RP* : A Family of Order Preserving Scalable Distributed Data Structures

W. Litwin[1] , M-A Neimat[2], D. Schneider[2]

ABSTRACT

Hash-based scalable distributed data structures (SDDSs), like LH or DDH, for networks of interconnected computers (multicomputers) were shown to open new perspectives for file management. We propose a family of ordered SDDSs, called RP*, providing for ordered and dynamic files on multicomputers, and thus for more efficient processing of range queries and of ordered traversals of files. The basic algorithm termed RP*$_N$, builds the file with the same key space range partitioning as a B-tree, but avoids indexes through the use of multicast. The algorithms, RP*$_C$ and RP*$_S$ enhance the throughput for faster networks, adding the indexes on clients, or on clients and servers, decreasing or avoiding the multicast. RP* files are shown highly efficient with access performance exceeding traditional files by an order of magnitude or two, and, for non-range queries, very close to LH*.*

Keywords:

Scalable Distributed Data Structures, Network files, Ordered files, Multicomputers, Multicast, Broadcast

1. INTRODUCTION

An increased amount of research has been devoted to the efficient use of mass produced computers interconnected through high speed networks. The computers (sites) are assumed to be PCs or workstations. The networks typically have bandwith of 10 Mb/s — 1 Gb/s : Ethernet, 100 Mb/s Ethernet, Token Ring, FFDI, ATM,... Such configurations are prevalent in many organizations. They typically consist of many computers, thousands in larger organizations. Cumulative processing and storage resources of such configurations are impressive, and often exceed those of a supercomputer. HP Labs today has 1300 interconnected workstations with a total of 32 GB of RAM for applications, and terabytes (TB) of disks. The UC Berkeley Soda Hall multicomputer should include 500 workstations, delivering 25 Gflops, 64 GB of RAM, and about 1 TB of disk storage [C94]. Many organizations have similar configurations.

At present, cooperation between interconnected computers is limited. Typically, an application uses the resources of a single computer, or perhaps, this computer is a client to some server. Many researchers believe that cooperation between interconnected computers should be much more extensive. An application should be able to use storage and processors of other computers whenever it needs, the number of computers involved being transparent. Terms are being coined for computers organized this way: *multicomputer*, *network computer*, and *distributed memory* are especially popular terms [ILP93], [BZS93]. Multicomputers should be an attractive alternative architecture to dedicated MPPs, and supercomputers, offering much better price-performance ratio [C94].

---

[1] University Paris 9, visiting HP Laboratories, Palo Alto CA & UC Berkeley
[2] HP Laboratories, Palo Alto CA.

A fundamental component of a multicomputer will be data structures, especially for files. Traditional data structures will not suffice, as multicomputers allow for distributed and parallel processing, distributed RAMs allow for very large files that otherwise had to be on disks, distributed disk files can be extremely large, and scalability is a basic requirement. Also, parallel data structures as defined for supercomputers will typically not be adequate [LNS93]. One class of data structures that is defined specifically for multicomputers are Scalable Distributed Data Structures (SDDSs) [LNS93]. An SDDS stores data on some sites called *servers,* and is used from some sites called *clients*, basically distinct, but not necessarily. Clients typically retain some file access computation parameters, e.g., parameters of the actual dynamic function in LH* or DDH, but not the actual records [LNS93], [D93]. These parameters create the *client's image* of the actual file.

Every SDDS must respect three design requirements. First, no central directory is used for data addressing, to avoid a hot-spot. Next, client's images can be outdated, being updated only through messages, called Image Adjustment Messages (IAMs). These are sent only when a client makes an addressing error. Client's autonomy is preserved in this way which is a major requirement for multicomputers [G88a]. Finally, a client with an outdated image can send a key to an incorrect address, in which case the structure should deliver it to the right server, and trigger an IAM.

Only hash-based SDDSs have been proposed: LH* [LNS93], [LNS93a], DDH [D93], and variants of LH* in [VBWY94], and [LNS94b]. All generalize popular extensible hashing algorithms. A file can start on one computer and scale up to practically any number of computers, and any size. It can be manipulated by any number of distributed clients, and supports parallel queries. It was shown that such files can be much faster, and larger than traditional files.

Hashing is not order-preserving. If the file is to support range queries, and ordered traversals, then traditional ordered data structures, e.g., B-trees, are faster than traditional hashing structures. Ordered structures partition the range of keys in the file in a way that a key and its successor are typically in the same bucket (page). A range query then needs to search only a small subset of the file, instead of typically all the hashed file. An ordered traversal visits each bucket (page) of the file only once.

Hash-based SDDSs are much more efficient than traditional hash files for range queries, and general multikey queries, through parallel processing [LNS93]. One can nevertheless still expect gains from an order preserving SDDS. If a range query retrieves for instance a hundred records, they could easily be in the same bucket in a large order preserving SDDS, being then delivered with a single message. A hashed SDDS would scatter them into different buckets, requiring a hundred reply messages instead. Latency delays would make this delivery typically much longer than that of the single, although longer, message.

We propose a family of order preserving, range partitioning, SDDSs called RP*. It consists of three structures, called RP*$_N$, RP*$_C$ and RP*$_S$. Their mutual relationship is illustrated in Fig. 1.

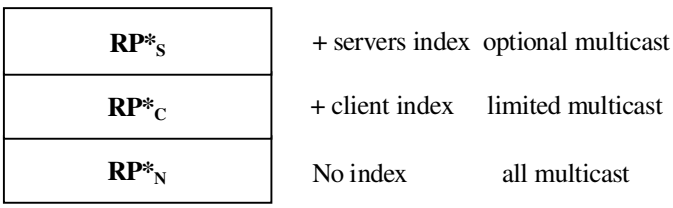

Fig. 1 RP* design trade-offs

RP*$_N$ creates the same range partitioning as a B$^+$-tree, but without any index. This property is due to the use of multicast messages available on most local nets, ATMs, and on wireless nets. RP*$_C$ file is an

RP*$_N$ file plus indexes built on clients through IAMs. The indexes allow for point-to-point messaging of key searches, inserts and deletes, while multicast remains used to resolve addressing errors. The major gain is improved throughput, especially on gigabit nets. Finally, RP*$_S$ adds indexes on the servers that further improve the throughput. Every query can be also processed using point-to-point messages only.

As we show, the load factor of an RP* file is the same as for a B^+ - tree under the same operations. The file can scale up to thousands sites, allowing, e.g., for a 32 GB RAM file on the HPL domain. Average messaging costs are about minimal ones : one message per update, and two per key search. Elapsed times are in order of millisecond (ms) on a 10 Mb/s net, and under 100 µs on a 1 Gb/s net, assuming a 100 MIPS CPU. Traditional files are not as large and fast, hence RP* schemes are highly promising.

With respect to the related work, there was no proposal of distributed ordered file structure without index. There were several proposals for parallel distributed B-trees, e.g., [MS90], [MS91], [JK93], but none defined an SDDS. The indexes existed only on the servers and a much more limited scalability would result from, if these proposals were applied to multicomputers.

Section 2 presents RP*$_N$, and Section 3 discusses its performance. Sections 4 and 5 describe RP*$_C$ and RP*$_S$. Section 6 compares the algorithms, and relates them to LH*. Section 7 concludes the paper.

2. RP*$_N$

2.1 Network characteristics

We consider a typical local network as in Fig. 2. A cable called *segment* connects a few, or hundreds of *sites*, hooked to the cable through *controllers*. Segments are linked through *routers*, recognizing non-local message addresses, and forwarding such messages. Site send *messages*, usually broke into *packets*, e.g., of at most 1500 bytes for Ethernet. Every message is listened to by every controller on the segment. A message can be of the following type:

- Point-to-point message. Such messages are retained by a single controller and delivered to its site.

- Multicast message. Such messages have an address recognized by several but typically not all controllers. They are retained only by these controllers. Other sites are not interrupted by their controllers. A typical controller can carry a dozen multicast addresses.

- Broadcast message. Such messages have an address recognized by all controllers of the net, and so are delivered to all the sites.

Broadcast messages are the basic type of message on wireless networks. As broadcast can be seen as the basic case of multicast, we will not name it separately in what follows.

A multicast message traverses a network segment in the same time as a point-to-point message. It can deliver an operation to several servers in a fraction of the time that would be needed if point-to-point messages were used. Multicast is most efficient when an operation should be performed in parallel on all the servers sharing a multicast address. It may even be preferable when the operation is destined to only some servers, the filtering of irrelevant sites being performed at the application level. However, as it interrupts several sites, such use of multicast may impair the throughput with respect to point-to-point messaging. The trade-off depends on the application.

A segment allows for one message at any time, but different segments can serve their sites in parallel, as long as messages at each segment address only sites local to the segment. An inter-segment message becomes a new message on each segment it traverse. Hence, a message traversing k segments, takes time of at least k messages. The network topology is usually chosen to make $k = 2,3$. Even for a thousand site local net a few routers suffice to directly connect most pairs of segments.

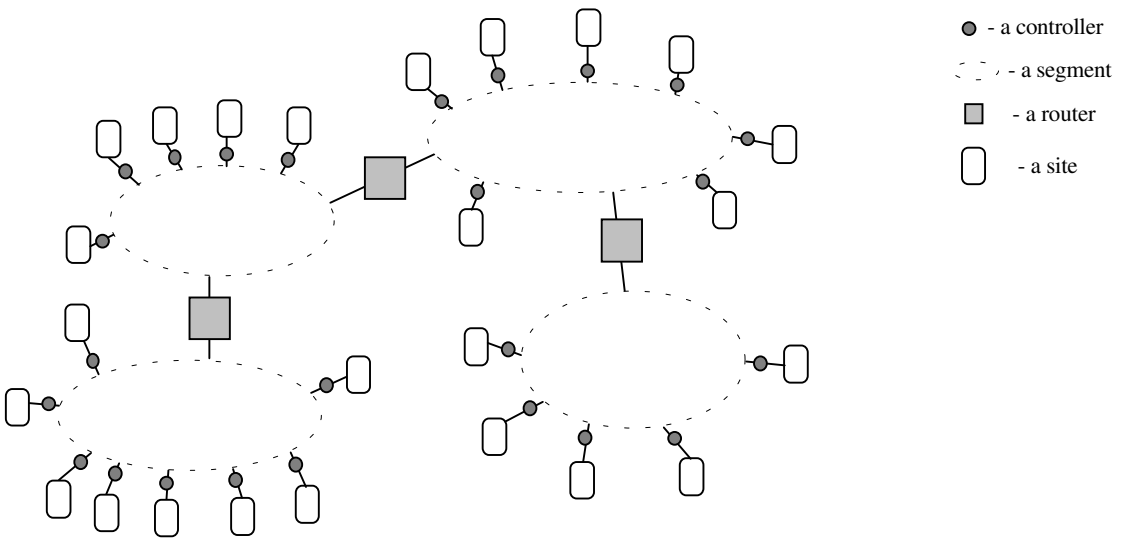

Fig. 2 A local network

2.2 File structure and creation

We consider records consisting of a key and of non-key fields. The key identifies the record and draws its value from a large key space with a total order. Records are grouped into buckets (pages, blocks...) with capacity of b records, $b >> 1$. Every bucket is at a different *server* (site). Logically, records in a bucket are in ascending order of keys. The internal organization of the sequence is not of concern here. Every bucket has a *header* with two values called respectively *minimal* and *maximal* keys, noted λ and Λ. The interval $(\lambda, \Lambda]$ is called the *range* of the corresponding bucket. A bucket can only contain a key c in its range, i.e., $\lambda < c \leq \Lambda$.

An RP*$_N$ file initially consists of a single bucket, named bucket 0, with $\lambda = -\infty$ and $\Lambda = \infty$. A multicast address can be assigned to the bucket, as well as to all potential sites for the file. All initial insertions go to bucket 0. When it overflows, it splits and a new bucket is created, termed bucket 1. In general, the new bucket is numbered bucket M if the file already has M buckets. It is assumed that some easy translation exists between bucket numbers and physical site addresses, e.g., as for LH* [LNS93].

Fig 3 illustrates the creation of an RP* file with $b = 4$. The file undergoes three splits, under the insertions of the most commonly used English words [LRLH91]. The RP*$_N$ split algorithm is as follows:

RP*$_N$ split algorithm:

1. Determine (as for a B-tree) the middle key c_m in the overflowing bucket B.
2. Attempt the creation of bucket M. Wait for ack, or denial if bucket M exists already.
3. If creation is denied, then $M \leftarrow M + 1$; go to Step 2.
4. Copy to bucket M the following content:
   - The header with :
     $\lambda \leftarrow c_m(B)$;
     $\Lambda \leftarrow \Lambda(B)$;
   - every record from B with $c > c_m$.
5. Decrease the maximal key in B:
   $\Lambda \leftarrow c_m(B)$;
   and remove records moved to bucket M.
6. Set $M \leftarrow M + 1$.

Each split refines the partition of the key space, and of the file, as it partitions the range of the bucket undergoing the split into two ranges and two buckets. The scheme obviously produces an ordered file. It also fulfills the axiom that a bucket contains only the keys in its range.

The ack message in Step 2 is a multicast message. Every bucket then increments *M* as in Step 6. The creation of bucket *M* may fail, when two buckets split concurrently. Steps 2 - 5 can be performed in one or more messages, and are discussed in more detail in Section 3.2. Step 2 may also propagate the multicast address of the file. In Step 5, bucket *B* waits for an ack from bucket *M*.

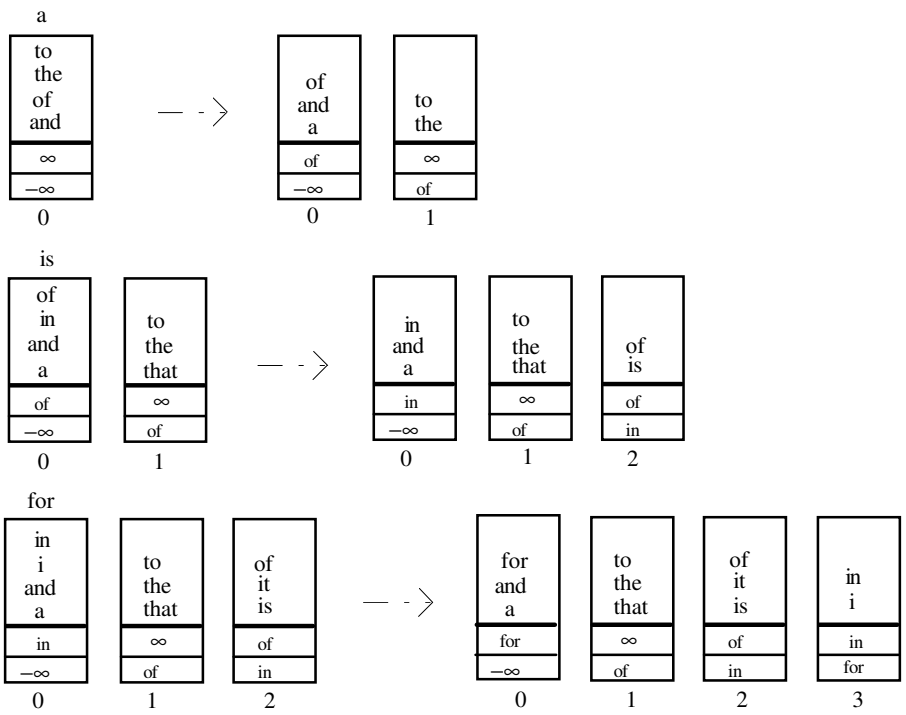

Fig. 3 RP* file expansion

2.3 File access

The file is manipulated through following *queries* issued by *client* sites. A (single) *key query* is a key search e.g., **get** 'that' in Fig. 3, or an exact match update : **insert** 'it', **delete** 'it', **modify** 'that' <update specs>, where the <update specs> concern a non-key field of the record, if there are any. A query can be also:

- a *range query* : a search of every key *c* within some range $[c_1, c_2]$; $c_1 < c_2$; or an update of non-key fields in records within the range.
- a *general query* which is a search for, or update of, every record fulfilling a condition on non-key fields of the record, and of the record only (hence joins, etc. are excluded).

A client can also perform an (ordered) **traversal** of the file, or of its part. A *traversal* consists of a sequential examination of every record in the file, or within some range, in ascending (descending) order. The client requests the first record, then issues **get_next** queries.

In traditional data structures, a key search or an insert is an operation on a single record. The availability of multicast allows RP* to also offer *bulk queries*, called *b-search, b-insert, b-delete*. Every bulk query concerns a bulk (set) of keys, or records, sent within the query, perhaps into different buckets, e.g., **insert** ('this', 'these', 'its'). Bulk operations may provide for better performance when the client has a

collection of keys to insert or search. Application examples are : file restructuring, batched updates, sorts, and joins.

2.4 Algorithms

2.4.1 Search and update

In RP*$_N$, every query is sent out using a multicast message with the file address. Hence it is received by every bucket. A server requested to insert key c, performs the operation iff $\lambda < c \leq \Lambda$. The split algorithm partitions the key space, hence every c ends up in exactly one bucket. Deletions and updates work in a similar way. For the client, an update terminates when sent out, unless the client specifically requests an ack.

A bucket receiving a search for c, searches through its records iff c is within its range. Only such a bucket replies, with its range and the record if found. The client terminates when it receives the reply. If the range alone is received, the search is an *unsuccessful search* for c. Every reply is a point-to-point message.

Ex. Consider the final file in Fig. 3. The search **get** 'this' would be multicast to all four buckets. There would be one reply : the range (of, ∞]. This means for the client an unsuccessful search. The update **insert** 'this', would be multicasted to all buckets. It would only enter the bucket with (of, ∞] range.

2.4.2 Range and general queries

A bucket receiving a range or general update updates all the relevant records. No ack is sent, unless required. A bucket receiving a range search replies iff its range overlaps the query range. It sends the selected records, if any, and its range. The client terminates the search when the union of received ranges covers the query range. Every key to be selected, must be in a bucket with one of received ranges.

The primary strategy for a general search is that a bucket replies iff it finds qualifying records. The client has a time-out t reinitialized after each reply, to a value making the probability of a missing reply negligible, e.g., 3 — 5 times the expected time of reply from one bucket. The client terminates the search when t expires.

The correctness of the result is only probabilistic. The strategy providing a time-independent deterministic termination is that every bucket replies with its range. The client terminates when the union of ranges reaches (-∞,∞].

2.4.3 Traversals

Buckets B_1 and B_2 are called left and right siblings iff $\Lambda(B_1) = \lambda(B_2)$. The right sibling is the successor of the left sibling with respect to the order on the key space.

A traversal is processed through the delivery to the client of one bucket at a time. To find the sibling, RP*$_N$ supports searches for buckets with $\lambda = c$, or with $\Lambda = c$. The client can set c to Λ or to λ of the bucket it has just visited, or to $\Lambda = \infty$ to traverse the file starting from its last bucket. Finally, to locate the starting bucket of the traversal in some range $[c_1, c_2]$, the file supports the query retrieving the (single) bucket with $\lambda < c \leq \Lambda$.

2.5 File contraction

As in a B-tree, the *merge* operation is the inverse of splitting, shrinking an RP* file that underwent many deletions. As files rarely shrink in practice, we assume no merges in what follows. See [LNS93b] and [LNS94] for the discussion of the subject.

3. RP*$_N$ PERFORMANCE

3.1 Performance measures

As usually, the basic performance factors for an SDDS are the load factor, and the access performance, measured in number of messages per operation The messaging costs are independent of the network and

CPU performance. When those factors are taken into account, one can compute elapsed times and throughputs.

For SDDSs, specific performance measures were introduced [LNS93]. Client image size is one concern. Also, access performance may vary between clients, because of different images and numbers of IAMs received. The main corresponding measures are :
- search cost of a new client starting with the initial image, i. e., $M = 1$, to access an existing static file, with $M >> 1$.
- the number of addressing errors for a new client till its image converges to the actual state of the file.
- the messaging costs of an addressing error, including the cost of the IAM.
- insert costs for each of m clients inserting random records at the respective activity rates of a_1 , a_2..,1. Basically, $m = 2$, and a_1 is set to $a_1 = 1, 10,...$ Less active clients get more IAMs, so their performance can be affected.

In what follows will address these measures for each RP* structure, with the overall comparison in Section 6.

3.2 Basic characteristics

RP*$_N$ splits generate the same partition of the key space, and of records into buckets, as a B$^+$-tree file with the same bucket size and under the same operations. Hence the load factor x / bM of an RP*$_N$ file is on the average about 70% for x random insertions, and is 50 % for x sorted insertions. However, RP*$_N$ does not need space for an index.

A particularity of RP*$_N$ as an SDDS is that there is no image of the file at a client site, so there are no addressing errors and no IAMs. Hence, all the associated specific costs are equal to zero.

An update of a record costs one multicast message. A key search costs two messages : a multicast, and a point-to-point reply. A range query selecting k keys requires one multicast, and triggers replies from buckets whose union of ranges covers its range. Assuming random inserts, there are $n \leq$ Min ($\lceil k / 0.7\, b + 1 \rceil$, M) such buckets (the constant 1 is due to the fact that if even two selected keys can be in different buckets). Hence the query requires at most n+1 messages. Finally, a non-key query selecting k records requires at most Min (k, M) + 1 messages, for the strategy with time-out, and M + 1 messages for the exhaustive one.

A split typically requires four messages. In the worst case, this cost is higher by 2 f messages, where f is the number of failed attempts to create bucket M in Step 2. However, this case should be unlikely as splits should be infrequent. In practice, an average cost of an insert during a file buildup using random keys, should be: $1 + 4 / 0.7\, b$ messages, i.e. one message.

All the message costs are independent of M, at least as long as one can ignore network topology. This is the case of a typical local net, as we discussed in section 2.1. Furthermore, the storage required at a client is also constant, as there is no index. An RP*$_N$ file can thus scale up to every site on local net with constant load and messaging cost performance. A RAM bucket at each site can be as large as RAM available for the applications, e.g., can be 32 MB on typical workstations. An RP*$_N$ RAM file can thus potentially scale up to gigabytes, e.g. to 32 GB at HPL or Soda Hall configurations outlined in Section 1. Such a file is much larger than a traditional RAM file, and larger than many traditional disk files. Terabyte files are possible if buckets are on disks on these sites, at the price of increased file access times.

3.3 Elapsed times

The elapsed time to service a query to a RP*$_N$ file, and the reply if any, consists of the time to traverse the network, to process the message by the servers' and client OSs, and to process the RP*$_N$ bucket(s). The elapsed time of a key query is independent of the file size M, as the cost of multicasting to M servers is equal to that of a point-to-point message to the bucket with the key, and bucket processing is executed in parallel on all servers. The elapsed time of range and general queries increases at least linearly with the number of qualifying buckets.

Table 1 shows elapsed times that should characterize RP*$_N$ files on popular nets, generically termed *m-net* (10 Mb/s), *h*-net (100 Mb/s), and *g-net* (1 Gb/s). Discussion of underlying formulas is in [LNS94]. The network model is from [G88]. Site speed is accordingly assumed 100 MIPS, and OS time to process a message is 25 µs (2500 instructions). A query message with key(s) is assumed 100 Byte long, except for bulk operations, and a message with a record, a reply, or an insert, is assumed 1 KB long. We further assume that buckets are in RAM, a bucket range check costs 1 µs, an in-bucket search costs 5 µs, an update of bucket costs 25 µs, and an in-bucket processing of a general query on non-key attributes costs 250 µs (25000 instructions). Range and general queries select 10 records, i.e., only a few, as in TPC-C benchmark. The table shows elapsed times for an insert (t_i), a search (t_s), a range query (t_r), a general query (t_g), and a bulk insert of 1000 keys ($t_{b\text{-}i}$). There are also the times to traverse the network, by an insert ($i_{i,\,t}$), and a search ($t_{s,\,t}$), computed according to [G88]. These are absolute bounds on any algorithm performing these inserts or searches, regardless of its own efficiency.

The elapsed times for search and insert are about 1 ms for an m-net, and under 100 µs for g-net. At least one or two orders of magnitude are gained over traditional disk files of similar sizes. Similar or greater performance gains characterize range and general queries. Bulk inserts performance is dominated by transfer time. The operation is most useful on g-nets, where, in our case, the bulk insert takes seven times less than the corresponding 1000 inserts. The time of a bulk operation can decrease further, if messages are compressed, as a longer message may achieve a much higher compression ratio. In general, for slower nets, elapsed times are dominated by the transfer times. So, little can be gained through improvement to query processing speed on sites with respect to RP*$_N$ performance.

	m-net	**h-net**	**g-net**
	10 Mb/s	100 Mb/s	1 Gb/s
t_i	1.061 ms	161 µs	71 µs
t_s	1.176 ms	186 µs	87 µs
t_r	10.141 ms	1.061 ms	152 µs
t_g	15.585 ms	1.555 ms	585 µs
$t_{b\text{-}i}$	1010 ms	100.06 ms	10.07 ms
$t_{i,\,t}$	1.010 ms	110 µs	20 µs
$t_{s,\,t}$	1.120 ms	130 µs	31µs

Table 1. Elapsed times for an RP*$_N$ file

3.4 Throughput

The throughput s_o is the number of operations *o* per second (o/s) that one can perform at best. Table 2 shows throughputs of inserts (s_i), and of searches (s_s) in an RP*$_N$ file with $M >> 1$. The formulas are in [LNS94]. The table also contains the corresponding network throughputs ($s_{i,\,t}$), and ($s_{s,\,t}$) resulting from the times $t_{i,\,t}$ and $t_{s,\,t}$, and from the OS time above. Finally, we show the percentage of CPU bandwidth that RP*$_N$ should use on a server.

The values of s_i and of s_s show that RP*$_N$ can sustain about 1000 o/s on m-net, up to 22,000 on g-net. These figures assume the use of 100 % of network bandwidth, hence practical figures should be somehow lower [G88]. Notwithstanding, these RP*$_N$ capabilities are above the requirements of most current applications, especially for performance on h-nets and g-nets.

The values of $s_{i,\,t}$ and of $s_{s,\,t}$ on m-net show that only 2 - 3 % gain in throughput can result from optimization of RP*$_N$ implementation on a site, e.g., faster in-bucket operations. The network throughput is an absolute bound on throughput of any algorithm over the net. The RP*$_N$ has an almost optimal messaging cost. Hence, no other algorithm one could ever invent for the range partitioning may be substantially faster than RP*$_N$ on an m-net.

The 2 — 3 % value of $\%_{CPU}$ show for m-net that RP*$_N$ files should be very acceptable at a site, as 97 % of CPU bandwidth remains available for other typical needs of a multicomputer. RP*$_N$ load should be also acceptable at h-net, especially since the 19 % in the table correspond to 7000 o/s. For a g-net however, the use of RP*$_N$ file at full throughput may load the site too much in the multicomputer context. The reason is that OS at all servers of a RP*$_N$ file must process every query. This overhead reduces the throughput potential of each server, and hence of the multicomputer.

The ratios $s_i\,/s_{s\,,\,t}$ and $s_i\,/s_{s,\,t}$ show for a g-net that an algorithm for range partitioning improving a site throughput has room to about double it with respect to RP*$_N$. In the next two sections, we introduce two algorithms which reduce, and eliminate the use of multicast to attain this goal.

	m-net	**h-net**	**g-net**
b_n	10 Mb/s	100 Mb/s	1 Gb/s
s_i	965 o/s	7352 o/s	21739 o/s
$s_{i,\,t}$	990 o/s	9991 o/s	50000 o/s
$\%_{CPU}$	3 %	19 %	57 %
s_s	872 o/s	6410 o/s	17544 o/s
$s_{s,\,t}$	893 o/s	7692 o/s	32258 o/s
$\%_{CPU}$	2 %	17 %	45 %

Table 2. Throughputs of an RP*$_N$ file

4. RP*$_C$

The algorithm RP*$_C$ creates an image of the file on the client. The image is a collection of bucket ranges and addresses. The client issues a point-to-point query whenever the key to search or update is in a range in its image. Otherwise, the client uses multicast. Multicast is also used by a server that receives a point-to-point query with the key out of the range, because of an outdated image of the server on a client.

4.1 Image structure and evolution

The image is a dynamic table T [0,1...]. Initially, T has no elements. Every element $T(i)$ encodes an address of a bucket and its range. One form of T is an ordered list of tuples $t = (A, C)$, as in Fig. 4. Its structure is similar to that of a single B$^+$ - tree node. Here, A is either the address of bucket A and C is $\Lambda\,(A)$, or A is null, denoted '*', and C is $\lambda\,(A')$ where A' is the address that immediately follows C in T. Initially, T consists of $t = (0, \infty]$, although this C does not need to be materialized in practice. The search for key c is performed as follows. Updates work similarly.

1. The client searches for $t \in T$ with the smallest $C \geq c$. If $A\,(t) \neq$ '*', then it sends c to bucket A using a point-to-point query. Otherwise, it multicasts the search.

2. A bucket receiving a search for c tests whether c fits its range. If not, and the search is a multicast, it terminates. If the search is a point-to-point query, then it multicasts the query, including its range and address, let it be A, in the message, and terminates.

3. If c is in the range, then the server performs the search, sends back record c if found, its own range, and address. If the query was a multicast from some bucket A, then it also adds the range of A to the reply (it was included in the message from A). It then terminates.

4. The client receiving a reply to a multicast query, finds a range and an address in the reply, let it be (λ, a, Λ). This is an IAM. A reply to a point-to-point query may carry two such triplets, which means it is an IAM as well. In both cases, the client updates T in th following cases, illustrated in Fig. 4. For each IAM, if there is no t with $C(t) = \lambda$, and $\lambda \neq -\infty$, then it adds $(*, \lambda)$ to T. Next, if there is $t = (a, C)$ with $C > \Lambda$, then if $C = \infty$, then the client sets t to (a, Λ) and adds $(*, \infty)$ to T, otherwise, for $C < \infty$, it only sets t to (a, Λ). Next, if there is $t = (*, \Lambda)$ in T, then it sets t to (a, Λ). Finally, if there is no (a, Λ) in T, then (a, Λ) is added to T.

Details of Step 4 are straightforward from Fig. 4. The figures illustrates the creation of T for the client searching the file in Fig. 3. The search for key 'it', is sent to bucket 0 which multicasts it. Bucket 2 gets the query, processes it, and replies with its range and address, and range and address of bucket 0. This leads to T_1. Then, **get** 'that' is multicast, and, as a result, returns the range of bucket 1, leading to T_2. Finally, **get** '**in**' is multicast, and leads to T_3 which contains the actual image of the file. From now on, as long as there are no splits, the client will only use point-to-point messages for key searches and key updates.

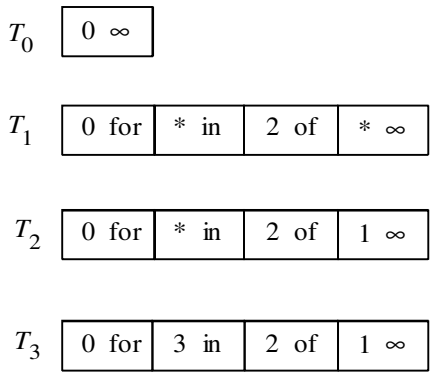

Fig. 4 Evolution of RP*$_c$ client image resulting from searches for keys
· **it, that, in** in the file in Fig. 3.

For every static file, an image T can acquire all the ranges in the file, provided the searches probe every bucket, e.g., through sufficiently many random searches in practice [LNS94]. The client then no longer multicasts key searches and updates. Splits progressively outdate existing images, triggering IAMs and multicasts. Range and general queries, and bulk operations remain multicasted, and do not trigger IAMs.

4.2 Performance

IAMs cause T to grow to M elements, i.e., $O[M(l_a + l_c)]$ bytes, where l denotes byte sizes of a key and of an address. Assuming $l_a = 4$, and $l_c = 40$, for an alphanumeric key, a thousand bucket file requires 44 Kbytes. Such storage requirements are easy to meet.

Once T has the exact image of the file, the client uses only point-to-point messages for searches and updates. If buckets are chosen randomly, the overall OS throughput s_n increases M times to $s_n = M / t_n$; t_n = 25 µs in our case [G88]. For a larger M, the throughputs s_i and s_s approach to the bounds $s_{i,t}$ and $s_{s,t}$, possibly leading to substantially better performance for g-nets, as in Table 2.

However, addressing errors deteriorate the access costs. In the worst case, a key search or update may need three messages, including one multicast, instead of one or two messages for RP*$_N$. A client building a file of M buckets, will make M -1 addressing errors, with on the average one error every 0.7 b (random) inserts. Additional messages impact accordingly the average costs, especially the insert cost for smaller b's. For $b >> 1$, the average insert cost remains one message in practice. The average search in this file, by the client that built it up, should similarly cost two accesses, as the client's image will be about accurate.

A new client performing searches will also make M -1 addressing errors. The number of messages per search will be the same as for a client with an accurate image, but M -1 messages will be multicasts, instead of point-to-point messages. When the client starts, almost every search will be a multicast, then, more and more will use point-to-point queries. Inserts of a new client will lead to at least M -1 addressing errors, with three messages per insert in each case, including one multicast. For searches when T is about empty, almost every search or insert triggers a multicast. It is easy to see from Table 2 that this may noticeably decrease the throughput of RP*$_C$ file on a g-net, to that of RP*$_N$ file for searches, and even under that for inserts.

If m clients build the file jointly to size M, each will make up to M -1 addressing errors, triggering up to m times more errors than a single builder would encounter. The average performance of each client will be affected with respect to the case of a single one. Slower clients will be more affected, as they will make fewer inserts for possibly the same number of addressing errors.

The numerical values of average search and insert costs of an RP*$_C$ file, were determined through simulations. These results are discussed in Section 6, comparatively to those of other RP* algorithms and LH*.

5. RP*$_S$

This algorithm eliminates multicast for key search and updates and speeds up the convergence of client images. Range and general queries can be multicasted or sent using point-to-point messages. This allows for an ordered SDDS on nets that do not support the multicast.

RP*$_S$ is RP*$_C$ plus an index on servers. This index, called *kernel* materializes the actual ranges. It is built by the servers, and is transparent to clients.

5.1 Kernel structure

The kernel structure is similar to a B$^+$-tree file structure. Fig. 5.a, shows a 2-level RP*$_S$ file and Fig. 5.b shows a 3-level one. Kernel's buckets called *nodes* form an m-ary tree whose leaves are file buckets. Every node may contain up to $m >> 1$ keys called *separators* ordered by key values, and pointers between them. The pointers are downward pointers to lower level buckets. Every two successive separators define the range of the bucket or node pointed to by the pointer between them. The last separator of a node does not need to be materialized inside the node, since it is in the header. Every header contains a range and a *parent pointer* that is upward pointer to the parent node (parent pointers do not exist in B-trees). These pointers are used when addressing errors occur. Nodes can be at sites distinct from those of buckets, or they can share these sites. We assume that nodes are on distinct sites.

Even for a large RP*$_S$ file, the tree can be 2-level only, when the value of m is in the hundreds or thousands. It might however, be more appropriate to distribute the separators into smaller nodes, leading to a 3-level tree. Higher kernels seem less likely.

Below, bucket addresses are denoted 0, 1... while nodes addresses are denoted *a, b*...

5.2 File evolution

Splits are carried on as in RP*$_N$, except for what follows. Let c_m denote the middle key in a bucket or node. The first split of bucket 0 creates the first node, let it be node *a*. Node *a* becomes the root of 2-level tree, with the range (-∞,∞] and the parent pointer '*'. Bucket 0 inserts into node *a* the triplet (0, c_m , 1). Every further split sends to node *a* the tuple (c_m, *r*), where *r* is the address of the right sibling the split creates. The split also forwards the parent pointer *a* to bucket *r*. When node *a* is full, it splits, creates a sibling *b*, and some node *c* that becomes the new root of a 3-level tree, with the range (-∞,∞], and the parent pointer '*'. Node *a* inserts into node *c* the triplet (*a*, c_m, *b*). Further splits insert into node *b* the tuples (c_m, *r*), etc.

Fig. 5.a shows node *a* created by the evolution of the file in Fig. 3. It is further assumed that $m = 3$, and that the file undergoes inserts of keys 'this' and 'these'. This triggers the overflow of node *a* and the creation of sibling *b* and of new root *c*, as in Fig. 5.b.

The RP*$_S$ splitting algorithm can obviously use multicast as RP*$_N$. It is also possible to eliminate it, e.g., through getting the address of new bucket from a split coordinator like for LH*.

5.3 File access

The clients performing a key query computes the address for the key *c* from its image. A leaf that receives a key search or an update with *c* in its range, performs the operation. If *c* is out of range, then the bucket forwards *c* to its parent. If *c* is in the parent's range, it forwards *c* to the offspring with *c* in the range. Otherwise, it forwards *c* to its parent, etc., untill either a leaf or the root is reached. Once the root is reached, the query follows the downward path as in a B$^+$ - tree. The final leaf sends the IAM to the client with the sub-tree visited by *c* (nodes and downward pointers, as parent pointers can be outdated, as it will appear). The sub-tree is called an *IA-tree*. The IAM can be piggy-backed on the result of the search, as for RP*$_C$.

As will be shown later, a client can also send *c* to a node, instead of a leaf. This is also an addressing error, occurring when the client can determine from its image that it does not have the address of the correct leaf. The address calculation remains similar, but the IAM contains only the downward path from the node.

Range queries can be multicast or sent using point-to-point queries. In that case, the client sends the query to every bucket or node address whose range in the image overlaps the range of the query. Each message contains the range of the bucket encoded in the image. If the actual range and the received one match, then the bucket replies, at least with its range. If the actual range is strictly within the received one (and it cannot be otherwise without merges), then the bucket forwards the query to its parent. The parent forwards the query to the appropriate buckets within the remaining range. If this range exceeds the node range, then it also forwards the query to its parent, etc. All buckets that reply also send their ranges. The client collects the replies and terminates through the range check as for RP*$_N$. A general query can be processed similarly.

5.4 Image adjustment

Consider that the image is encoded in a table *T* as for RP*$_C$. The following algorithm may be used. The pair (*a*, *s*) denotes a pointer *a* and separator *s* following it, in a node or in an entry of *T*. The last pointer in a node is followed by Λ, i.e., by the upper bound of the range for the node.

RP*$_S$ IA algorithm

While visiting the IA-tree bottom-up, and level by level :

1. For every bottom node *n* and every (*a*, *s*) ∈ *n*, modify *T* only in the following cases :
   - (a) - If $s \notin T$, then add (*a*, *s*) to *T* (at appropriate position with respect to the order on *T*).
   - (b) - If *T* contains (*a*, *s'*) ; $s' \neq s$, then set *s'* to *s*.
   - (c) - If *T* contains (*a'*, *s*) where $a' \neq a$, then set *a'* to *a*.
2. For every (*a*, *s*) in node *n* at non-bottom level of IA-tree, modify *T* only in the following cases:
   - (a) - If $s \notin T$, then add (*a*, *s*) to *T*.

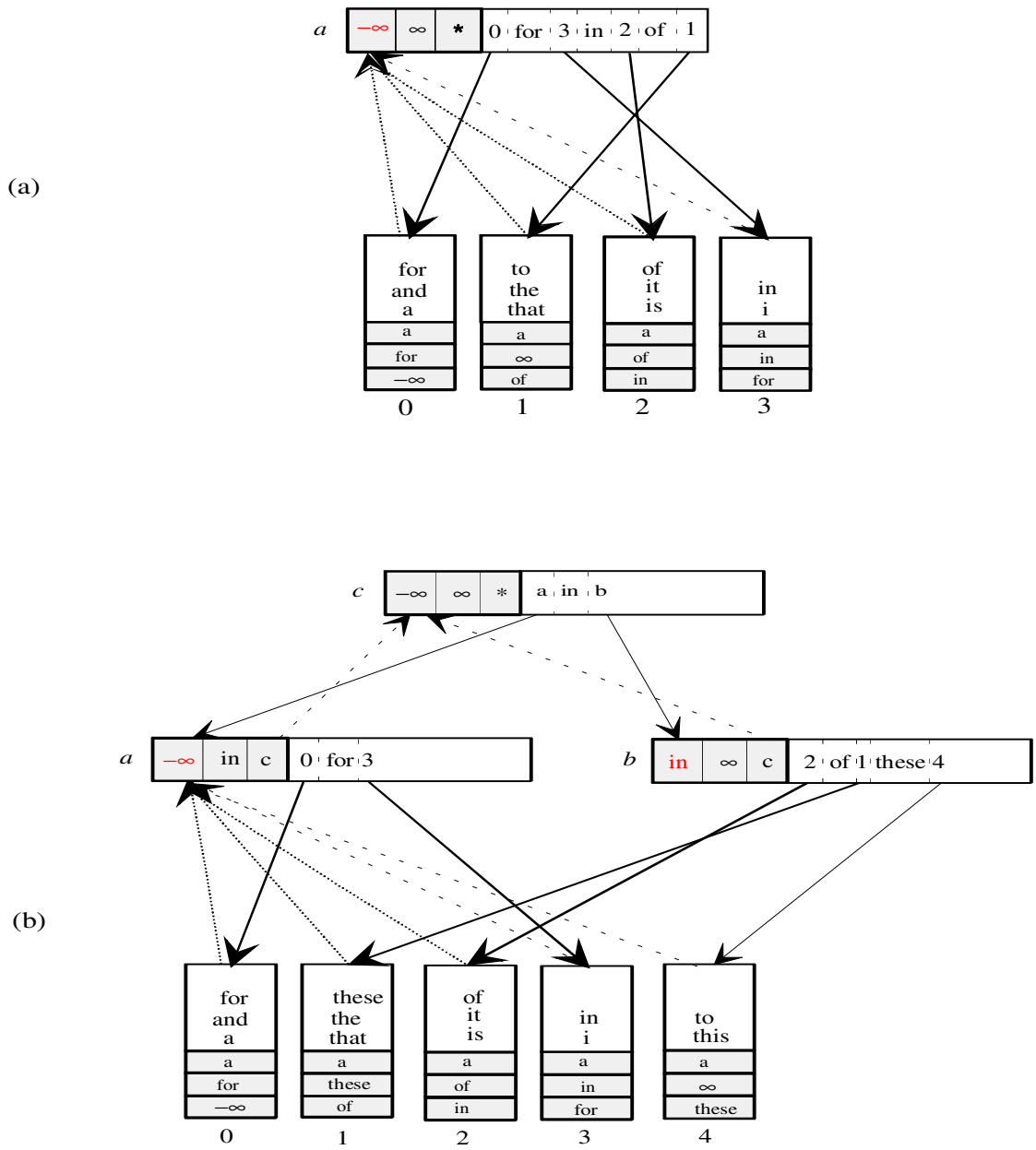

Fig. 5 An RP*$_S$ file with (a) 2-level kernel, and (b) 3-level kernel

.

(b) - If T contains (a', s) where $a' \neq a$, and if $s = \Lambda(n) \neq \infty$, then:
let s' be the predecessor of s in n, and let s'' be the predecessor of s in T,

- if $s'' < s'$, then set a' to a.

Steps 1.a bring to T new bucket addresses and separators. Step 1.b updates values of Λ for buckets with addresses already in T. Step 1.c updates the address a' for s already in T, hence $s = \Lambda(a')$, that became Λ of another bucket a in the meantime, because of a split of bucket a', and, perhaps, of right siblings resulting from this split. The address a' can also be node address brought to T by Step 2.a. Step

1.c progressively refines such addresses in T to bucket addresses. To allow for such refinements is the rationale for visiting IA-tree bottom-up.

Node addresses in T can occur for 2-level kernels and higher. An IA-tree can then bring separators with pointers to nodes, some of which are outside the IA-tree. This would be the case for instance if the kernel in Fig. 5.b expanded further to nodes d and e. Then, the first IA-tree sent to a new client would bring to T three nodes, e.g., (a, c, d), and node c would have separators s_b and s_e that are Λ's and last keys in nodes b and e, and pointers to these nodes. A further IAM bringing, e.g., node b would lead to refinement of (b, s_b) in T to some (N, s_b), found in node b, where N is a bucket address.

In a kernel with more than two levels, an IAM can bring an address a' pointing to a node other than the bottom node of the kernel. A further IAM that brings some (a, s) and finds (a', s) in T, should only refine T, which means here that it should set a' to a only if a is the address of a node under node a. Otherwise, the adjustment would make search path through the kernel longer, and would deteriorate access performance. Step 2.b guarantees that every adjustment of T performed in such a case is a refinement. Indeed, only $s = \Lambda (n) < \infty$ could be replicated in a node at level higher than this of n. In this case, s'', also necessarily brought previously to T from that node, must precede every s' found under that node. Hence, the replacement of a' should be performed only in this case, as Step 2.b does.

As result, T contains only bucket addresses, or node addresses instead of null pointers '*' of RP*$_C$. The client that does not have bucket address for a key, sends the query to a node, and so never needs to multicast the key, unlike in RP*$_C$. The node is possibly the closest to the bucket in the kernel tree. Further IAMs progressively bring the missing bucket addresses. A new client's image always eventually converges towards the actual range partitioning of the file, provided that every bottom level node of the kernel is probed, and that no splits occurred during the probing.

5.5 Kernel adjustment

When a node n splits, parent pointers to n that should point to the new sibling are not refreshed (obviously, it would not be efficient to do it). The kernel structure itself needs then an adjustment that we call *kernel adjustment*. This is the situation of bucket 4 in Fig. 5.b. A bucket n' detects the need for adjustment when an IAM message comes from its parent. At every such message, every node compares its parent pointer n to the address where the message comes from. Especially, it may detect the need for adjustment when it splits, and sends c_m to its former parent. There, c_m is found out of range. The former parent sends c_m to its parent, etc. untill c_m is found within the range. Then c_m is sent down to the actual parent node. This node finally sends an IAM to bucket n' with the actual parent pointer.

5.6 Performance

The load factor of RP*$_S$ file is that of RP*$_C$, except for additional storage for the kernel. As keys are typically much shorter than records, this storage should be comparatively small hence the load factor should remain about 0.7 in practice (under random inserts). The kernel size of a 2-level file is O [m]. Larger files will likely be 3-level files, hence the kernel size should be O [M / 0.7 + m]. Assuming as in Section 4.2 that l_a = 4, and l_c = 40, a 100 KB kernel should suffice for a 1000 bucket file.

Messaging cost increases in the worst case to:

1+ 2 (h - 1) + 1 messages in the worst case per key insert or search in an h-level RP*$_S$ tree.

5 + 2 (h -1) + 5h per split in the worst case, when the parent's pointer is outdated, and a split creates a new root.

The latter formula assumes the split algorithm using a split coordinator and point-to-point messages only, leading to 5 messages per split, instead of 4, if the multicast were used as for RP*$_N$. Note that the worst case split is very unlikely.

The convergence of a new client is much faster than for RP*$_C$. The image converges when the client received the ranges from all the bottom layer nodes. This is one IAM for a 2-level tree, as the kernel has only one node. It is at most O [$\lceil M / (0.7\ m) \rceil$] IAMs in general, assuming random inserts. In practice the

convergence can be faster, as an IAM often brings two bottom nodes with addresses and separators new for *T*.

The average search and insert performance for RP*$_S$, obtained through simulations are presented in the next section.

6. COMPARATIVE ANALYSIS OF RP* FAMILY

6.1 Messaging costs

Fig. 6 shows average messaging costs of inserts and searches for RP* files created with bucket size *b* ranging from 50 to 2000. The fanout *m* for RP*$_S$ is *m* = 100. The curves result from simulations like those performed for LH* [LNS93]. LH* messaging costs are plotted for comparison.

Insert costs are total messaging costs to build the file, including the split messages and IAMs, divided by the number of inserts. In these experiments, each file was built through insertions of 100,000 random keys. The insert costs show excellent performance of all the algorithms, with the values under 1.25 messages per insert. As one could expect, performance is better for buckets with larger capacity, especially for $b > 50$ when the cost drops below 1.1 messages per insert. The results also confirm that for larger *b*'s the differences in insert costs between RP* algorithms are negligible. Also, the RP* family appears competitive to that of LH*. This demonstrates that it is possible to build an SDDS that efficiently supports range queries while still maintaining excellent performance for key-based queries.

Within the family, RP*$_S$ is always the worst performer, although differences become infinitesimal for larger *b*. This is due to extra messages to build the kernel, and, interestingly, despite much fewer IAMs than for RP*$_C$. RP*$_N$ is the best performer, performing even better than LH*. However, this results from the highest use of multicast, thus potentially impacting throughput.

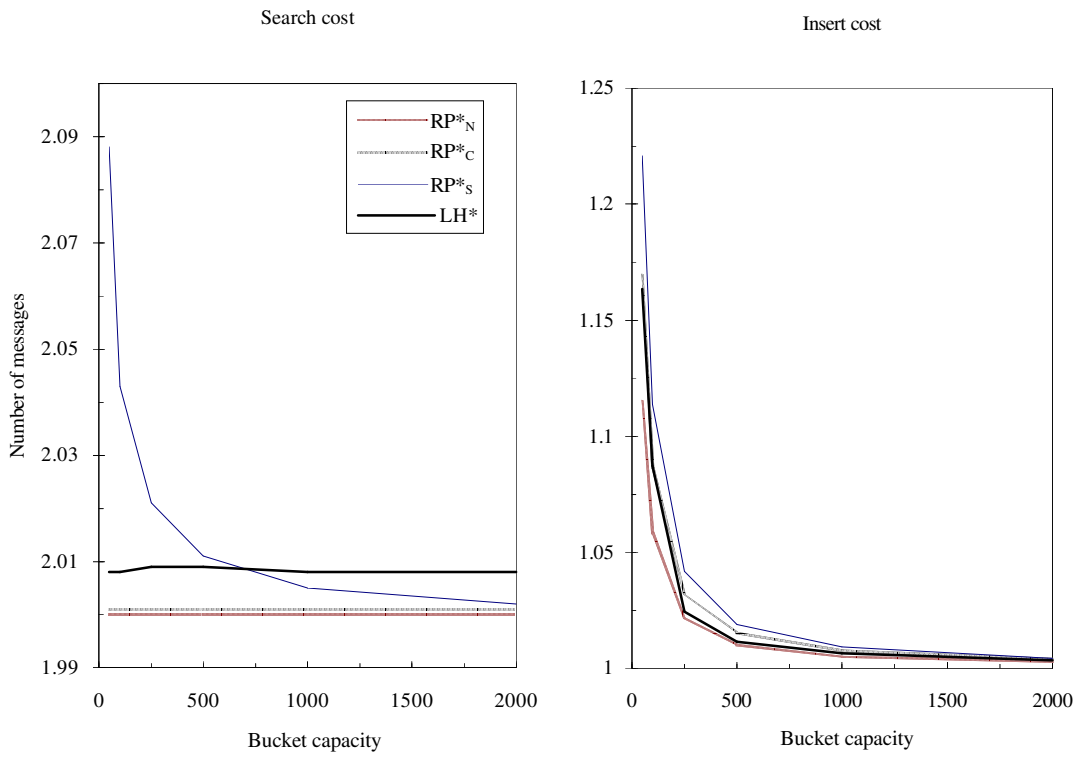

Fig. 6 Search and insert costs of the RP* family and of LH*

The search costs are the average costs of new clients. The files are populated with 100,000 records and the clients retrieve 1000 random keys in each experiment, starting each time with en empty image of the file. The curves confirm the expectations of excellent search performance, i.e., 2 messages per search in practice, for all the RP* algorithms. The RP*$_N$ optimal performance is hardly a surprise. RP*$_C$ cost of 2.001 messages per retrieval results from the fact that all IAMs are piggybacked on reply messages, hence only the first search directed to bucket 0, costs three messages, instead of two. This explains why RP*$_S$ search cost is always higher, despite faster convergence of the image.

The search cost of RP*$_N$ and of RP*$_C$ is always slightly lower than that of LH*, but LH* messages are only-point-to-point. RP*$_S$ cost is higher for smaller b 's. For larger b 's, RP*$_S$ slightly outperforms LH*. This is due to decreasing file size, as the number of inserts is constant in the experiment. For a small file, image convergence is faster for RP*$_S$ than for LH*, leading to better performance, as it will be confirmed below. The situation reverses as the file size scales-up.

6.2 Image convergence

Table 3 shows the number of IAMs that are required for a new client of a static file to have its image of the file converge to the actual file state. The client performed a series of random searches in 100,000 key files, built through random inserts, until its image converged. RP*$_N$ does not appear, as it does not maintain an image of the file. The results clearly confirm the expectations on the convergence of RP*$_C$ and of RP*$_S$. The former is M - 1, the latter is hundred times faster, as the fanout in our experiment was set to m = 100. LH* convergence is under O ($\log_2 M$) [LNS93].

b	RP*$_C$	RP*$_S$	LH*
50	2867	22.9	8.9
100	1438	11.4	8.2
250	543	5.9	6.8
500	258	3.1	6.4
1000	127	1.5	5.7
2000	63	1.0	5.2

Table 3. Number of IAMs until image convergence.

6.3 Performance of less active clients

In these experiments, there are two clients. Client C_1 inserts N keys for every key inserted by the client C_2 ; N = 1, 10, 100, 1000. Split cost is ignored since splits are assumed to take place concurrently with inserts, and we are interested only in the client access cost. With respect to a single client case, performance of both clients may be affected [LNS93]. Tables 4 summarizes the results obtained for the RP* family, and includes for comparison those of LH*. The insert costs are computed for a small and a large b, and various N values. Client C_1 always inserts 100,000 keys.

The results confirm the expectations. RP*$_N$ performs best, and none of the clients is not affected. Otherwise, greater is N , i.e., less client C_2 is active, the more it is affected. Equally active clients are both affected with respect to a single client performance (which is in practice that of client C_1 for N = 1000). RP*$_S$ generally outperforms RP*$_C$ for both clients, reversing what appeared in Fig. 6 for a unique client. This results from faster image convergence, hence from better accuracy of the RP*$_S$ images. RP*$_C$ performs slightly better only for the least active client C_2 ($N \geq 1000$) for b = 50, where the benefit of m = 100 fanout fades out.

The performance loss of RP*$_C$ with respect to RP*$_S$ is always negligible for the faster client C_1. For client C_2, RP*$_S$ can be notably faster, e.g., for N = 100 and b = 50, and for N = 1000 and b = 500. Insert costs of RP*$_S$, and of RP*$_C$ are always higher than of LH*, although in general not much higher. For the RP*$_S$ file with for b = 500, the difference is under 2 % regardless of N value. However, for client C_2 with $N \geq 100$ and b = 50, LH* is notably faster than both algorithms.

***b* = 50**

N : 1	1	1	10	1	100	1	1000	1
	C_1	C_2	C_1	C_2	C_1	C_2	C_1	C_2
RP*$_S$	1.052	1.051	1.036	1.126	1.034	1.405	1.034	2.170
RP*$_C$	1.108	1.108	1.061	1.447	1.055	2.018	1.055	2.070
RP*$_N$	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
LH*	1.011	1.011	1.010	1.034	1.010	1.130	1.010	1.430

***b* = 500**

N : 1	1	1	10	1	100	1	1000	1
	C_1	C_2	C_1	C_2	C_1	C_2	C_1	C_2
RP*$_S$	1.009	1.009	1.006	1.028	1.005	1.123	1.005	1.420
RP*$_C$	1.010	1.010	1.005	1.052	1.005	1.414	1.005	2.040
RP*$_N$	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
LH*	1.003	1.004	1.003	1.013	1.003	1.060	1.003	1.330

Table 4. Performance of two clients.

6.4 Overall characteristics

Table 5 sums up the overall characteristics of RP* algorithms and of LH*. All other factors are the same for RP* algorithms, or about the same, especially the load factor and access costs. All the formulas are in order of magnitude, the *O* - notation being avoided for simplicity. In throughput formulas, *n* denotes the throughput of the network, and *s* that of a server.

The overall conclusion with respect to the RP* family is that if the network is of m-net type, RP*$_N$ is probably the best choice. It reaches almost optimal performance, and is the simplest one. If multicast should be limited, or throughput is a moderate concern, and the network is an h-net or a g-net, then RP*$_C$ is the simplest choice. If network is a g-net, and high performance is the major goal, or multicasting is not available, RP*$_S$ is best choice. It can also potentially scale up to larger sizes than RP*$_N$ or RP*$_C$. If range partitioning is useless, LH* provides better access performance, is simpler, and can scale up even further, as it requires neither index nor multicasting.

	Client index	Servers index	Multicast	Image convergence	Throughput scalability	Size scalability
RP*$_S$	M	M	Optional	$\lceil M / (0.7\ m) \rceil$	Min (n , Ms)	> 1000
RP*$_C$	M	0	Limited	M - 1	Min (n , Ms -Δ)	1000
RP*$_N$	0	0	Every query	No image	Min (n , s)	≤ 1000
LH*	0	0	Optional	$\log_2 M$	Min (n , Ms)	>> 1000

Table 5 Basic characteristics of RP* family and of LH*

7. CONCLUSION

Exploring the perspectives that multicomputers open up is one of most challenging areas of computer science. Hash-based SDDSs were shown very efficient data structures for multicomputers with performance far beyond those of traditional hash files. Ordered SDDS add the capability of efficient processing of range queries and of ascending or descending traversals. We have shown that the RP* family form an efficient class of ordered SDDSs. Depending on the network type, and requirement on throughput, the user can choose the algorithm that is best suited for storage requirements, algorithmic complexity, and access performance. All three algorithms we have studied are faster than traditional ordered structures of similar sizes by an order of magnitude or two.

RP*$_N$ proves by its existence that it is possible to design practical ordered data structures without any index. This results from the use of multicast, already shown useful for many prominent algorithms, e.g., for the Byzantine agreement, and now proves important also for data structures. RP*$_C$ imposes indexes on clients for higher throughput on faster networks. RP*$_S$ allows for the highest performance at the price of the indexes on servers and clients. It also allows for an ordered SDDS on a network without multicast.

Future research should concern deeper analysis of performance of RP* family, under various conditions of usage. The algorithms should be implemented, measured, and benchmarked. Numerous interesting design decisions omitted above will then also be addressed, e.g., the internal organization of buckets, of bulks, and of image, the overall organization of file management etc. Bulk operations lead to interesting generalizations of in-bucket, in-bulk, and in-image binary search, to find more efficiently several keys [K93]. The overall goal of implementation of a file system able to scale up to a thousand sites is clearly not simple, but is among most important goals in the computer science at present.

On the other hand, variants keeping only a part of the image, to save storage on the client at the price of some deterioration of access performance, should be of interest to many applications. One should study also ordered SDDSs applying ideas in numerous traditional variants of B-trees, and in other ordered structures [S89]. Finally, one should analyze the RP* family as support for database operations and transaction processing.

8. ACKNOWLEDGMENTS

We thank Jim Davis (HP Labs) and Domenico Ferrari (UC Berkeley) for valuable information on broadcast and multicast protocols. We also thank John Byers and Micah Adler (UC Berkeley) for helpful comments and discussions.

9. REFERENCES

[BZS93] Bershad, B., Zekauskas, M., Sawdon, W. The Midway Distributed Shared Memory System. IEEE-COMPSAC 1993, 528-537.

[C93] Culler, D. & al. LogP: Towards a Realistic Model of Parallel Computation. ACM-SIGPLAN Symp. on Principles and Practice of Parallel Programming, 1993.

[C94] Culler, D. NOW: Towards Everyday Supercomputing on a Network of Workstations. EECS Tech. Rep. UC Berkeley, to app.

[ChS92] Chamberlin, D., Schmuck, F. Dynamic Data Distribution (D^3) in a Shared-Nothing Multiprocessor Data Store. VLDB-92, 1992.

[D93] Devine, R. Design and Implementation of DDH: Distributed Dynamic Hashing. Int. Conf. on Foundations of Data Organizations, FODO-93. Lecture Notes in Comp. Sc., Springer-Verlag (publ.), Oct. 1993.

[G88] Gray, J. The Cost of Messages. 7th ACM Symp. on Principles of Distributed Systems. 1988.

[G88a] Garcia-Molina, H. Kogan, B. Node Autononmy in Distributed Systems. IEEE-PDIS. 1988, 85-93.

[K93] Karp, R. M. A Generalization of Binary Search. Algorithms and Data Structures. Lecture Notes in CS. Springer-Verlag, 1993. Dehne & al (ed.).

[ILP93] Iftode, L., Li, K., Petersen, K. Memory Servers for Multicomputers. IEEE-COMPSAC 1993, 538-547.

[JK93] Johnson, T. and P. Krishna. Lazy Updates for Distributed Search Structure. ACM-SIGMOD Int. Conf. On Management of Data, 1993.

[LNS93] Litwin, W. Neimat, M-A., Schneider, D. LH* : Linear Hashing for Distributed Files. ACM-SIGMOD Int. Conf. On Management of Data, 1993.

[LNS93a] Litwin, W., Neimat, M-A., Schneider, D. LH*: A Scalable Distributed Data Structure. (Nov. 1993). Submitted for journal publ.

[LNS93b] Litwin, W., Neimat, M-A., Schneider, D. RP* : a Scalable Distributed Data Structure using Multicast (extended abstract) HPL-DTD-93-009, (Sept. 1993).

[LNS94] Litwin, W., Neimat, M-A., Schneider, D. RP* : A Family of Order Preserving Scalable Distributed Data Structures. HPL-DTD-94-012, (Feb. 1994).

[LRLH91] Litwin, W., Roussopoulos, N., Levy, G., Hong, W. Trie Hashing With Controlled Load. IEEE-TSE, 17, 7 1991, 678-691.

[MS90] Matsliach, G., Shmueli, O. Distributing a B+-tree in a loosely coupled environment. Inf. Proc. Letters, 34, 1990, 313-321.

[MS91] Matsliach, G., Shmueli, O. An Efficient Method for Distributing Search Structures. IEEE-PDIS, 1991.

[PLH89] Perrizo, W., Lin, J., Hoffman, W. Algorithms for Distributed Query Processing in Broadcast Local Area Networks. IEEE TKDE, 1, 2, 1989, 215-225.

[S89] Samet, H. The Design and Analysis of Spatial Data Structures. Addison-Wesley, 1989.

[VBWY94] Vingralek, R., Breitbart, Y., Weikum, G. and R. Yavatkar. Distributed File Organization with Scalable Cost/Performance. ACM-SIGMOD Int. Conf. On Management of Data, 1994.