Array Combining Scatter Functions on Coarse-Grained, Distributed-Memory Parallel Machines

Seungjo Bae, Khaled A. Alsabti, and Sanjay Ranka

CRPC-TR97732-S
March 1997

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Submitted November 1997

Array Combining Scatter Functions on Coarse-Grained, Distributed-Memory Parallel Machines

Seungjo Bae,[*,1] Khaled A. Alsabti,[†,2] and Sanjay Ranka[†]

[*]Corresponding author

2–120 Center for Science and Technology

School of Computer and Information Science

Syracuse University

Syracuse, NY 13244-4100

Phone: (315) 469-9211

Email: *sbae@top.cis.syr.edu*

[†]CSE Building, #301

Department of Computer Science

University of Florida

Gainesville, FL 32611

Phone: (352) 392-1526

Email: *kalsabti, ranka@cis.ufl.edu*

March 1997

[1]Current address: Parallel Programming Section, Computer System Department, Electronics and Telecommunications Research Institute, Yusong P.O. Box 106, Taejon 305-600, Korea, Email: sbae@computer.etri.re.kr

[2]He is currently visiting from Syracuse University.

Array Combining Scatter Functions on Coarse-Grained, Distributed-Memory Parallel Machines

Seungjo Bae,* Khaled A. Alsabti,[†,1] and Sanjay Ranka[†]

*Corresponding author

Parallel Programming Section

Computer System Department, Computer Technology Division

Electronics and Telecommunications Research Institute

Yusong P.O. Box 106, Taejon 305-600, Korea

Phone: +82-42-860-5667

Email: *sbae@computer.etri.re.kr*

[†]CSE Building, #301

Department of Computer Science

University of Florida

Gainesville, FL 32611

Phone: (352) 392-1526

Email: *kalsabti, ranka@cis.ufl.edu*

[1]He is currently visiting from Syracuse University.

Abstract

In High Performance Fortran (HPF) *array combining scatter functions* are generalized array reduction functions. Elements in a subset of a source array are combined and the resulting value is sent to a position in a resulting array. Array combining scatter function can be defined in terms of *random access write* (RAW).

In this paper we present several local and global combining schemes and their impact on overall performance. First we present two global combining schemes for array combining scatter functions with arbitrary hot spots (processors) on coarse-grained, distributed-memory parallel machines: *direct (one-stage) algorithm* and *extended two-stage algorithm*. We also present four local combining (collision resolution) schemes, *whole local combining using open address hashing*, *whole local combining using direct address hashing*, *partial local combining without compression*, and *partial local combining with compression*, to reduce variance in message size. In each global combining scheme any one of the local combining schemes may be used or no local combining may be performed. Thus the above two global combining algorithms and four local combining schemes can result in ten schemes.

Our schemes for the array combining scatter functions are relatively architecture-independent and can be easily extended to arrays of arbitrary dimensions with arbitrary distribution along every dimension. Extensive experimental results are given on the CM-5. The performance evaluation presented in this paper can be used as a guideline for deciding which scheme can be effectively applied in applications requiring the array combining scatter primitive.

Key Words Array combining scatter functions, high performance fortran, random access write, hot spot, hot processor, direct algorithm, two-stage algorithm, global combining, local combining

Relevant technical areas Parallel algorithms and applications, compilers (runtime support)

1 Introduction

In High Performance Fortran (HPF) *array combining scatter functions* are generalized array reduction functions. Elements in a subset of a source array are combined, and then the resulting value is sent to a position in a resulting array. At this time each subset is non-overlapping [5, 8]. Twelve array combining scatter functions are defined in High Performance Fortran (HPF) and have the form `XXX_SCATTER(ARRAY, BASE, INDX`$_0$`, ..., INDX`$_{r'-1}$`, MASK)` where `XXX` is one of the reduction operations defined in HPF, and r' is the rank of a base array [5, 8]. Arguments are explained in Table 1. Each array combining scatter function returns a resulting array that is conformable with the base array and has the same type as the source array.

`ARRAY`	Source array of rank r
`MASK`	Logical mask array conformable with the source array
`BASE`	Base array of rank r', which has the same type as the source array
`INDX`$_i$	Index array conformable with the source array where $0 \leq i < r'$

Table 1: Description of arguments used in array combining scatter functions

The array combining scatter function can be defined in terms of *random access write* (RAW) [12]. In random access write (RAW) each element i $(0 \leq i < N)$ may need to write (or send) a datum, $D(i)$, to another element [12]. At this time $I(i)$ indicates a pointer such that element i needs to write a datum, $D(i)$, to an element, $I(i)$. Thus parallel execution of the do-loop,

$$do \;\; (i = 0 : n-1) \;\; A(I(i)) = D(i),$$

results in RAW [16]. Figure 1 shows an example of a RAW.

Assume that two elements i and j $(0 \leq i, j < N)$ need to write the data, $D(i)$ and $D(j)$, to the same target element (i.e., $I(i) = I(j)$). We use the term *collision* to refer to this situation. There are several methods that can be used to resolve the collision if the number of data elements causing a collision is bounded [16]. In this paper we assume collisions are resolved by using a predefined binary associative operator. In Figure 1 collisions are resolved using a binary associative operator $+$.

Element index	0	1	2	3	4	5	6	7
Pointer array	7	7	1	1	2	6	3	3
After RAW	–	$D(2)+D(3)$	$D(4)$	$D(6)+D(7)$	–	–	$D(5)$	$D(0)+D(1)$

Figure 1: Example of random access write (RAW)

We call a target element where a collision occurs a *hot spot* [16]. In Figure 1 hot spots are elements 1, 3, and 7. On coarse-grained, distributed-memory parallel machines all elements are distributed among processors, thus the presence of hot spots may cause a high variance in incoming traffic on target processors. Such hot spots are inherent features of applications such as histogramming, geometric hashing [14], the Hough transformation [20, 10], and database join [11].

In this paper we present two global combining algorithms for the array combining scatter functions on coarse-grained, distributed-memory parallel machines: *direct (one-stage) algorithm* and *extended two-stage algorithm*.

In the direct algorithm all source data are sent directly to target processors from source processors. However, in the two-stage algorithm they are sent via intermediate processors in order to prevent a high variance in incoming traffic due to the presence of hot spots [16]. Also, we present four local combining (collision resolution) schemes, *whole local combining using open address hashing*, *whole local combining using direct address hashing*, *partial local combining without compression*, and *partial local combining with compression*, to reduce outgoing traffic as well as variance in incoming traffic. In each global combining scheme any local combining schemes may be used or, clearly, none of the four local combining schemes may be used. Thus, the above two global combining algorithms and four local combining schemes can result in ten schemes. These ten schemes are summarized, along with their abbreviations used in this paper, in Table 2.

Abbr.	Scheme
NLC-OSC	Direct algorithm without local combining
O-WLC-OSC	Direct algorithm with whole local combining using open address hashing
D-WLC-OSC	Direct algorithm with whole local combining using direct address hashing
N-PLC-OSC	Direct algorithm with partial local combining without compression
C-PLC-OSC	Direct algorithm with partial local combining with compression
NLC-TSC	Two-stage algorithm without local combining
O-WLC-TSC	Two-stage algorithm with whole local combining using open address hashing
D-WLC-TSC	Two-stage algorithm with whole local combining using direct address hashing
N-PLC-TSC	Two-stage algorithm with partial local combining with compression
C-PLC-TSC	Two-stage algorithm with partial local combining without compression

Table 2: Summary of ten schemes for array combining scatter functions

Our schemes for the array combining scatter functions are relatively architecture-independent and can be easily extended to arrays of arbitrary dimensions with arbitrary distribution along every dimension. Extensive experimental results are given on the CM-5 and the performance evaluation presented in this paper will be a guideline for deciding which scheme can be used effectively in applications requiring the array combining scatter primitive.

This paper is organized as follows. The coarse-grained, distributed-memory parallel machine model used in this paper is presented in Section 2, and definitions and notations used in this paper are explained in Section 3. Two global combining schemes, direct algorithm and two-stage algorithm, are presented in Section 4, and then four local combining schemes that include a whole local combining scheme using open address hashing, a whole local combining scheme using direct address hashing, a partial local combining scheme without compression, and a partial local combining scheme with compression are explained in Section 5. Experimental results are presented in Section 6. Finally, our conclusion is stated in Section 7.

2 Coarse-Grained, Distributed-Memory Parallel Machine Model

A coarse-grained, distributed-memory parallel machine consists of several processors that have a private local memory system and are connected by an interconnection network. We assume a two-level model of computation rather than any specific underlying network. In the two-level model we assume that the cost for a remote-

memory access is fixed and that it is independent of the distance between a sender and a receiver. The cost for communication between two processors consists of the start-up cost, τ, and the data-transfer rate, $1/\mu$. We assume that τ and μ are constant and independent of the link congestion and distance between two processors.

It follows that the underlying interconnection network can be regarded as a virtual crossbar network in the two-level model. These assumptions closely model the behavior of the CM-5 on which our experimental results are presented [6]. Although our algorithms are analyzed under these assumptions, most of them are architecture-independent and can be efficiently implemented on meshes and hypercubes with wormhole routing [6, 13]. Under the assumption that there is no node contention, the time taken to send a message from one processor to another is modeled as $\tau + \mu m$, where m is the size of the messages [9, 17].

2.1 Communication Primitives

Vector Reduction and Prefix Sum The *vector reduction-sum* computes an element-wise reduction sum of the local vector located in each processor, and the *vector prefix-sum* computes an element-wise exclusive prefix sum of the local vector located in each processor. The basic structure of the algorithm for the prefix-sum is the same as that for the reduction-sum. Therefore, if the same input vector is provided for both communication primitives, we can combine two primitives in order to reduce the communication overhead, especially the total amount of the start-up cost. Now we define the communication primitive, *prefix-reduction-sum*, which performs vector prefix-sum and reduction-sum simultaneously where the prefix operation is exclusive.[1] The time taken to perform the vector prefix-reduction-sum is modeled as $\tau \lg P + \mu M$ where M is the vector size [1]. Readers are referred to [1] for detailed algorithms.

Transportation The *transportation* is a communication primitive that performs many-to-many personalized communication with possibly high variance in message sizes (i.e., with highly non-uniform messages) [15]. The transportation primitive with P processors is represented by communication matrix, $P \times P$ matrix $\Phi = (\phi_{ij})$ where each row index i and column index j ($0 \leq i, j < P$) indicate a source processor and a target processor numbers, respectively. A matrix entry ϕ_{ij} gives the size of message sent by processor i to processor j. Also, each row and column are called send vector *sendl* and receive vector *recvl* [15]. For communication matrix $\Phi = (\phi_{ij})$, the following parameters are defined:

- Outgoing traffic r_i and incoming traffic c_j such that $r_i = \sum_j \phi_{ij}$ and $c_j = \sum_i \phi_{ij}$
- Outgoing traffic bound r, incoming traffic bound c, overall traffic bound t such that $r = \max_i r_i$, $c = \max_j c_j$, and $t = \max(r, c)$

The transportation primitive with incoming and outgoing traffic bounds, r and c, can be completed in time $\mathcal{O}((r+c)\mu)$ when either $r \geq \mathcal{O}(P^2)$ or $c \geq \mathcal{O}(P^2)$. If the overall traffic bound is t, communication can be done in time $\mathcal{O}(2t\mu)$ when $t \geq \mathcal{O}(P^2)$ [15]. Generally, the transportation primitive requires a preliminary communication in order to exchange *sendl* vector, which can be done in time $\mathcal{O}(\tau P)$.

[1] If a control network is available, as on the CM-5, we do not need to combine reduction-sum with prefix-sum. Each primitive can be performed in $\mathcal{O}(M)$ time where M is the vector size.

3 Definitions and Notations

This section presents the notations and definitions that will be used throughout the paper. First, each argument used in the array combining scatter functions is defined as `ARRAY` $= A$, `MASK` $= M$, `INDX`$_k = I_k$ where $0 \leq k < r'$ and `BASE` $= B$. Also, we assume each function returns a resulting (target) array, R. A source array, A, has rank r and shape $(N_{r-1}, N_{r-2}, \ldots, N_1, N_0)$, and a mask and index arrays, M and I_k $(0 \leq k < r')$, are conformable with A. A base array, B, has rank r' and shape $(N'_{r'-1}, N'_{r'-2}, \ldots, N'_1, N'_0)$, and a resulting array, R, is conformable with B. Note that, if logical reduction operations are used in the array combining scatter functions, a mask array takes the place of the source array.

Throughout the paper we assume `SUM` is used as a reduction operation in the array combining scatter functions. Thus collision is resolved by binary associative operator $+$. Also, we assume argument `MASK` is omitted. Even though a mask array is provided, the overall complexity for our schemes will not be significantly changed.

In Sections 4 and 5 we will present global combining schemes and local combining schemes for the array combining scatter functions under the assumption that all arrays have rank one and are distributed among P processors. However, all schemes presented in this paper can be easily extended to multidimensional arrays. For more details, readers are referred to [1, 2].

Now we assume each array defined above is distributed among P processors[2] as follows: source array A of size $N(= N_0)$ and base array B of size $N'(= N'_0)$ are distributed among P processors using `BLOCK-CYCLIC`(W_0) and `BLOCK-CYCLIC`(W'_0), respectively. Also, we assume M and I_0 are aligned to A, and R is aligned to B. If not, we may need to redistribute those arrays as a preliminary step. According to the distribution information, local arrays are defined: A_l, M_l, and Il_0 have shape (L) where $L = N/P$. Also, R_l and B_l have shape (L') where $L' = N'/P$. For the sake of simplicity we assume that $P \backslash N$ and $W_0 \backslash N$, and $P \backslash N'$ and $W'_0 \backslash N'$. Also, we assume that the row-majored indexing scheme is used and all indices start from zero.

4 Global Combining Schemes

In this section we present two global combining algorithms for the array combining scatter functions: *direct (one-stage) algorithm* and *extended two-stage algorithm*. First, for ease of presentation, we define for each i $(0 \leq i < P)$:

- E_i: Set of global indices corresponding to the set of elements of base array B owned by processor P_i. For example, in `BLOCK` distribution $E_i = \{j \mid iL' \leq j < (i+1)L'\}$.
- X_i: Subset of index array I_0 such that $X_i = \{I_0(j) \mid 0 \leq j < N \ \wedge \ I(j) \in E_i\}$.
- $\alpha_i = \mid X_i \mid$: Total number of pointers referring to target processor P_i where $0 \leq \alpha_i \leq N$.
- $\alpha = \max_{(0 \leq i < P)} \alpha_i$.

Also, for index transformation we define two mapping functions, $\mathcal{T}_P$ and $\mathcal{T}_L$, which are explained in Figure 2. Note that in `BLOCK` distribution two functions can be simplified as $\mathcal{T}_P(g) = g \ \mathsf{div} \ L'$ and $\mathcal{T}_L(g) = g \ \mathsf{mod} \ L'$.

[2] We assume the number of source processors is equal to that of target processors.

Mapping each element $Il_0(i)$ $(0 \leq i < L)$ to a processor number

$\mathcal{T}_P : \{0, \ldots, N'-1\} \rightarrow \{0, \ldots, P-1\}$ $\ni$

$\mathcal{T}_P(g) = (g \text{ div } W'_0) \bmod P$

Mapping each element $Il_0(i)$ $(0 \leq i < L)$ to an index of B_l owned by processor $\mathcal{T}_P(Il_0(i))$

$\mathcal{T}_L : \{0, \ldots, N'-1\} \rightarrow \{0, \ldots, L'-1\}$ $\ni$

$\mathcal{T}_L(g) = g \bmod W'_0 + W'_0 \times (g \text{ div } (W'_0 \times P))$

Figure 2: Index transformation functions, $\mathcal{T}_P$ and $\mathcal{T}_L$

Similarly, in cyclic distribution $\mathcal{T}_P(g) = g \bmod P$ and $\mathcal{T}_L(g) = g \text{ div } P$.

In the array combining scatter primitive with arbitrary hot spots, outgoing traffic in sending processors is uniform (i.e., total message size on any sending processor is $L = N/P$). However, an incoming traffic in each receiving processor is α_i. Thus an incoming traffic is bounded by α where $\alpha = \mathcal{O}(N)$ and $\alpha = \Omega(L)$. This may be caused by the presence of hot spots. However, if $N < N'$, this may happen even in the absence of any hot spots. We call a processor, P_i $(0 \leq i < P)$, with an incoming traffic greater than L (i.e., $\alpha_i > L$) a *hot processor*.

Note that in the worst case $\alpha = N$ if all processors send messages to one processor. An example of such a worst case is the single-node accumulation operation. Figure 3 shows an example of the array combining scatter primitive with arbitrary hot spots. In this example $N = 32$ and $N' = 16$, and arrays are distributed in `BLOCK`

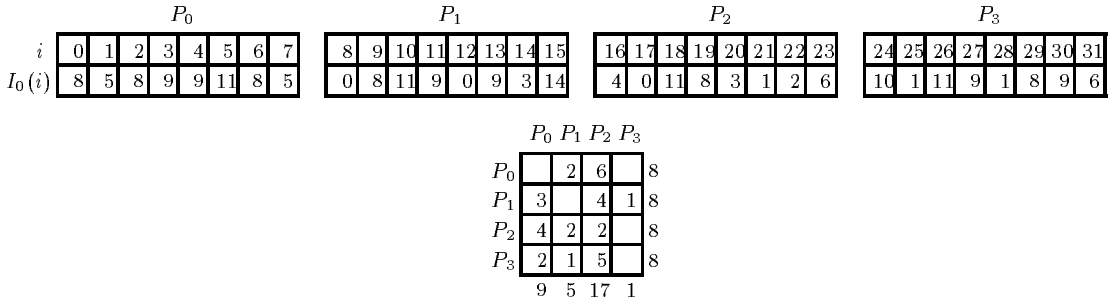

Figure 3: Example of array combining scatter primitive with hot spots

among four processors. Due to the presence of hot spots two processors, P_0 and P_2, are hot processors and $\alpha = 17$. An example of the array combining scatter primitive with hot processors, but without hot spots, is shown in Figure 4. In this figure $N = 32$ and $N' = 64$, and arrays are distributed in `BLOCK` among four processors. We can see that P_0 and P_1 are hot processors and that $\alpha = 16$ even in the absence of any hot spots.

4.1 Direct (One-Stage) Algorithm

In the direct algorithm each source datum is sent directly to target processors from source processors under the control of the index array, and combining is then performed on target processors. The direct algorithm for the array combining scatter primitive consists of four steps:

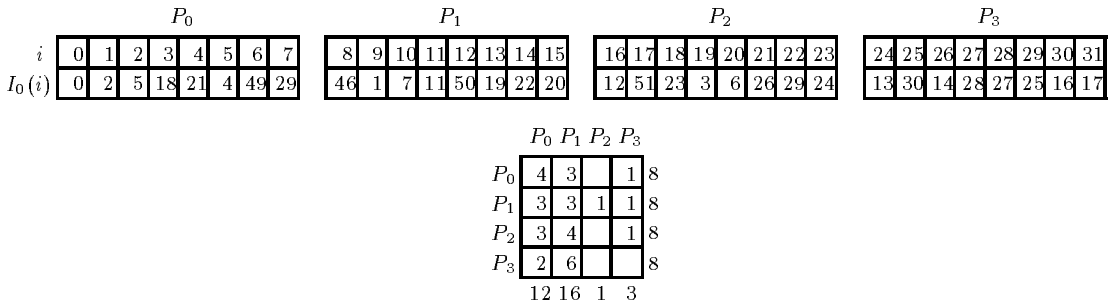

Figure 4: Example of array combining scatter primitive with hot processors, but without hot spots

1. Initial scan for generating *sendl* vector
2. Message composition (second scan)
3. Transportation between source and target processors
4. Message decomposition

In Step 1 we scan each element of the local index array while updating the *sendl* vector by using mapping function $\mathcal{T}_P$. Then we compose messages to be sent to each processor in Step 2 by using the *sendl* vector. At this time each message consists of pairs having format [local index, datum]. Note that the outgoing traffic is uniform, while the incoming traffic varies. Thus for good load balancing it is better to transform each global index into an index locally valid on a receiving processor before sending messages on source processors. This can be done by using mapping function $\mathcal{T}_L$. In Step 3 the transportation primitive is performed where the outgoing traffic is uniform[3] and the incoming traffic is upper-bounded by α. In Step 4 the resulting array is first initialized, that is, all data in the base array are copied into the resulting array, and then all received data are accumulated in the resulting array as combining colliding elements.

The time taken by Steps 1 and 2 is bounded by $\mathcal{O}(L)$ and $\mathcal{O}(2L)$, respectively, but in Steps 3 and 4 the costs are bounded by $\mathcal{O}(\alpha)$ and $\mathcal{O}(\alpha + L')$, respectively.[4] Note that in Step 2 we need to scan two arrays, A_l and Il_0, while we need to do only one array, Il_0, in Step 1.

4.2 Two-Stage Algorithm

First, we introduce Shankar and Ranka's two-stage algorithm for random access write [16] and then present our extended two-stage algorithm for the array combining scatter functions.

4.2.1 Two-Stage Algorithm for Random Access Write

We assume all arrays of rank one have the same shape (N) and are distributed using `BLOCK` distribution among P processors. Then the *two-stage algorithm for random access write* splits the communication required in RAW

[3] In practice, on each processor we need only to send out elements referring to other processors, so outgoing traffic is less than or equal to L.

[4] We use constants in complexity for the purpose of comparison, although the constants are insignificant in terms of asymptotic complexity.

into two stages, which results in lower traffic bound $\mathcal{O}(L)$ at Stages 1 and 2. That is, RAW is not directly performed, but done in two stages through P intermediate processors.

In this algorithm, L elements located in each processor are initially partitioned into equal-sized K buckets (bucket size $= L/K$) where $K \geq 1$. The basic idea is that, during RAW, these buckets are dynamically stretched or shrunk to reduce communication overhead. A high-level description of the algorithm is given in Figure 5. For more details, readers are referred to [16].

1. *Preprocessing of Stage* I: In every source processor a bucket vector of size KP is created. The number of writes to the range $[i\frac{L}{K} : (i+1)\frac{L}{K} - 1]$ is recorded in the i^{th} element of the bucket vector, $0 \leq i < KP$. The local count is followed by a vector reduction sum (replicated on every processor) which gives the total number of writes to each bucket. This resulting bucket vector is used to balance the communication.
2. *Stage* I *Communication*: Bounded transportation to send writes to intermediate processors from source processors using bucket vector such that the traffic is bounded by L.
3. *Preprocessing of Stage* II: Processing at the intermediate processors. That is, combining is performed through the segmented vector prefix sum operation.
4. *Stage* II *Communication*: Bounded transportation between the intermediate and target processors such that the traffic is bounded by L.
5. *Processing at the target processors.*

Figure 5: Two-stage algorithm for random access write (RAW)

4.2.2 Two-Stage Algorithm for Array Combining Scatter Functions

In this section we explain how to extend the two-stage algorithm for RAW (presented in the previous section) to the array combining scatter functions. In RAW it is assumed that the size of the source array is equal to that of the target and that all arrays are distributed using `BLOCK` distribution. Thus our extension will deal with *different values of N and N'*, and with *block-cyclically distributed base arrays*. First, we assume that a hybrid algorithm [16] is used in our presentation. In the hybrid algorithm each bucket is classified as a *dense* or a *sparse* bucket, and then only dense buckets are processed through two-stage communication. For ease of presentation, the extended two-stage algorithm may be divided into eight steps, as presented in Figure 6.

Different Values of N and N'

In this section we show how to apply a two-stage algorithm to a case in which the size of a source (index) array, N, is different from that of a base array, N'. First, we assume that all arrays are distributed using `BLOCK` distribution among P processors. In the next section we will explain how to deal with block-cyclically distributed arrays.

When $N \neq N'$, each local resulting array R_l is partitioned into K buckets where $0 < K \leq L'$. Each bucket size is L'/K and there are a total of KP buckets. Then, on each source processor we can define pointer-set PS_i

1. First scan: Generate bucket vector of size KP (say, $Lcount$).
2. First preprocessing: Redefine buckets through vector reduction/prefix-sum and local prefix-sum operations.
3. Message composition (second scan): Prepare two buffers, one for sparse buckets and the other for dense buckets.
4. First communication: Perform the following two-phase communication:
   - Phase 1: Send the buffer for sparse buckets to target processors.
   - Phase 2: Send the buffer for dense buckets to intermediate processors.
5. Initialization of a resulting array and message decomposition for sparse buckets on target processors.
6. Second preprocessing: Collision resolution through vector segmented prefix-sum (direct address hashing).
7. Second communication from intermediate to target processors.
8. Message decomposition for dense buckets on target processors.

Figure 6: Eight steps in two-stage algorithm for array combining scatter functions

and vector $Lcount$ in such a way that

$$PS_i = \{Il_0(j) \mid 0 \leq j < L \ \wedge \ \frac{L'}{K}i \leq Il_0(j) < \frac{L'}{K}(i+1)\} \ \text{ and } \ Lcount(i) = \mid PS_i \mid, \tag{1}$$

where $0 \leq i < KP$. Since a bucket where local source element $A_l(i)$ has to be written can be identified as $Il_0(i)$ div (L'/K), we can generate vector $Lcount$ while scanning Il_0 in Step 1.

In Step 2 we first combine P local $Lcount$ vectors located on all source processors through the vector reduction sum. Assume that a resulting vector for reduction-sum is $Bcount$ such that for all i $(0 \leq i < KP)$ $Bcount(i) = \sum_{j=0}^{P-1} Lcount(i)$. If $K = 1$, $Bcount(i)$ will be equal to α_i, as defined in Section 3. If $Bcount(i) > L/K$, bucket i is called a dense bucket; otherwise, it is called a sparse bucket. It should be noted that we use L/K as a criteria, not L'/K, since our goal is to eliminate the presence of hot processors. In a hybrid algorithm we need two phases of communication in Step 4, as mentioned before. During Step 4 the total outgoing traffic on each source processor, that is, sums of outgoing traffic for Phases 1 and 2 are uniform (i.e., L). Also, incoming traffic on target processors is bounded by L.

In Stage II we deal with dense buckets. After identifying which buckets are located on each intermediate processor, we resolve collision by performing direct address hashing in Step 6 [3]. To do so we first assign a hash table of size L'/K to each bucket. The number of elements of a dense bucket is greater than L/K. Hence, on each intermediate processor, there exists at most $K-1$ full buckets and two partial buckets. Therefore, the number of buckets on each intermediate processor is bounded by $\mathcal{O}(K+1)$, and the time taken to allocate and initialize hash tables will be $\mathcal{O}((K+1)L'/K)$. For this reason the hashing overhead increases with the size of target array N'. The hashing can be done in three phases:

1. *Local hashing*: For each bucket on an intermediate processor, perform hashing with local elements received from all source processors. Since each intermediate processor received at most L elements in Stage I, the time taken for local hashing is $\mathcal{O}(L)$.

2. *Global hashing*: Combine all local hash tables for each bucket through the vector segmented prefix-sum. Its cost is bounded by $\mathcal{O}(\tau \lg P + (L'/K)\mu)$.

3. *Supplementary global hashing*: Only if a processor includes both a succeeding partial bucket and a preceding partial bucket, does the preceding partial bucket need to be sent to the processor where the bucket started. This takes time $\mathcal{O}(\tau + (L'/K)\mu)$.

After finishing the collision resolution we send only data fields of hash tables to target processors in Step 7. Note that we do not need to send any index, since all elements of a hash table are already in order and each hash table is itself sent directly to a target processor. The outgoing and incoming traffic are bounded by $\mathcal{O}(L')$. In Step 8 only such data that are not equal to the identity value are processed during message decomposition. Table 3 shows a detailed complexity analysis of the extended two-stage algorithm for the array combining scatter functions. Note that in this table we assume that a control network is not available.

Block-Cyclically Distributed Base Array

When base array B is distributed using `BLOCK` distribution among P processors, any two neighboring elements in a bucket have consecutive global indices. Thus a bucket number where a source element, $A_l(i)$, has to be written can be easily identified as $Il_0(i)$ div (L'/K) where $0 \leq Il_0(i) < N'$ and $0 \leq i < L$. But, if array B is not distributed using `BLOCK` distribution, the global indices of any two neighboring elements in a bucket are not always consecutive. Thus we need first to identify a target processor number, $\mathcal{T}_P(Il_0(i))$ (say, p), and a local index valid on the target processor, $\mathcal{T}_L(Il_0(i))$ (say, l), to identify a bucket number. Now we define the transformation function, $\mathcal{T}_B$, to map a pair [target processor number p, local index l] to a bucket number:

$$\mathcal{T}_B : \{0, \ldots, P-1\} \times \{0, \ldots, L'-1\} \rightarrow \{0, \ldots, K-1\} \ni \mathcal{T}_B(p, l) = K \times p + l \text{ div } \frac{L'}{K}.$$

Therefore, we need a total of three index transformations to identify a bucket number if base array B is not distributed using `BLOCK` distribution. We would like to note that the distribution of source and index arrays does not affect the performance significantly, since we need only to scan all local source elements sequentially.

In the two-stage algorithm we need to identify bucket numbers in the following three steps: Step 1 for generating *Lcount* vector, Step 3 for message composition, and Step 6 for local hashing.

Indexing Overhead Suppose three transformation functions, $\mathcal{T}_P, \mathcal{T}_L$, and $\mathcal{T}_B$, can be completed in constant time c_p, c_l, and c_b, respectively.[5] In the direct algorithm, indexing overhead due to a block-cyclically distributed base array is $\mathcal{O}(L(2c_p + c_l))$. On the other hand, in the two-stage algorithm it is $\mathcal{O}(3L(c_p + c_l + c_b))$. Therefore, we have more indexing overhead in the two-stage algorithm. For optimization schemes to decrease indexing overhead, readers are referred to [1].

4.3 Modeling

In this section we present modeling for two schemes, NLC-OSC and NLC-TSC, and then compare the two schemes based on our modeling. We first explain how to generate sample index arrays, which were also used in

[5] Three functions, $\mathcal{T}_P, \mathcal{T}_L$, and $\mathcal{T}_B$, and the associated constants, c_p, c_l, and c_b, will be used in the rest of this presentation.

Step	Cost	Description
1	$\Theta(L)$	Initial scan
2	$\Theta(2(\tau \lg P + KP\mu))$	Vector reduction-sum and prefix-sum
	$\Theta(KP)$	Classifying buckets while updating *sendl* for sparse buckets
	$\Theta(2KP)$	Exclusive local prefix-sum and element-wise vector sum
	$\mathcal{O}(KP)$	Update *sendl* for dense buckets
3	$\Theta(2L)$	Two phases of message composition
4	$\mathcal{O}(L\mu)$	Bounded transportation for sparse bucket
	$\mathcal{O}(L\mu)$	Bounded transportation for dense bucket
5	$\Theta(L')$	Initializing a resulting array
	$\mathcal{O}(L)$	Message decomposition for sparse bucket
6	$\mathcal{O}(KP)$	Identifying bucket number
	$\mathcal{O}(K+1)$	Calculating buffer size while updating *sendl*
	$\mathcal{O}((K+1)L'/K)$	Initializing the buffer by identity value
	$\Theta(L)$	Copying to the buffer (local hashing)
	$\mathcal{O}(KP)$	Identifying segment
	$\mathcal{O}(\tau \lg P + (L'/K)\mu)$	Segmented prefix sum
	$\mathcal{O}(\tau + (L'/K)\mu)$	One-to-one communication
7	$\mathcal{O}(L'\mu)$	Many-to-many communication for dense bucket
8	$\mathcal{O}(L')$	Message decomposition for dense bucket

Table 3: Complexity analysis of two-stage algorithm for array combining scatter functions

our experiments. We assume all arrays have rank one and are distributed in `BLOCK` for simplicity of modeling.

4.3.1 Generation of Sample Index Array

Let H and δ be the number of hot processors and hot degree where $0 \leq H < P$ and $0.0 \leq \delta \leq 1.0$. The hot processors, H_j $(0 \leq j < H)$, are selected randomly. Note that all hot processors are different from each other. All elements of a local index array, Il_0, then consist of two sets:

- Set of random indices of size $(1-\delta)L$: each element r is chosen at random in the range of $0 \leq r < N'$.
- Set of skew indices of size δL: This set consists of H subsets, $S_0, \ldots, S_{H-1}$. The size of the first (δL `mod` H) subsets is $\lceil \delta L/H \rceil$ and that of the remaining subsets is $\lfloor \delta L/H \rfloor$. Each element s_i^j belonging to S_j is selected at random in the range of $H_j L' \leq s_i^j < (H_j+1)L'$, where $0 \leq j < H$ and $0 \leq i < \lceil \delta L/H \rceil$ or $\lfloor \delta L/H \rfloor$.

For simplicity of modeling we assume $H \backslash \delta L$. Then, for each hot processor H_j, α_{H_j} is defined $\alpha_{H_j} = \left(\frac{(1-\delta)L}{P} + \frac{\delta L}{H}\right) \times P = L + \delta\left(\frac{N}{H} - L\right)$. Also, for processor k, which is not a hot processor, $\alpha_k = (1-\delta)L$. Note that $\alpha = \alpha_{H_j} = L + \delta\left(\frac{N}{H} - L\right)$. Therefore, α is lower-bounded by L with $\delta = 0.0$ and upper-bounded by N with $H = 1$ and $\delta = 1.0$.

4.3.2 Modeling

On processor i $(0 \leq i < P)$, time taken for local computation and communication is proportional to $\mathcal{C}_{\mathcal{L}} L + \mathcal{C}_{\mathcal{L}'} L' + \mathcal{C}_{\mathcal{K}} K + \mathcal{C}_{\mathcal{B}} B + \mathcal{C}_{\alpha} \alpha$, where B is the total number of buckets such as $B = KP$, and $\mathcal{C}_{\mathcal{L}}, \mathcal{C}_{\mathcal{L}'}, \mathcal{C}_{\mathcal{K}}, \mathcal{C}_{\mathcal{B}}$, and $\mathcal{C}_{\alpha}$

are constants and not less than zero. The values of these five constants will depend on the scheme. Note that $\mathcal{C}_{\mathcal{K}} = \mathcal{C}_{\mathcal{B}} = 0$ in the direct algorithm, while $\mathcal{C}_{\alpha} = 0$ in the two-stage algorithm.

We have already analyzed the costs for each step in two schemes, NLC-OSC and NLC-TSC, in Sections 4.1 and 4.2, respectively. Thus, based on such complexity analysis, we may find estimated values of the above-mentioned constants. For the sake of simplicity we assume total communication time is dominated by data-transfer time. That is, startup costs are disregarded in our modeling.

The costs for the direct algorithm without local combining (NLC-OSC) and the two-stage algorithm without local combining (NLC-TSC) can then be modeled as given in Table 4. According to the cost model we may expect NLC-TSC to have lower local computation time as compared to NLC-OSC when

$$\alpha > 6KP + K + 1 + 2L + \left(\frac{1}{K} + 2\right) L' \tag{2}$$

$$\equiv \delta \left(\frac{N}{H} - L\right) > 6KP + K + 1 + L + \left(\frac{1}{K} + 2\right) L'. \tag{3}$$

Similarly, we may expect NLC-TSC to have lower communication time than NLC-OSC when

$$\alpha\mu > \left(2KP + \frac{L'}{K}\right)\mu + \left(\left(\frac{1}{K} + 1\right) L' + 2L\right)\mu \tag{4}$$

$$\equiv \delta \left(\frac{N}{H} - L\right)\mu > \left(2KP + \left(\frac{2}{K} + 1\right) L' + L\right)\mu. \tag{5}$$

Let right-hand sides of Inequalities (2) and (4) be T_l and T_c, respectively. The total cost can be modeled by $\alpha(1+\mu)$ and $T_l + T_c$ in NLC-OSC and NLC-TSC, respectively. Now we may expect the time taken by NLC-TSC to be less than that by NLC-OSC when

$$\alpha(1+\mu) > T_l + T_c \equiv \left(L + \delta\left(\frac{N}{H} - L\right)\right)(1+\mu) > T_l + T_c \tag{6}$$

$$\equiv \delta > \frac{T_l + T_c - (1+\mu)\, L}{(N/H - L)\,(1+\mu)}. \tag{7}$$

Estimation of threshold hot degree Based on the cost model we may estimate a threshold value, ω, such that NLC-TSC is better than NLC-OSC when $\delta \geq \omega$. The right-hand side of Inequality (7) will be a threshold hot degree, ω. However, μ is a machine-dependent constant based on network bandwidth. For machine-independent modeling (i.e., independent of μ) alternatively, we can say that NLC-TSC will be better than NLC-OSC if $\alpha > T_l \ \wedge \ \alpha > T_c$, which is a stricter condition than Inequality (6) (i.e., a sufficient condition). We define two analytic threshold hot degrees, ω_l and ω_c, such that

$$\omega_l = \frac{T_l - L}{N/H - L} \text{ and } \omega_c = \frac{T_c - L}{N/H - L}. \tag{8}$$

Now we may say that NLC-TSC will be analytically better than NLC-OSC with hot degree δ such that $\delta > \omega_a = \max(\omega_l, \omega_c)$. Since we use a strict condition, in some cases ω_a may be larger than ω based on Inequality (6) (i.e., $\omega \leq \omega_a$).

Note that if $K \geq 1$, $(1/K + 2) \geq (2/K + 1)$. It follows that $T_l > T_c/\alpha$. Thus, if Inequality (2) is satisfied, Inequality (4) is also satisfied. Since $K \geq 1$ in the two-stage algorithm, $\max(\omega_l, \omega_c)$ will be equal to ω_l. Later, in Section 6.2, we will compare these analytically estimated threshold values with an experimental one.

Direct algorithm without local combining

Local computation	$3L + \alpha + L' = 4L + L' + \delta\left(\frac{N}{H} - L\right)$
Communication	$\alpha\mu = \left(L + \delta\left(\frac{N}{H} - L\right)\mu\right)$

Two-stage algorithm without local combining

Local computation	$6KP + K + 1 + 5L + \left(\frac{1}{K} + 3\right)L'$
Global communication	$\left(2KP + \frac{L'}{K}\right)\mu$
Transportation	$\left(\left(\frac{1}{K} + 1\right)L' + 2L\right)\mu$

Table 4: Cost model for direct and two-stage algorithms without local combining

Analysis As H decreases and δ increases, NLC-TSC will give a much better performance than NLC-OSC. Given a fixed $L = N/P$ and $L' = N'/P$, while the performance of NLC-OSC does not greatly depend on L' (size of target array), that of the second stage in the NLC-TSC is greatly affected by L'. Therefore, when L' is relatively small compared to L, NLC-TSC will work better. Also, as P increases, NLC-TSC will be much better than NLC-OSC. Since N has to increase as much as P increases with a fixed L, the performance of NLC-OSC is affected more by the increase of N than is the performance of NLC-TSC. More precisely, the increase of N and P affect the cost $\delta(N/H)$ in NLC-OSC and KP in NLC-TSC with a fixed L and L' (see Inequalities (3) and (5)).

Decision System for Global Combining If information about the input index array is not available, either one-stage or two-stage algorithm may be automatically selected through a decision system that can be developed based on the modeling presented in the previous section. The criteria should be based on α (the maximum number of writes among all target processor). Readers are referred to [1] for more details.

5 Local Combining Schemes

In the direct algorithm the presence of hot processors, due to the variance in incoming traffic on target processors, degrades the performance of the last two steps, including the transportation between the source and target processors and message decomposition on target processors. We have developed four local combining schemes in order to reduce outgoing traffic at source processors as well as variance in incoming traffic on target processors:

- *Whole local combining using open address hashing* (O-WLC)
- *Whole local combining using direct address hashing* (D-WLC)
- *Partial local combining without compression* (N-PLC)

- *Partial local combining with compression* (C-PLC)

The local combining scheme performs collision resolution locally, which may possibly reduce message size, by using the hashing. In the folowings, our presentation will first be limited to the direct algorithm. Later we will explain how all the local combining schemes can be applied to the two-stage algorithm.

5.1 Whole Local Combining (WLC)

To perform collision resolution locally we can perform hashing during the first scan. The two hashing techniques, *open address hashing* and *direct address hashing*, can be used [3].

Whole Local Combining Using Open Address Hashing We perform open address hashing during the first scan [3]. In open address hashing each element of Il_0 is used as a key. The difference between the open address hashing here in WLC and the usual one is that if a collision happens during hashing and two colliding elements have the same key (pointer), we combine two elements according to the predefined collision resolution rule, rather than search for the next available slot. The element of a hash table is a global index, $Il_0(i)$ (hashing key), with a datum (satellite information).

Whole Local Combining Using Direct Address Hashing The whole local combining scheme using open address hashing entails a significant hashing overhead. To reduce such an overhead, direct address hashing [3] can be used instead of open address hashing. We can use a hash table of size N' where each entry is a datum. Note that we do not need to store any index used as a hashing key. Whenever a collision occurs, we need only to combine two colliding elements. WLC using direct address hashing entails much less hashing overhead than WLC using open address hashing. Note, however, that there is a restriction such that the size of a local memory should be not less than N' on a machine.

5.2 Partial Local Combining (PLC)

In the whole local combining scheme we perform local combining through all elements of the local source array. Hence we do not care whether there exist any hot pointers or which hot spots (target elements) are referred to by any hot pointers. Note that we call a pointer (i.e., an element of Il_0) that refers to a hot spot a *hot pointer*. For example, in Figure 3 three pointers on P_0, $Il_0(0)$, $Il_0(2)$, and $Il_0(6)$, are hot pointers referring to target element 8, which is a hot spot.

In the *partial local combining scheme* (PLC) we first obtain information about the existence of hot pointers and the locations of referred hot spots. We then perform local combining for a specific part of the source elements based on such information.

Partial Local Combining without Compression Partial local combining is a bucket-based hashing, thus we partition each local base (resulting) array into K buckets in the same manner as is done in the two-stage algorithm. On each source processor we define pointer-set PS_i and vector *Lcount* in the same way as Equation (1) where $0 \leq i < KP$. During the initial scan, we can easily update a vector *Lcount* instead of a *sendl* vector while

scanning Il_0. After updating *Lcount* each pointer-set PS_i is classified to *locally dense* or *locally sparse* set. If $Lcount(i) > \beta\,(N'/KP)$, PS_i is regarded as a locally dense set where β is a local scale factor as a a criteria. Note that if $Lcount(i) > N'/KP$, there must exist a hot pointer referring to a hot spot in bucket i. However, it should be noted that $Lcount(i) \leq N'/KP$ does not necessarily mean that there are no hot pointers.

For each locally dense pointer-set PS_i we first reset $Lcount(i)$ to N'/KP and then perform direct address hashing [3] during message composition. Note that the size of a hash table is N'/PK and each entry of a hash table is a local index (hashing key), valid on a target processor with a datum. Whenever a collision occurs, we need only to combine two colliding elements. The main idea here is that each hash table for a locally dense pointer-set PS_i will be directly sent to the corresponding target processor.

Partial Local Combining with Compression In the previous scheme, if there exists at least one locally dense bucket, the outgoing traffic will be reduced, but dummy data may be included in the message. In PLC with compression we send only actual data through compression; that is, we pack a hash table with actual data, which results in decreasing the message size. However, this incurs extra cost.

5.3 Comparison of Local Combining Schemes

The main goal of the local combining schemes presented in the previous sections is to reduce outgoing traffic on each source processor by removing hot spots to be identified locally, which may possibly reduce incoming traffic on hot processors, that is, an incoming traffic bound. Thus the performances of local combining schemes largely depend on how much the hashing overhead can be amortized by decreasing the time taken by communication and message decomposition on hot processors. If the data-transfer rate $(1/\mu)$ in communication is relatively low and the cost for unit local computation is relatively small in a coarse-grained, distributed-memory parallel machine, local combining will have a better performance. Note that no local combining schemes are better in the absence of hot spots. Thus, if $L \gg L'$, performing a local collision resolution will be a better option.

Generally, the whole local combining schemes require more hashing overhead than the partial local combining schemes, but we can remove all hot spots identified locally. As a result, if any hot spots are identified locally, outgoing traffic should be reduced. In the partial local combining schemes, local combining is done partially only for locally dense buckets, so we cannot always remove all hot spots. The advantage is that we have less hashing overhead. As K increases, it may be possible for more hot spots to be removed locally, but the time required to manage *Lcount* will increase.

Let H be the number of hot processors. Then, if H is relatively small and $\alpha_{H_j} \gg L$ for each hot processor H_j $(0 \leq j < H$ and $0 \leq H_j < P)$, local combining can be effectively performed. If most hot pointers belonging to X_{H_j} are originally distributed in a small number of source processors rather than distributed evenly among all source processors, the performance of local combining schemes will be better. Especially, in the partial local combining schemes, if all pointers belonging to X_{H_j} actually refer to a small portion of target elements in H_j, collision can be resolved more efficiently, because all pointers will belong to a few buckets. Thus, the range of hot spots in target processors will directly affect the performances of partial local combining schemes. The main factor in the performances of local combining schemes will be *locality of hot pointers* in source processors as well as *locality of hot spots* in target processors.

5.4 Local Combining Schemes in Two-Stage Algorithm

Although our presentation is limited to the direct algorithm, all the local combining schemes mentioned above can easily be applied to the two-stage algorithm. Actually, Steps 1, 2, and 3 in Figure 6 can be modified in a manner similar to that in the direct algorithm. Note, however, that in TSC schemes transportation for dense buckets originally has uniform incoming traffic at each intermediate processor and reduction in incoming traffic due to local combining is less than L. On the other hand, in OSC schemes local combining may not only reduce variance in incoming traffic, but also the incoming traffic bound on target processors. Note that less variance results in a better performance, both in transportation and in message decomposition on target processors. Also, the decrease of the incoming traffic bound in OSC must be much less than that in TSC. Therefore, local combining schemes in TSC schemes may not improve the performance as much as those in OSC schemes do.

6 Experimental Results

All schemes for the array combining scatter functions presented in this chapter were programmed in C on the CM-5. As stated in Section 4.3, the performances of these schemes mainly depend on α, L, L' and P. Our experiments were conducted by using three categories of data sets (see Figure 7) to study the effects of various L, L' and P. To evaluate the performances, we need to generate various index arrays showing different degrees

- Category 1: $L = L'$
  Four one-dimensional source and base arrays with local array size $L = 1024, 2048, 4096, 8192$. Given an L, $P = 64, 128$.
- Category 2: $L < L'$
  Four pairs of one-dimensional source and base arrays with local array sizes $(L, L') = (1024, 2048), (1024, 4096), (2048, 4096), (2048, 8192)$. Note that $L' = 2L$ or $4L$. Given a pair (L, L'), $P = 64, 128$.
- Category 3: $L > L'$
  Four pairs of one-dimensional source and base arrays with local array sizes $(L, L') = (2048, 1024), (4096, 1024), (4096, 2048), (8192, 2048)$. Note that $L = 2L'$ or $4L'$. Given a pair (L, L'), $P = 64, 128$.

Figure 7: Three categories of data sets

of data skew (i.e., different α ranging from L through N). By generating sample index arrays as described in Section 4.3.1, we can make various α values according to two factors, H and δ. Note that α increases with δ with a fixed H, while it decreases as H increases with a fixed δ. For this reason we conducted the following two types of experiments with the three categories of data sets mentioned above:

1. Performance evaluation of ten schemes (summarized in Table 2) with $H = 1$ and $\delta = 0.0, 0.2, 0.4, 0.6, 0.8, 1.0$ for arrays of rank one that are distributed in `BLOCK`.
2. Experimental threshold hot degree in two global combining schemes without local combining with $H = 1, 2, 4$ for arrays of rank one that are distributed in `BLOCK`.

In the first experiment we compared the performances of ten schemes with various α values ranging from L to N. In the second experiment we evaluated two global combining schemes without local combining by studying the effects of H.

In each category we used five different K such that $K = 1, 2, 4, 8, 16$ for bucket-based algorithms, and in partial local combining schemes we set local scale factor to one. To measure the maximum time among all processors, several barriers were placed between routines, but in general these barriers are not necessary. Note that hot processors generally show the maximum time. Also, for exact timing on the CM-5, processor 0 was always selected as a hot processor. For the transportation primitive, linear permutation using active messages and a randomized distributed algorithm using active messages were used in the direct algorithm and the two-stage algorithm, respectively [21, 22, 15]. Also, for the vector prefix-reduction sum, a CM-5 global function was used [4].

6.1 Performance Evaluation of Ten Schemes

In each category we used various hot degrees $\delta = 0.0, 0.2, 0.4, 0.6, 0.8, 1.0$ and fixed the value of H to 1 in order to compare the performances of ten different schemes for the array combining scatter functions. For succinctness, only partial results are presented in Figures 8 through 10. Full results are available in [2]. For bucket-based algorithms, only the best results with optimal K values are shown. Although the optimal K values are not shown in figures, our experimental results showed that they are 1,2, or 4 in most cases.

As can be seen in the results, most TSC schemes give better performances than the OSC schemes, except when δ is relatively small. As noted in Section 4.3, the performance of TSC schemes depends on both L and L', while that of OSC schemes is largely affected by L. Thus, in Category 3, TSC schemes work much better.

6.1.1 Effect of δ

We define γ such that $\gamma = \frac{T(\delta+x)-T(\delta)}{x}$ where $T(\delta)$ is the time taken with hot degree δ and $x > 0$. In all categories NLC-OSC gives the highest γ among all ten schemes, that is, it is most sensitive to δ. Note that in NLC-OSC, $\gamma > 0$ for all δ, thus we may regard γ as a rate of increase. Moreover, γ increases with L and P, mainly due to the increased incoming traffic in a hot processor.

NLC-TSC and two PLC-TSC schemes are least sensitive to δ in all categories. The time taken by these three TSC schemes generally decreases as δ increases, unlike in OSC schemes. Thus γ now represents a rate of decrease (i.e., $\gamma \leq 0$) and γ is close to zero. D-WLC-TSC also gives γ less than zero, but it has $|\gamma|$ greater than in the above three schemes. As described before, α increases with δ, thus the increase of δ degrades the performances of OSC schemes but does not degrade those of the TSC schemes. As δ increases, TSC schemes are much better than OSC schemes.

6.1.2 Effects of L, L' and P

In Category 1, as δ increases, the time taken by all OSC schemes is directly proportional to L and also directly proportional to P. In all TSC schemes the performance is proportional to L. However, when P is increased,

the time taken by TSC schemes, even for a large value of the hot degree, does not increase as much as in OSC schemes. In other words, the increase in time is not large as compared to that in OSC schemes.

In Category 2, given a fixed L, increasing L' does not greatly affect the performances of NLC-OSC and two PLC-OSC schemes. However, it does affect the performances of two WLC-OSC schemes and all TSC schemes. In TSC schemes the performance of Stage 2 is affected by L'. The two D-WLC schemes need to deal with hash tables of size $N' = L'P$. Thus, given a fixed P, the time for initializing the hash table is directly proportional to L'. In Category 3, given a fixed L', the time taken by all ten schemes mainly depends on L. Two O-WLC schemes are especially affected by L due to hashing overhead, which results from the increase of the total amount of linear probing.

6.1.3 Evaluation of Local Combining Schemes

Although any local combining schemes based on OSC do not give the best performances among the ten schemes, local combining in OSC schemes significantly improve the performance with relatively high δ. This effect is clearly more significant than in TSC schemes. In Category 1, two WLC-OSC schemes improve performances significantly with relatively high δ, while only O-WLC-OSC does in Category 2. On the contrary, in Category 3 D-WLC-OSC and two PLC-OSC schemes perform well as compared to O-WLC-OSC. D-WLC-OSC has less hashing overhead as L' decreases. The two bucket-based PLC-OSC schemes work much better with small L'. Note that C-PLC schemes entail more overhead in local computation than N-PLC schemes. However, the outgoing traffic in C-PLC schemes is not always larger than that in N-PLC schemes, due to compression. Actually, we can see that C-PLC-OSC is better than N-PLC-OSC in Category 3 when δ is relatively high.

As mentioned in Section 5.4, local combining schemes in TSC schemes do not improve the performance as much as in OSC schemes. In Category 1, only O-WLC-TSC improved the performance for $\delta \geq 0.8$. It also performs better as L and P increase. Two PLC schemes are always a little worse than NLC-TSC, even when $\delta = 1.0$, but the difference is not substantial. This difference shows the basic overhead entailed by two PLC schemes when locally dense buckets are absent. In Category 2, no local combining schemes improve performance. On the contrary, in Category 3 both O-WLC-TSC and N-PLC-TSC are better than NLC-TSC, with relatively high hot degree. Generally, O-WLC-TSC is better than NLC-TSC only when $\delta = 1.0$, while N-PLC-TSC is better for $\delta > 0.6$ with $L = 4L'$ and $\delta > 0.8$ with $L = 2L'$. However, when both O-WLC-TSC and N-PLC-TSC are better than NLC-TSC, O-WLC-TSC is better than N-PLC-TSC.

The results show that the overhead due to local combining in TSC schemes is not amortized by the reduction in incoming traffic at intermediate processors as fully as is that in OSC schemes. Actually, we can see that neither D-WLC-TSC nor N-PLC-TSC improves the performance, while all local combining schemes in OSC improve the performance with relatively high δ in Categories 1 and 3. Note that if the data-transfer rate $(1/\mu)$ in communication is relatively low and the cost for unit local computation is relatively small in a coarse-grained, distributed-memory parallel machine, local combining will be performed more efficiently, even in the two-stage algorithm. We expect this to be the case in other machines such as the IBM SP2 [7, 18] and the Intel Paragon [19].

6.2 Experimental Threshold Hot Degree with $H = 1, 2, 4$

In the previous experiment we fixed H to one. By using two more values of H (2 and 4), we compared performances of two schemes, NLC-OSC and NLC-TSC, to evaluate the experimental threshold for hot degrees. This experiment was motivated by the modeling presented in Section 4.3. In Table 5 we show experimental threshold values ω_e such that NLC-TSC is better than NLC-OSC for $\delta \geq \omega_e$. In this experiment we used various hot degree δ ranging from 0.0 to 1.0 and increasing by 0.02. Thus, actual threshold ω_a is in the range of $\omega_e - 0.02 < \omega_a \leq \omega_e$. Table 5 also shows two analytically estimated threshold hot degrees, ω_l and ω_c, as presented in Section 4.3. The optimal K values obtained from the experiments were used to estimate threshold values. As stated before, we estimate the threshold hot degree as $\max(\omega_l, \omega_c) = \omega_l$.

Since our modeling is based on complexity analysis, it is difficult to estimate the exact threshold values. For this reason, the experimental threshold ω_e is generally greater than the estimated threshold hot degree. However, ω_e is affected by L, L', P, and H in the same manner as ω_l and ω_c, as described in Section 4.3. For Category 2, the estimated threshold is close to ω_e. Also, it is much closer to ω_e as L and L' increase with respect to P. In the following we will examine how the experimental threshold hot degree is affected by various input factors.

Effect of P Given fixed L and L', as P increases, NLC-TSC is much better than NLC-OSC, hence the increase of N affects the performance of NLC-OSC more than that of NLC-TSC, as seen in Section 4.3. In all categories we can see that ω_e with $P = 128$ is less than that with $P = 64$.

Effect of L and L' Given fixed P and H, we can see that ω_e decreases as both L and L' increase. This is due to the increase of incoming traffic in the hot processors in NLC-OSC. Note that the increase of L' greatly degrades the performance of NLC-TSC, but not that of NLC-OSC.

Effect of H Since α decreases as H increases, the increase of H improves the performance of NLC-OSC significantly, but not that of NLC-TSC. So ω_e increases as much as H does. Thus NLC-TSC will be more useful when H is relatively small and δ is relatively high (i.e., α is relatively large). As can be seen in Table 5, NLC-TSC works much better than NLC-OSC in Category 3, especially when L is much larger than L'. On the contrary, in Category 2 we can see that the performance of NLC-TSC is poorer than that of NLC-OSC, even with relatively small H.

7 Conclusions

Two-stage communication-based schemes (five TSC schemes) are useful with relatively high hot degrees (δ) and a small number of hot processors (H). They work much better than one-stage communication-based schemes (five OSC schemes) when the size of a base array is much less than that of a source array. The performances of one-stage communication (direct algorithm) based schemes are significantly affected by H and δ, while those of TSC schemes are not. If there are few hot spots or the size of a base array is much greater than that of a source array, OSC schemes will be better than TSC schemes.

Even by adding local combining in the OSC schemes, the performance is significantly worse than those in the TSC schemes for almost all cases. Local combining schemes based on the TSC schemes improve the performance with relatively high hot degree (δ) and a small number of hot processors (H). The locality of hot pointers in source processors and locality of hot spots in target processors are main factors in the performance of local combining schemes. Also, if the data-transfer rate ($1/\mu$) in communication is relatively low and the cost for unit local computation is relatively small in a coarse-grained, distributed-memory parallel machine, as compared to the CM-5, the overhead entailed by local combining can be amortized more effectively by the decrease in time taken by communication.

If sufficient knowledge about the input index array, such as the existence and locality of hot spots and the locality of hot pointers is available, the performance evaluation presented in this paper will be a good guideline for deciding which scheme can be used effectively in applications requiring the array combining scatter primitive. On the other hand, in case no knowledge at all is available, either a one-stage or a two-stage algorithm may be automatically selected for global combining based on our modeling. In addition, a partial local combining scheme itself can be directly used as a decision system for local combining without any extra cost.

Acknowledgments

We are grateful to Minnesota Supercomputing Center for allowing us to use their CM-5. Also, we would like to thank Elaine Weinman for proofreading this paper. The work of Sanjay Ranka was supported by AFMC and ARPA under contracts F19628-94-C-0057 and WM-82738-K-19 and a subcontract from Syracuse University. The content of the information does not necessarily reflect the position or the policy of the United States government and no official endorsement should be inferred.

References

[1] Seungjo Bae. *Runtime Support for Unstructured Data Accesses on Coarse-Grained Distributed-Memory Parallel Machines.* PhD thesis, Syracuse University, August 1997.

[2] Seungjo Bae, Khaled A. Alsabti, and Sanjay Ranka. Performance Evaluation of Random Access Read/Write on Coarse-Grained Distributed Memory Parallel Machines. Technical report, School of Computer and Information Science, Syracuse University, September 1996.

[3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.

[4] Thinking Machines Corporation. *CMMD version 3.0 Reference manual*, May 1993.

[5] C. H. Koelbel et al. *The High Performance Fortran Handbook.* MIT Press, 1994.

[6] C. Leiserson et al. The network architecture of the Connection Machine CM-5. In *Proc. of 4th Annual ACM Symposium on Parallel Algorithms and Architectures, San Diego, CA*, 1994.

[7] Tilak Agerwala et al. IBM SP2 System Architecture. *IBM Systems Journal*, 34(2):152–184, 1995.

[8] High Performance Fortran Forum. *High Performance Fortran Language Specification: Version 1.1*, November 1994.

[9] V. Kumar, A. Grama, G. Karypis, and A. Gupta. *Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cummings Publishing Company, Inc., 1994.

[10] Wei-Keng Liao and Sanjay Ranka. Hough Transformation. Technical report, School of Computer and Information Science, Syracuse University, March 1996.

[11] H. Lu, B. Ooi, and K. Tan. *Query Processing in Parallel Relational Database Systems*. IEEE Computer Society Press, 1994.

[12] D. Nassimi and S. Sahni. Data Broadcasting in SIMD Computers. *IEEE Transactions on Computers*, C-30(2):101–107, 1981.

[13] Lionel M. Ni and Philip K. McKinley. A survey of wormhole routing techniques in direct networks. *IEEE Computer*, 26(2):62–76, 1993.

[14] V. K. Prasanna and C. Wang. Scalable Data Parallel Object Recognition Using Geometric Hashing on the CM-5. In *Proc. of Scalable High Performance Computing Conference*, 1994.

[15] Ravi V. Shankar, Khaled A. Alsabti, and Sanjay Ranka. The transportation primitive. In *Proc. of Frontiers 95*, 1995.

[16] Ravi V. Shankar and Sanjay Ranka. Random Data Accesses on a Coarse-Grained Parallel Machine II. One-to-Many and Many-to-one Mappings. *Journal of Parallel and Distributed Computing*, 44(1):24–34, July 1997.

[17] Ravi V. Shankar and Sanjay Ranka. Random Data Accesses on a Coarse-Grained Parallel Machine I. One-to-One Mappings. *Journal of Parallel and Distributed*, 44(1):14–23, July 1997.

[18] Marc Snir. The Communication Software and Parallel Environment of the IBM SP2. *IBM Systems Journal*, 34(2):205–221, 1995.

[19] Intel Corporation Supercomputer System Division. *Paragon XP/S Product Overview*, 1991.

[20] S. Tanimoto. *The Elementary of Artificial Intelligence Using Common LISP*. Freeman and Company, 1990.

[21] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proc. of ISCA 1992*, pages 256–266, 1992.

[22] J-C. Wang, T-H. Lin, and S. Ranka. Distributed Scheduling of Unstructured Collective Communication on the CM-5. In *Proc. of Hawaii International Conference on System Sciences*, 1993.

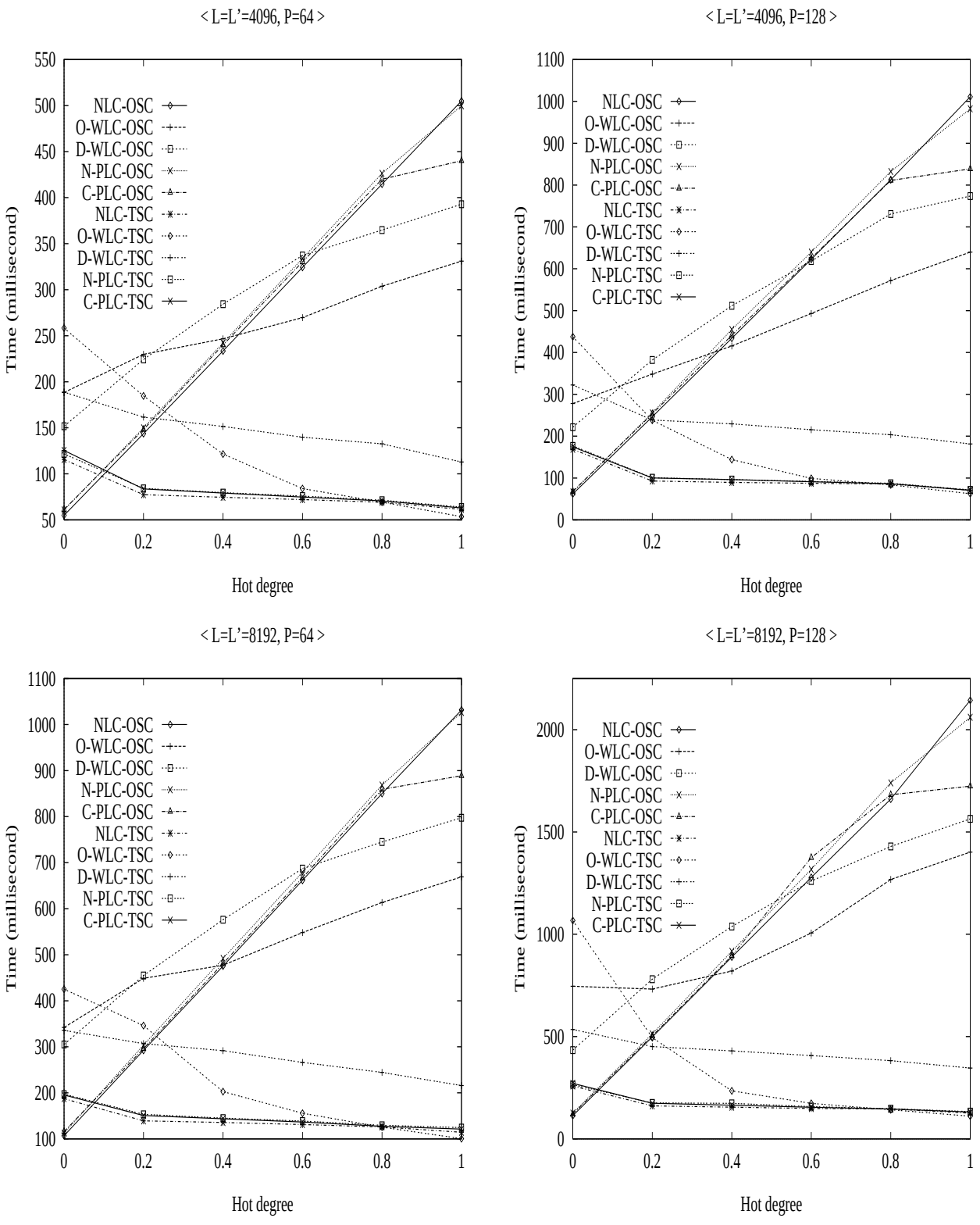

Figure 8: Total execution time (*millisecond*) for ten schemes for array combining scatter functions with $H = 1$ and $L = L'$

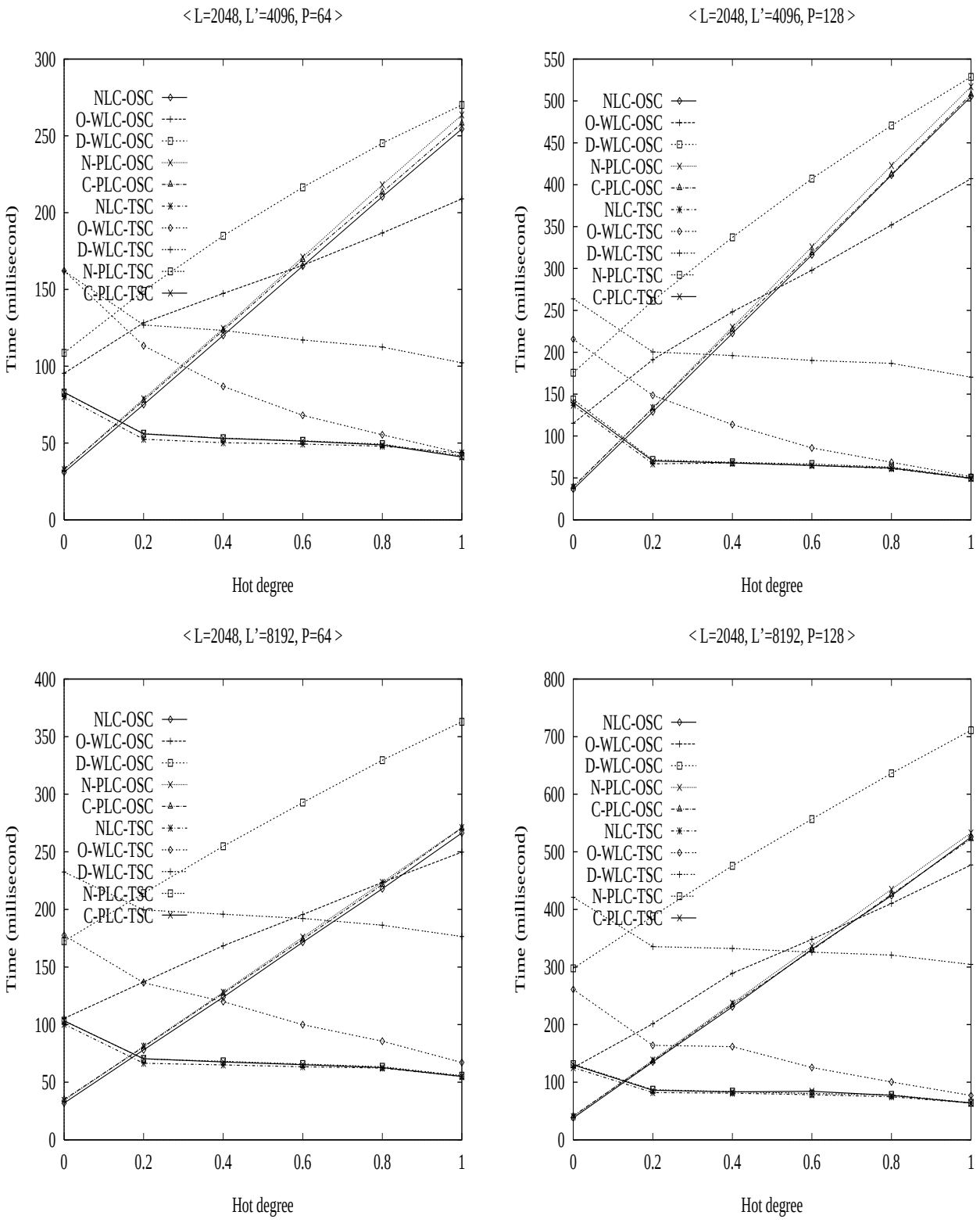

Figure 9: Total execution time (*millisecond*) for ten schemes for array combining scatter functions with $H = 1$ and $L < L'$

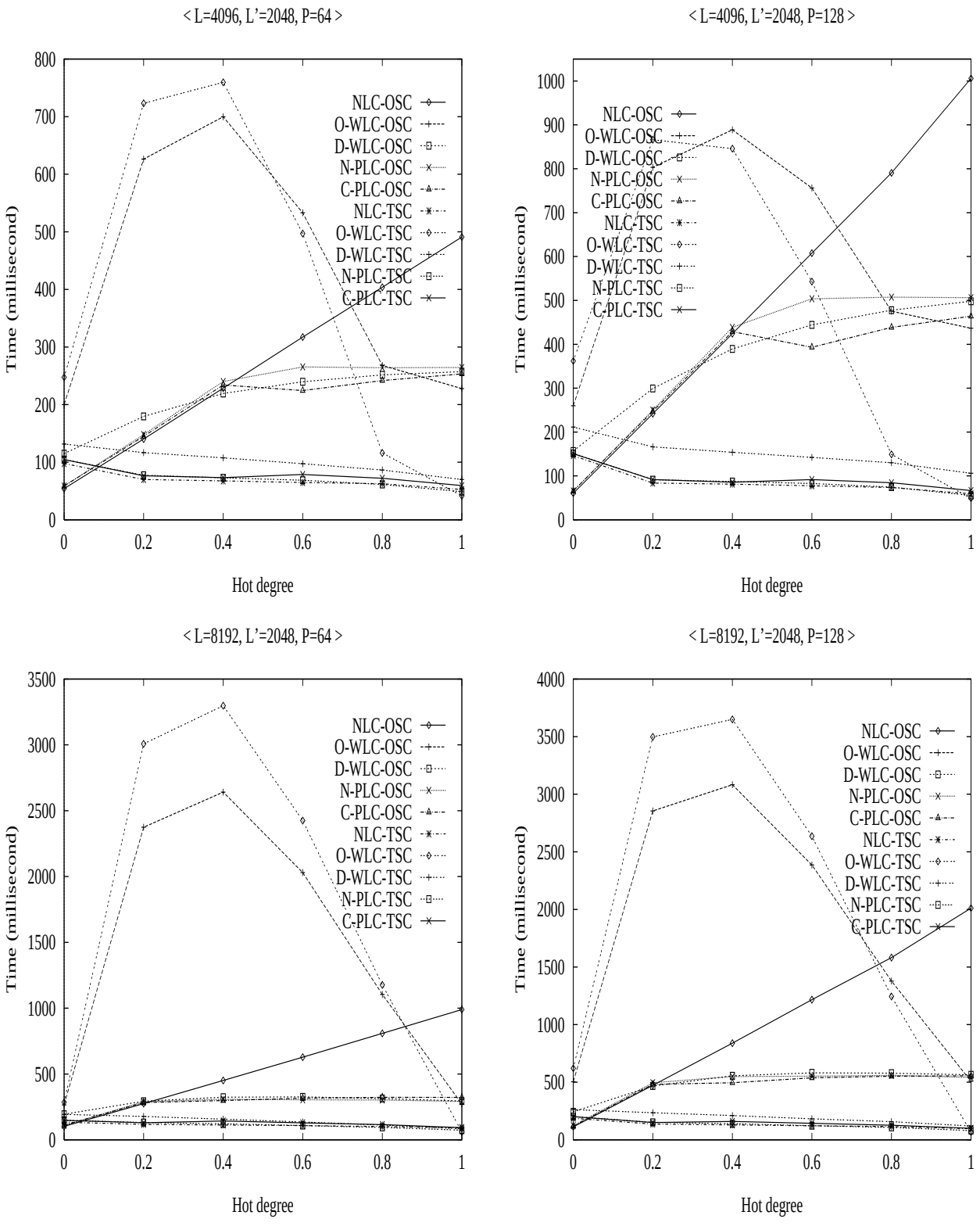

Figure 10: Total execution time (*millisecond*) for ten schemes for array combining scatter functions with $H = 1$ and $L > L'$

Category 1: $L = L'$

H	P	ω	(1024, 1024)	(2048, 2048)	(4096, 4096)	(8192, 8192)
1	64	ω_e	0.10	0.08	0.06	0.06
		ω_l	0.07	0.06	0.06	0.05
		ω_c	0.07	0.05	0.05	0.04
	128	ω_e	0.08	0.06	0.06	0.04
		ω_l	0.04	0.03	0.03	0.03
		ω_c	0.03	0.03	0.02	0.02
2	64	ω_e	0.20	0.16	0.14	0.10
		ω_l	0.14	0.13	0.12	0.11
		ω_c	0.10	0.10	0.08	0.08
	128	ω_e	0.18	0.12	0.10	0.08
		ω_l	0.08	0.07	0.06	0.06
		ω_c	0.06	0.05	0.04	0.04
4	64	ω_e	0.50	0.38	0.30	0.26
		ω_l	0.28	0.26	0.25	0.23
		ω_c	0.22	0.21	0.20	0.17
	128	ω_e	0.38	0.26	0.20	0.16
		ω_l	0.16	0.14	0.13	0.12
		ω_c	0.11	0.10	0.10	0.08

Category 2: $L < L'$

H	P	ω	(1024, 2048)	(1024, 4096)	(2048, 4096)	(2048, 8192)
1	64	ω_e	0.14	0.20	0.12	0.14
		ω_l	0.11	0.18	0.10	0.17
		ω_c	0.08	0.12	0.07	0.12
	128	ω_e	0.10	0.14	0.08	0.12
		ω_l	0.06	0.10	0.05	0.09
		ω_c	0.04	0.07	0.04	0.06
2	64	ω_e	0.32	0.42	0.24	0.36
		ω_l	0.22	0.37	0.20	0.35
		ω_c	0.17	0.24	0.14	0.23
	128	ω_e	0.22	0.30	0.16	0.22
		ω_l	0.12	0.21	0.11	0.18
		ω_c	0.09	0.13	0.07	0.12
4	64	ω_e	0.72	0.90	0.58	0.86
		ω_l	0.47	0.77	0.42	0.72
		ω_c	0.30	0.50	0.28	0.48
	128	ω_e	0.48	0.64	0.38	0.50
		ω_l	0.24	0.42	0.23	0.37
		ω_c	0.18	0.26	0.15	0.24

Table 5: Experimental and estimated threshold hot degrees, ω_e, ω_l and ω_c, with $H = 1, 2, 4$

Category 3: $L > L'$

H	P	ω	(2048, 1024)	(4096, 1024)	(4096, 2048)	(8192, 2048)
1	64	ω_e	0.06	0.04	0.04	0.04
		ω_l	0.04	0.03	0.04	0.03
		ω_c	0.03	0.02	0.03	0.02
	128	ω_e	0.06	0.04	0.04	0.02
		ω_l	0.02	0.02	0.02	0.01
		ω_c	0.02	0.01	0.02	0.01
2	64	ω_e	0.12	0.06	0.10	0.06
		ω_l	0.08	0.06	0.08	0.06
		ω_c	0.07	0.05	0.07	0.05
	128	ω_e	0.10	0.06	0.08	0.06
		ω_l	0.05	0.03	0.04	0.03
		ω_c	0.04	0.03	0.03	0.02
4	64	ω_e	0.28	0.16	0.20	0.12
		ω_l	0.18	0.12	0.16	0.11
		ω_c	0.14	0.10	0.14	0.10
	128	ω_e	0.20	0.12	0.16	0.10
		ω_l	0.09	0.06	0.08	0.06
		ω_c	0.08	0.05	0.07	0.05

Table 5: (*Cont.*) Experimental and estimated threshold hot degrees, ω_e, ω_l and ω_c, with $H = 1, 2, 4$