ParaDict, a Data Parallel Library for Dictionaries*

Joaquim Gabarró    Jordi Petit i Silvestre
Universitat Politecnica de Catalunya
Departament de Llenguatges i Sistemes Informatics
C/ Pau Gargallo, 5
08028 Barcelona, Spain
{gabarro, jpetit}@lsi.upc.es

Abstract

ParaDict, a data parallel library for dictionaries having two different interfaces is presented. The first interface is written in C* for data parallel users and the second interface in C, for users that want to use a parallel library but not to write parallel programs. We have seen that C* is an adequate tool to code theoretical PRAM algorithms into readable programs. These programs were ran on a CM 200 with better times than other existing implementations. They also have much better asymptotic behaviour when compared to a sequential implementation on a workstation. Finally, the relationship between data parallelism and vectorization is explored, transforming C* code into C code plus compiler directives and running the result on a Convex C3480 machine. Even if (almost all) the loops were vectorized, the performances were modest. All these facts allow us to look at the development of other parallel libraries with moderate optimism.

1. Data parallel libraries

Sequential abstract data types are well known from theory and practice and complete libraries exist for them. For instance, the well known LEDA library developed by K. Mehlhorn and S. Näher [12] written in C++ is widely used (even on introductory courses). It belongs to the so called SP/SL model:

SP/SL model: A Sequential Program calls a Sequential Library. In this case, the program and the library are both written in a sequential language.

The situation is quite different on massive parallelism. We have a very sophisticated theory on parallel data structures (cf. J. JáJá [11]) but very few practical work. Therefore, it was really tempting to explore the “practical” issues of this theory writing (a small part of) a library in a data parallel language. We choose C* [18] as a high level data parallel language since using it we can code data parallel algorithms with reasonable time and effort getting clear and readable programs. These programs were ran in a SIMD Connection Machine 200 [10].

What kind of users could be interested in such a library? Data parallel programmers could find it interesting, but this community is rather small and we would like to recover sequential programmers. Therefore we introduce the following two models (and interfaces):

PP/PL model: A Parallel Program calls a Parallel Library. This approach is addressed to data parallel programmers. This means in practice a C* program calling functions written in C*. Thanks to data parallelism, the use of this interface is almost identical to that used in sequential environments.

SP/PL model: A Sequential Program calls a Parallel Library. Addressed to sequential programmers. Since C* contains C as a subset, programmers can write C code and call operations of a parallel library. This library starts transforming sequential data into parallel data and runs data parallel procedures in the parallel system. Depending on the problems this approach can be interesting.

In this paper, we consider the design, implementation and evaluation of a parallel library for handling dictionaries (ParaDict). This is a theoretically well known domain: taking only the research based on dynamic data structures on PRAMs we have among others, the work of W. Paul, U. Vishkin and H. Wagener based on 2-3 trees [14] (on which our implementation is based), the work of L. Higham and E. Schenks on B-trees [9] the work of J. Gabarró, C. Martinez and X. Messegue based on Skip Lists [7] or the work

---

*This work has been partially supported by ESPRIT LTR Project no. 20244 — ALCOM-IT.
of J. Gabarró and X. Messeguer based on AVL trees [8]. However, less implementations have been realized. We know the works done by M. Gastaldo et al. [5] and by X. Messeguer [13]. We shall compare their results with ParaDict.

The remaining of this paper is organized as follows. By the means of a toy example, we first develop the use of the previous parallel interface models. Then we show how, inspired by the LEDA sequential library, we have designed ParaDict’s interfaces. After that, we comment our implementation using 2-3 trees and we present some experimental results aiming to evaluate it. Finally we show how we have ported the C* programs to a vectorial machine. We close the paper with some concluding remarks.

2. PP/PL and SP/PL models: a toy example

Let us develop a little bit more these three models with a toy example based on sorting integers. Figure 1 sketches the three approaches. In the SP/SL model the sequential program can use the procedure

```c
void Sort (int a[], int n) {
    Sequential sorting code
}
```

In this case, the sequential program calls `Sort(a,n)` where `a` is a sequential array. For the PP/PL model, the main data parallel program calls `Sort(&a)` where `a` is a parallel variable having a shape decided at run-time (using the `current` keyword) in order to have a neutral data distribution [6]:

```c
void Sort (int:current *a) {
    [Rank(*a)]*a=*a;
}
```

In the SP/PL approach, the sequential programmer calls `Sort(a,n)` where `a` is a sequential array variable:

```c
void Sort (int a[], int n) {
    shape [n];
    with (s) everywhere {
        int:current pa;
        pa=write_to_pvar(a); /* FE->CM */
        [Rank(pa)]pa=pa;
        read_from_pvar(a,pa); /* CM->FE */
    }
}
```

The procedure uses the C* primitives `write_to_pvar` and `read_from_pvar` connecting sequential and parallel variables. Remark that the core of the data parallel sorting algorithm (`[Rank(pa)]pa=pa`) can be the same in the PP/PL and in the SP/PL models. Moreover the SP/SL and SP/PL headers (`Sort(int a[],int n)`) also coincide.

To construct the SP/PL library from the PP/PL case, we just need to add a bridge level (see figure 1) made of two steps that wrap the parallel program: the first (`write_to_pvar`) spreads the sequential array to the parallel subsystem, the second (`read_from_pvar`) gathers the result to the front end.

Of course, it can be argued that the introduction of the bridge level can produce a decrease of the performances of programs, mainly due to the overload of connecting parallel and sequential variables. We have found, however, that this approach is useful for some kinds of problems. For instance, figure 2 compares the running time of the previous programs. Thus, it can be seen which is exactly the time consumed by the bridge and that (even with it) the SP/PL model is faster than the SP/SL (which uses `qsort` from `stdlib.h`). This will happen, in general, when dealing with big problems, with large amounts of data or long execution times.

3. The design of ParaDict’s library interface

The library we present, ParaDict, implements the operations to efficiently handle dictionaries on parallel computers using a data parallel approach. Concretely, we offer two...
different implementations (a prototype kind, and a 2-3 tree
based kind) with two different interfaces: a parallel one and
a sequential one. The former interface is written in C* and
it is aimed for parallel programs (PP/PL model); the latter is
written in C and it is designed to be used within sequential
programs (SP/PL model). Both interfaces contain the same
operations which are based on the sequential implementa-
tion of dictionaries in LEDA [12]. We think that this is a
good starting point, due to the relevance of the LEDA li-
brary in sequential computing. Refer to [17] for a complete
description of ParaDict’s interfaces.

**Parallel interface.** In order to use parallelism, LEDA
headers have to be enhanced. The way to do it in C* is easy:
where LEDA expects a single key, value or item, ParaDict
expects a parallel variable of them. In C*, a parallel vari-
able can be seen as a usual array where each component
has a (virtual) processor associated to it. All our functions
maintain the meaning of the context, so the inactive posi-
tions will not be treated. For instance, let us consider the
operation that, given a key in a dictionary returns its associ-
ated value. The name of this method in LEDA is
\( \text{Access} \) and its header in C++ syntax is
\[
T\text{Val} \text{d.Access}(\text{TKey key})
\]
where \( \text{TKey} \) is the type of the keys, \( \text{TVal} \) is the type of the
values and \( \text{d} \) is a dictionary object. The header for the same
operation in the PP/PL interface of ParaDict is
\[
\text{TVal:current Access}(\text{TDict } \ast \text{d}, \text{TKey:current keys})
\]
where \( \text{TDict} \) is now the dictionary ADT. The meaning for
the parallel version is the extension of the sequential one:
for each active key, its associated value in the dictionary is
returned. The following headers list the main functions of
ParaDict’s parallel interface:

\[
\begin{align*}
\text{TItem:current Insert}(\text{TDict } \ast \text{d}, \text{TKey:current keys}, \\
\text{TInf:current infs}) \\
\text{TItem:current Lookup}(\text{TDict } \ast \text{d}, \text{TKey:current keys}) \\
\text{void DelItems}(\text{TDict } \ast \text{d}, \text{TKey:current keys}) \\
\text{void Change}(\text{TDict } \ast \text{d}, \\
\text{TItem:current items}, \text{TInf:current infs}) \\
\text{bool:current IsNil}(\text{TDict } \ast \text{d}, \text{TItem:current items}) \\
\text{TKey:current Keys}(\text{TDict } \ast \text{d}, \text{TItem:current items}) \\
\text{TInf:current Infs}(\text{TDict } \ast \text{d}, \text{TItem:current items})
\end{align*}
\]

**Sequential interface.** The same operations than before
are available, but in order to follow Ansi C, some of the
parameters have been changed and some have been added.
In fact, since all these functions work with open arrays, the
user has to supply an integer \( k \) representing their size. For
convenience, a new parameter called \( \text{mask} \) simulating the
setting of the context in C* has also been added. This can
be seen in the SP/PL version of the Access procedure:
\[
\text{void Access}_{-}(\text{TDict } \ast \text{d}, \text{int } k, \text{bool mask[]}, \\
\text{TKey keys[]}, \text{TVal vals[]})
\]
The following headers list the main functions of ParaDict’s
sequential interface:

\[
\begin{align*}
\text{void Insert}_{-}(\text{TDict } \ast \text{d}, \text{int } k, \text{bool mask[]}, \\
\text{TKey keys[]}, \text{TInf infs[]}, \text{TItem items[]}); \\
\text{void Lookup}_{-}(\text{TDict } \ast \text{d}, \text{int } k, \text{bool mask[]}, \\
\text{TKey keys[]}, \text{TItem items[]}); \\
\text{void IsNil}_{-}(\text{TDict } \ast \text{d}, \text{int } k, \text{bool mask[]}, \\
\text{TItem items[]}, \text{bool arenil[]}); \\
\text{void DelItems}_{-}(\text{TDict } \ast \text{d}, \text{int } k, \text{bool mask[]}, \\
\text{TItem items[]}); \\
\text{void Change}_{-}(\text{TDict } \ast \text{d}, \text{int } k, \text{bool mask[]}, \\
\text{TItem items[]}, \text{TInf infs[]}); \\
\text{void Keys}_{-}(\text{TDict } \ast \text{d}, \text{int } k, \text{bool mask[]}, \\
\text{TItem items[]}, \text{TKey keys[]}); \\
\text{void Infs}_{-}(\text{TDict } \ast \text{d}, \text{int } k, \text{bool mask[]}, \\
\text{TItem items[]}, \text{TInf infs[]});
\end{align*}
\]

**Lacks and drawbacks.** The most important of them
is the lack of genericity or polymorphism. Once the
types \( \text{TKey} \) and \( \text{TVal} \) have been defined, dictionaries
parametrized with them have to be used. Another impor-
tant restriction is that, for the current implementations, the
\( \text{TKey} \) type has to be arithmetic. It is hard to correct these
drawbacks in a coherent form without changing the lan-
guage in which the code of library is written, C*. A more
object oriented, yet data parallel language could be of inter-
est.

### 4. C* implementation

To implement ParaDict we choose the algorithms given
trees (a class of search trees where all leaves have the same
depth and internal nodes have two or three sons). These al-
gorithms on a EREW (exclusive read, exclusive write) have
time
\[
T(n, k) = \mathcal{O}(\log n + \log k)
\]
where $k$ is the number of keys to search, insert or delete and $n$ is the number of leaves in the tree.

We found them interesting because a 2-3 tree is an irregular structure and because the algorithms involve many interesting programming points. To give a flavour of them, we briefly describe some procedures of the main algorithms (search, insertion and deletion). The complete documented implementation can be found in [15].

Search. The basic data structure we need (besides the tree) is the “packet”. Each key $[i]$ keys to search belongs to a packet $[j]pckts$. A packet has the following structure:

```c
typedef struct {
    char state; // Active, Passive or Located
    TItem node;
    nat firstP, lastP;
    TKey firstK, lastK;
    TCell cell;
} TPacket;
```

The fields firstP and lastP are pointers to the first and last key that a packet contains. firstK and lastK are these keys. The field node is a pointer to the node where the packet is located and cell is a copy of it (it contains the keys and pointers to the sons and parent). These redundancies avoid concurrent reads.

The function Locate creates a set with a unique packet that contains all the keys to search (InitPackets) and, from the root to the leaves, splits and routes down this set of packets (RoutePackets). When all the packets become inactive, the located leaves are notified to each member of every packet with a segmented copy scan operation.

```c
void RoutePackets (TDict *d,
    TPacket:current *pckts,
    TKey:current keys)
{
    char:current dir = Direction(*pckts);
    where (dir==DirL) {
        pckts->node=pckts->cell.Lp;
        pckts->cell=[pckts->cell.Lp]*d->cells;
    } else where (dir==DirR) {
        pckts->node=pckts->cell.Rp;
        pckts->cell=[pckts->cell.Rp]*d->cells;
    } else where (dir==Stop) {
        pckts->state=Located;
    } else /* where (dir==Split) */ {
        SplitPackets(d,pckts,keys);
    }
}
```

Procedure SplitPackets handles the packets that have to be splitted. Using a dichotomic scheme, new packets are created, and their fields are correctly updated.

```c
void SplitPackets (TDict *d,
    TPacket:current *pckts,
    TKey:current keys)
{
    TItem:current mid;
    mid=1+(pckts->firstP+pckts->lastP)/2;
    [mid]*pckts=*pckts;
    [mid]pckts->firstP=mid;
    [mid]pckts->firstK=[mid]keys;
    pckts->lastP=mid-1;
    pckts->lastK=[mid-1]keys;
}
```

Insertion. Parallel insertions use a divide and conquer scheme. They are done using a bottom-up tree reconstruction using pipelines, with several hanging requests at different levels of the tree, forming waves. Basically, a hanging request contains a source subtree to hang to a destination node. This is the data structure which leads the algorithm:

```c
typedef struct {
    bool active;
    TItem source, destination;
    TPos pos;
} THang;
```

While there exist active requests, the main insertion routine alternates two calls to LeavesUp with one call to HangsUp. LeavesUp is used to launch hanging requests at the base of the tree; it uses an implicit divide and conquer. When possible, HangsUp hangs the hanging requests, otherwise it splits the nodes and lifts up these requests to the
upper level of the tree. In order to not have interferences between the levels they modify, waves of requests must be separated at least by two levels.

```c
InitHangs(leaves,&hangs,&nodes);
LeavesUp(d,leaves,&nodes,...,&hangs);
while (|= hangs.act) {
  HangsUp(d,&hangs);
  HangsUp(d,&hangs);
  LeavesUp(d,leaves,&nodes,...,&hangs);
}
```

**Deletion.** The deletion algorithm works in a similar way: the TEraser auxiliary data structure is used to direct the divide and conquer and pipeline processes. Again, waves must be separated by two levels:

```c
typedef struct {
  char state;
  TItem source;
} TEraser;

DelItems2 (TDict *d, TItem:current items) {
  TEraser:current erasers;
  InitErasers(&erasers,items);
  while (|= (erasers.state!=Passive)) {
    RemoveLeaves(d,&erasers);
    ArrangeLevels(d,&erasers);
    ArrangeLevels(d,&erasers);
  }
}
```

We found that programming all these techniques was not obvious and quite challenging for a C* programmer. Against our first impression, we found C* well adapted.

### 5. Experimental results

In order to evaluate the performance of some usual operations of our library, we have measured and analyzed their running time on a CM 200. Experiments have been repeated enough times; results shown below are their mean. The variances were not substancial.

**Evaluation of the Lookup and Insert operations.** The experimental results obtained for searching or inserting $k$ keys in a dictionary storing $n$ elements are shown in figure 5. For comparison with a well-known workstation, we also show the times needed for the equivalent sequential insertions. We conclude that, with our machines, even if the sequential implementation is faster than the parallel one for reasonable values of $k$, the time increase is smoother, making clear the scalability of our parallel library.

Comparisons with other implementations. Figure 5 compares results from [13] with ours. Since the experimental conditions where the same, we can affirm that ParaDict’s implementation with 2-3 trees is slightly more efficient than X. Messegue'r's implementation based on skip lists. Moreover, it saves space and can store much more elements.

The comparison of our results against the ones given in [5] by M. Gastaldo, shows that our implementation is 5 times faster. For instance, an insertion of 500000 elements can be done in 43 seconds (7 if the dictionary is empty) on a Connection Machine with 16K processors with ParaDict, whereas on a MasPar-1 with 1K processors it takes 240 seconds. However, we have to cautious about this kind of information, because we are not only comparing the algorithms but the parallel machines involved in the measurements.
6. C#*, vectorization using data parallelism

There is a lot of resemblance between the data parallel and vectorial programming models: both consist of applying the same operation to different data. For this reason, we found interesting to port our parallel programs written in C* to a vectorial computer (a Convex C3480, a machine with registers of 128 elements and 8 banks of memory). We achieved it by defining a set of transformations to convert C* code to C code augmented by compiler directives (which have the #pragma form) [4]. In the following, this language is called C#/*. In this paper, we describe only some of these transformations. The complete set we used can be found in [16].

Send operation. The first transformation we show is for the send operation, which in C* is written as:

```c
int:current dst, src, indx;
[indx]dst = src;
```

where `indx` is supposed to be a permutation. Its correct transformation to a C#/* efficient program is the following:

```c
int src[], dst[], indx[];
for (i=0; i<n; i++) indx2[i] = indx[i];
for (i=0; i<n; i++) src2[i] = src[i];
#pragma _CNX force_vector
for (i=0; i<n; i++) dst[indx2[i]] = src2[i];
```

Auxiliary copies have to be generated, because these vectors could be the same. The compiler directive has to be given, since the compiler cannot recognize `indx2` as a permutation. This operation (or its dual, the get operation) can be efficiently executed if the processor has scatter (or gather) instructions, which is the case on the Convex.

Parallel prefix. This second operation is related to another important operation in data parallel programming. Given an operator `⊕` and a vector `a = [a₀, ..., a₀₋1]`, we define `/₀ a = [b₀, ..., b₀₋1]` where `bᵢ = ⊕ⱼ₌₀⁻¹ aⱼ`. In C* this function is called scan and belongs to the standard communication library. A naive implementation would be:

```c
for (b[0]=a[0], i=1; i<n; i++)
  b[i] = b[i-1] + a[i];
```

but the vectorizing compiler could not solve the recurrence and would leave this loop scalar. However, implementing the classic PRAM algorithm [11] as:

```c
for (i=0; i<n; i++) b[i] = a[i];
for (j=1, p=pow2(j-1); j=log2(n); j++)
  for (i=p/2; i<n; i++) aux[i] = b[i];
for (i=p; i<n; i++) b[i] = aux[i-p] + aux[i];
```

the compiler can stripmine the `i` loops. We have found that on the Convex machine, the second implementation is 2.5 times faster than the first one when `n > 4096`. This result is negative, since according to Amdahl's law, this slow speedup will have effect on all the algorithms that contain it.

Conditioning. Let us now consider the conditioning instruction where `(cond) S`. We code the active context with an unidimensional array. As the contexts have a block structure, we need a stack. This approach has been already considered by L. Bougé and J. Levaire [3] to give an operational semantics to a basic data parallel language, L. The corresponding transformation is:

```c
for (i=0; i<n; i++) if (cond[i]) S(i);
```

As the Convex C3480 and CM 200 machines have a similar masking behaviour the transformation is efficient. Both have also the same drawback: the execution time does not decrease when only few unmasked elements of a vector are processed. The instruction where `(cond) S1 else S2` can be rewritten as `t = cond; where (t) S1; where (!t) S2`, and the preceding technique applies.

Example: Radix sort. Sorting is also a usual operation on data parallel machines. Radix sort is a good candidate because it is easily implemented on the CM 200 machine and it will be interesting to see what kind of C vectorizable code we will obtain. First of all, recall the RadixSort procedure written in C* as:

```c
void RadixSort (nat:current *a, nat n)
{
  nat:current enu;
  nat d,k,D=boolsizeof(nat:current);
  for (d=0,k=1; d<D; d++,k<<=1) {
    enu=(n-1) - enumerate(0,CMC_downward,
      CMC_exclusive,CMC_none,CMC_no_field);
    else
    enu=enumerate(0,CMC_upward,
      CMC_exclusive,CMC_none,CMC_no_field);
    [enu]*a=*a;
  }
}
```

Its transformation is the following:

```c
void RadixSort (nat a[], nat n) {
  nat:current enu;
  nat d,k,e,D=sizeof(nat:current);
  for (d=0,k=1; d<D; d++,k<<=1) {
    where (*a & k)
    enu=(n-1) - enumerate(0,CMC_downward,
      CMC_exclusive,CMC_none,CMC_no_field);
    else
    enu=enumerate(0,CMC_upward,
      CMC_exclusive,CMC_none,CMC_no_field);
    [enu]*a=*a;
  }
}
```
Note that the transformation into C# give us a code where only the more external loop remains sequential. The other loops corresponding to enumerations, copy and send operations have been vectorized by the Convex C compiler.

**Segmented scans.** Segmented scans have been extensively studied by G. Bliekel in [1, 2]. Assume we have an array \( v \) and another array \( f \) of flags. Each flag specifies the start of a new segment. For instance, if we consider the following array \( v = [1 1 1 7 1 4 1 5 6 8 4] \) and \( f = [1 0 0 1 0 1 0 0 0 1 0] \) the segmented array is represented as \( s = [1 1 1 7 1 4 1 5 6 8 4] \) and the segmented prefix sum will be \( s = [1 2 3 7 8 4 5 10 16 8 12] \). The segmented version of an \( \oplus \) operation can be implemented as \( (v_r, f_r) \oplus (v_b, f_b) = (v_r \oplus v_b, f_r \lor f_b) \) where \( v_r = v_0 \) if \( f_0 \) holds, \( v_r = v_a \oplus v_b \) if \( \neg f_b \) holds and \( f_r = f_0 \lor f_b \). In C# we get the following code:

```csharp
for (j=1,k=log2(n); j<=k; j++) {
    p=pow2(j-1);
    for (i=p/2; i<n; i++) {
        v2[i]=v[i]; f2[i]=f[i];
    }
    for (i=p; i<n; i++) if (!f2[i]) {
        v[i]=v2[i-p]+v2[i];
        f[i]=f2[i-p];
    }
}
```

**A first judgement.** We presented a way to transform directly data parallel programs written in C* into C# programs. Therefore it is possible to transform data parallelism into highly vectorizable code and we have a connection between these two approaches to parallelism. However, as we have seen in the parallel prefix transformation, the speedup is not so important. Therefore, transformations seem to be good but the speedup seems to be bad. To get a better inside into the behaviour of C# we developed a vectorial version of ParaDict. Next section explores the results.

**7. C# vectorial dictionaries**

Applying these kinds of transformations to a subset of ParaDict, we have obtained its vectorial implementation (with the SP/PL interface). Even if almost every loop was made vectorial by the optimizing compiler, the performances achieved at run-time were very poor: table 7 reports it. The speedup of the vectorial implementation with respect to the sequential one is 5 for the building operation and 2 for the search operation. Vectorizing the insertion operation is self-defeating.

**Table 1. Some measures characterizing the behaviour of the different implementations. Searches and insertions are made on a tree with 150000 leaves.**

<table>
<thead>
<tr>
<th>Operation</th>
<th>CM 16K Par</th>
<th>Sca Par</th>
<th>Convex Vec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build with ( 2^{16} ) leaves</td>
<td>7.118</td>
<td>26.723</td>
<td>7.658</td>
</tr>
<tr>
<td>Search ( 2^{14} ) keys</td>
<td>1.776</td>
<td>6.624</td>
<td>3.340</td>
</tr>
<tr>
<td>Insert ( 2^{14} ) keys</td>
<td>—</td>
<td>2.407</td>
<td>40.309</td>
</tr>
</tbody>
</table>

We conjecture that the reason for these modest improvements (when they exist) is again the highly irregular structure we are dealing with, and the bottleneck it creates accessing the limited set of memory banks.

**8. Conclusions**

In parallelism, there seems to be a gap between theory and practice. In many cases it is quite difficult to measure the effort necessary to transform an informal algorithm into readable code. We got a pleasant surprise with C*, because sophisticated algorithms were easily coded. As in the sequential case, the program development by stepwise refinement has been extensively used to get readable programs.

Moreover it has been possible to define two complete and useful interfaces: a sequential and a parallel one. This is important because they reflect two different views of parallelism. The sequential interface is planned to be used by sequential programmers using data parallelism in a hidden way. The data parallel interface is to be used by programmers having a knowledge of data parallelism. Both classes of programmers coexist today. People coming from computer science uses friendly data parallel environments, but most of the people coming from other disciplines prefers a sequential environment. However both classes of programmers can take advantage of data parallel libraries. More important, only one kind of data parallel program has to be developed. These parallel programs can be used from a sequential environment just adding a bridge level, always easy to build. We claim that the whole approach will remain true for other applications.

Experimental results left open many questions. Whereas our dictionary implementation (a highly irregular and dynamic object) performs better than previous implementations, parallel results are not so good in relation to their sequential counterpart. However, in many cases these comparisons are not so clear and, in our case, involve rather old
SIMD machines against new and powerful sequential computers. It is unclear what happens with more modern parallel machines. Moreover, in the near future, compilers for data parallel languages on MIMD machines could become more popular. If this happens, data parallel programming could become every day programming, but care has to be taken when dealing with irregular data structures as ours.

Many questions remain open about the distinction between high and low level approaches to programming in parallel. Programming with a sequential languages plus a message passing library (e.g. PVM) is for us a low level approach. We also consider a low level activity to write (directly) good sequential vectorizable programs. From the other side we consider C* programs as high level. Transformations from C* to C# connects high and low levels, but the results were poor. We guess the same will happen if we try a transformation from data parallel programs to sequential programs + message passing. In both cases the problem seems to be the fine grain and the irregularity of the application, but this is just a feeling without any theoretical proof. In any case, we think that data parallel languages will continue to be an interesting and elegant high level counterpart to other low level approaches.

Further information

Further information regarding ParaDict can be found on the World Wide Web at the address

http://www-lsi.upc.es/ParaDict

Acknowledgements

The implementation and evaluation of our programs has been done at the Centre Europeu de Paral·lelisme de Barcelona (CEPBA, now C4) and at the Parallelldatorcentrum (PDC) of the Kungl Tekniska Högskolan (Sweden). We acknowledge X. Messeguer for help and letting us use its code.

References