depend on whether a sequential or parallel implementation is selected for it. C_{broad} is the cost of broadcasting the result of the function *test* to all the processors in the network.



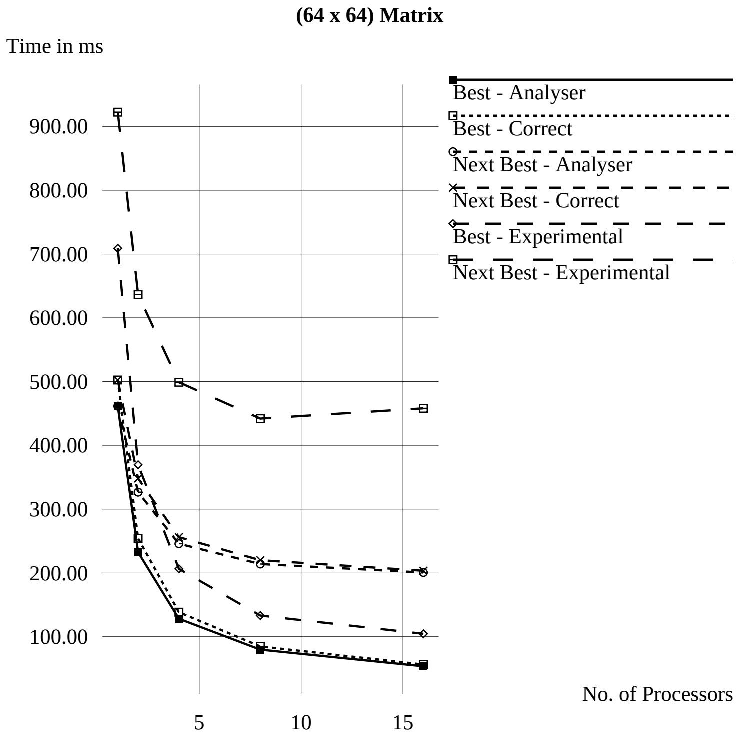


Figure 7.6: The Results for Jacobi Iteration

The three sets of costs for the *best* and *second-best* implementations are plotted in Figure 7.6. Again, the predicted *trend* is also obtained experimentally. In the case of the (32×32) matrix, for the *best* implementation a speed-up is obtained with 16 processors, but for the *second-best* implementation, the optimal number of processors is predicted to be 8, and this is also obtained experimentally. The analyser under-estimates the costs slightly, but still (in this case), the most cost-effective implementation is the one that is selected. (This has been verified by computing the costs manually). The similar feature of the gradient of the experimental curves being steeper than that of the predicted ones is also observed in this example. The distance between the predicted and experimental curves in the *second-best* implementation is larger than that between the corresponding curves in the *best* implementation. This again, is probably because the *second-best* implementation involves a *gather* operation before the function *test* can be implemented and this involves more communication than in the case of the *best* implementation. The distance between the corresponding curves in this example is much larger than that in the *Merge Sort* example. This again is probably due to the fact that the *Jacobi Iteration* example involves more communication than the *Merge Sort* example.

7.5 Conclusions

The performance of three programs expressed in terms of the recognised functions has been studied on a hypercube topology. The analyser computed the most cost-effective implementation for each of the programs. These, along with the *second-best* parallel implementations have been executed on the hypercube, and the predictions made by the analyser have been confirmed experimentally. The three programs are *regular* in the context of the HOPP model, and it is, therefore, possible to express them in terms of the existing constructs. It may not be possible to effectively parallelise irregular problems such as quicksort that depend on the actual input data values, by using this scheme. This is a limitation of the scheme and arises due to the use of compile-time techniques to predict program performance, as well as the restrictive nature of the recognised functions.