PBML

The Prague Bulletin of Mathematical Linguistics
NUMBER 98 OCTOBER 2012 87–98

Parallel Phrase Scoring for Extra-large Corpora

Mohammed Mediani, Jan Niehues, Alex Waibel

Karlsruhe Institute für Technology

Abstract

This paper presents a C++ implementation of the phrase scoring step in phrase-based systems that helps to exploit the available computing resources more efficiently and trains very large systems in reasonable time without sacrificing the system's performance in terms of Bleu score.

Three parallelizing tools are made freely available. The first exploits shared memory parallelism and multiple disks for parallel IOs while the two others run in a distributed environment.

We demonstrate the efficiency and consistency of our tools, in the framework of the Fr-En systems we developed for the WMT and IWSLT evaluation campaigns, in which we were able to generate the phrase table in one third up to one seventh of the time taken by *Moses* in the same tasks.

1. Introduction

Phrase scoring is one of the most important and yet very expensive steps in phrase-based translation system training. Typically, it consists of estimating the corresponding scores for each unique phrase pair extracted from an aligned parallel corpus. Usually, the scores are estimated based on two directions (from source to target and vice versa). Therefore, the process is accomplished in two runs. In the first run, counts are collected and then the scores are estimated based on the source phrases while in the second run a similar task is performed based on the target phrases.

This process is memory greedy. However, for non large corpora it could be performed efficiently in the physical memory by some implementations. For instance, *memscore* (Hardmeier, 2010) uses a lookup hash table based on STL[1] maps to index

[1]C++ Standard Template Library `http://www.sgi.com/tech/stl/`

Corresponding author: `mohammed.mediani@kit.edu`
Cite as: Mohammed Mediani, Jan Niehues, Alex Waibel. Parallel Phrase Scoring for Extra-large Corpora. The Prague Bulletin of Mathematical Linguistics No. 98, 2012, pp. 87–98. doi: 10.2478/v10108-012-0011-z.

the phrases. Then the hash identifiers are used to directly access the corresponding phrases in order to update the marginal and joint counts. Unfortunately, this does not scale very well for corpora of large sizes. As a matter of fact, a memory requirement of more than 60GiB was reported for a corpus of 4.7M sentence pairs (Hardmeier, 2010).

On the other hand, most systems such as the widely used phrase-based system *Moses* (Koehn et al., 2007), handle the memory limitation by streaming the large data sets, keeping only a limited amount of data into memory, and saving temporary results into disk. In fact, all the pairs which correspond to a given phrase should be kept into memory while gathering the marginal and joint counts for this phrase. Consequently, the streamed data must be sorted depending on whether the computation is being held based on source phrases or target phrases . In *Moses*, this is achieved by performing two sorting operations using the standard Unix *sort* command.[2] Even though, being a good external memory sorting tool, the Unix *sort* command is not optimal when the corpus is very large. For instance, the runs are formed and sorted serially, it lacks support for multiple disks, and the IO could not be overlapped with the computations.

Gao and Vogel (2010) developed a platform for distributed training of phrase-based systems starting from word alignment until phrase scoring. Even though excellent speed gains were reported, this system runs on top of the Hadoop framework, and therefore needs the cluster to fit this special infrastructure.

Unlike applications which operate exclusively on data stored in main memory, applications which involve external memories such as hard disks face an additional challenge with the high data transfer latency between the external and main memory. For this purpose, data structures and algorithms have been developed in order to minimize the IO overhead and to exploit the available resources such as parallel disks and multiple processors more efficiently (Vitter, 2008). Luckily, different external memory APIs have been created in order to make the underlying disk access and low level operations transparent to programmers. Such platforms include, but are not limited to, LEDA-SM (Crauser and Mehlhorn, 1999), TPIE (Arge et al., 2002), Berkeley DB (Olson et al., 1999), and STXXL (Dementiev and Kettner, 2005).

The main goals of our tools for phrase scoring are to exploit CPU and disk parallelism in an external memory environment, so that the phrase sorting and score computation are performed more efficiently. The CPU parallelism is ensured by the OpenMP library (Chapman et al., 2007) (eventually coupled with an MPI implementation (Pacheco, 1996)), while the disk parallelism and other external memory functionalties are ensured by the STXXL library. STXXL is preferred over other environments due to its superior performance, ease of use (STL-compatible interface), and explicit support for parallel disks (Dementiev et al., 2008).

Most of our tools are written in C++. The underlying CPU parallelism comes in three flavours: multithreaded, hybrid, and distributed. The multithreaded version

[2] `http://unixhelp.ed.ac.uk/CGI/man-cgi?sort`

uses shared memory parallelism and therefore runs on a single node. In the hybrid setting, multiple nodes can be used. On each of these nodes the shared memory parallelism is exploited. The distributed tool proceeds in a MapReduce strategy (Dean and Ghemawat, 2008): Starting from Giza alignments, the large corpus is split into partitions and training is performed independently on the partitions. For each part, standard *Moses* tools are used for alignment combination, lexical scoring, and phrase extraction. For phrase scoring, we use a slightly modified version of the multithreaded tool. It allows all the partial counts to be saved as well. The partial phrase tables are then merged and the probabilities are reestimated using the new updated counts.

In the next section, the external memory sorting is briefly presented in the framework of the STXXL implementation. Afterwards, the architecture and underlying algorithms of our different software flavours are dissected and its usage is explained. Then some experimental results are presented and discussed. Finally, a conclusion about the main findings and eventual extensions ends the paper.

2. External memory sorting in STXXL

Due to its extreme importance, the external memory sorting has received continuous improvements over the years. The different techniques can be categorized in two classes: distribution sorts and merging sorts. A detailed survey of both approaches can be found in Vitter (2008).

Details about STXXL sort implementation are given in Sanders and Dementiev (2003). In the following, we briefly review its important aspects.

STXXL implements a multiway-merge sort. It assumes that the data records are of fixed size. The processing then could be held on fixed size data blocks. The STXXL library forms the backbone of many sorting benchmark[3] winners in the past years (Andreas et al., 2011; Rahn et al., 2009; Beckmann et al., 2012). The two key steps of STXXL sorting are as follows:

Run formation In a double buffering strategy, two threads cooperate to read/sort the different runs. The first thread sorts the run which occupies half of the sorting memory, while the second thread is either reading the next run or writing the sorted run. The sorter thread creates lighter data structure consisting of only the keys and pointers to the actual elements. After that, it sorts the keys in the new data structure where the sorting method depends on their number (straight line code if it doesn't exceed 4, insertion sort if it is between 5 and 16, otherwise it uses quicksort).

Multiway merging In order to define the order in which blocks will be streamed into the merger, the smallest elements in each block are recorded in a sorted list during run

[3] http://sortbenchmark.org/

formation. The position of an element in this list defines when its containing block will enter the *merging buffers*. The merger keeps a number of blocks equal to the number of the sorted runs in merging buffers. In order to minimize the time of selecting the current smallest element, the keys of the smallest elements of all blocks in merging buffers are kept in a tree sctructure.

STXXL uses an *overlap buffer* for reading and a *write buffer* for writing in order to overlap IOs and merging. The size of the overlap buffer depends both on the number of runs and the number of parallel disks while the size of the write buffer depends on the number of disks only. If the write buffer has a number of blocks which exceeds the number of disks, a parallel output is submitted. Similarly, if the overlap buffer has a number of free blocks which exceeds the number of disks, a parallel read is performed.

Distributed External Memory sorting (DEMSort) is an extension of the STXXL sorting so that it fits the distributed case where the sorting is rather performed on multiple machines (Rahn et al., 2010). The key difference here is the introduction of an additional intermediate phase between run formation and multiway merging: the so-called *Multiway selection*.

Like the distribution sorts, the multiway selection tends to find global splitting points over all the sorted runs. By the end of this operation, each node knows its exclusive range of data. Afterwards, the data are redistributed globally over the nodes using an all-to-all operation to satisfy the range constraints. In this case, the MPI interface is used for the inter-node communication. Finally, the merging is done locally as explained before.

3. Software architecture and algorithms

Like *Moses* scoring tool, our phrase scoring tools take three files as input and produce a phrase table as output. The first input file contains the extracted phrases (called `'extract.0-0.gz'` in Moses convention) and the other files are two bilingual dictionaries which model $\Pr(s \mid t)$ and $\Pr(t \mid s)$ for every source and target words s and t if they are aligned at least once (`'lex.0-0.f2e'` and `'lex.0-0.e2f'` in Moses convention).

Typically, the phrase table records 4 scores for every extracted phrase pair. Relative frequency and lexical score for each direction (source to target and vice versa). Our lexical score is identical to the one produced by Moses Scoring tool, whereas our relative frequency is smoothed using modified Kneser-Ney smoothing as described in Foster et al. (2006).

The development of our tools led to three different levels of parallelism: multithreaded, hybrid, and distributed. The multithreaded version forms the core of the other two versions. The multithreaded and hybrid versions parallelize only the phrase scoring whereas the distributed version parallelizes the former steps too. In the following, we explain each of these versions.

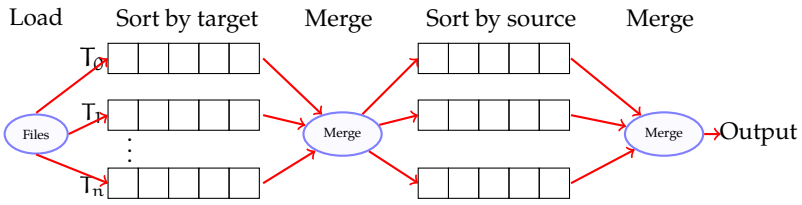

Figure 1. Multithreaded phrase scoring anatomy

3.1. Multithreaded phrase scoring

The basic data structure used in this software is STXXL vector whose interface is similar to STL vector but it rather stores data which does not all reside in memory. STXXL vector elements are stored in the form of key, value. The keys of this vector are the phrase pairs (source and target phrases concatenated) and the values are the different counts. In order to satisfy the fixed size record of STXXL vectors, the keys are represented by a fixed-length string.

As depicted in Figure 1, the process consists of several threads, each of which takes care of one large STXXL vector of data. The phrase table is the result of five consecutive steps. Details about each of these steps are presented in what follows.

Loading the data First of all, the lexical dictionaries are loaded into two STL maps (one for each direction). Afterwards, each thread reads one phrase pair at a time, computes its lexical score, and then loads it into its corresponding STXXL vector. This multithreaded way allows for computations and IOs to be overlapped.

There are two ways to read pairs from the file into memory. The fast way: where all the threads read the same file concurrently one line at a time. In this case, the input file should not be zipped. The alternative way allows to read directly from the zipped file, the master reads from the file and pushes the lines into a FIFO queue. The other threads pop lines from the queue and process them.

As soon as the loading is complete, the lexical maps are disposed since they will not be needed anymore.

Sorting by target phrases Every thread sorts its vector by simply calling the STXXL sort function which performs a multiway merging sort on the corresponding vector.

Merging and computing the target-based scores The merging follows the same approach as the multiway merging. The first elements from all vectors are organized in a tree structure. Whenever an element is taken out, it is replaced with the next element from the same vector.

Parallel threads acquire a lock on the tree and get all the pairs with the same target phrase in a local vector, then release the lock for the next thread. After collecting the pairs, every thread updates the corresponding count fields and writes the updated records to a new STXXL vector. Since the identical pairs have to be uniquified in this step, our implementation allows chosing one lexical score and one alignment based on maximal lexical score or the most occurring one.

Sorting by source phrases Again, this is done in parallel by the STXXL sort.

Merging and computing the source-based scores This operation is identical to the merge based on target phrases.

Writing out the phrase table Like the loading phase, two writing ways are possible. The way which supports writing zipped phrase table is performed by a single thread while the multithreaded way writes only unzipped files.

Optionally, all the counts can be recorded for further use (as in the distributed version). It is as well possible to write out an optional abridged phrase table containing only phrases which match a list of given n-grams.

3.2. Hybrid parallel phrase scoring

The extension DEMSort allows us to effieciently sort an STXXL vector spread over multiple interconnected machines. There are only few changes in the architecture compared to the previous version. We suppose that the nodes dispose of a shared disk space. First of all, all the nodes build the lexical maps in the same way. Afterwards, every node reads a quota of the input file of phrase pairs into an STXXL vector. Running the DEMSort could raise the following issue: the phrase pairs which correspond to a given phrase could be spread between two adjacent nodes due to the redistribution as explained in Section 2. To fix this, every node sends all the phrase pairs corresponding to the first phrase to its immediate predecessor. As a consequence of this sorting approach, no further data exchange between the nodes is needed.

Every node performs the local merging and scoring strictly identical to the multithreaded version. In our development process, this resulted in an unbalanced load between the nodes. Consequently, we extended the merging with a dynamic load balancing strategy. The final merging procedure executed on every node looks as follows:

1. Execute a multithreaded merging and listen to signals from other nodes
2. If request for sharing is received from another node, then send half of the remaining pairs to that node
3. When finished, signal all other nodes
4. If all nodes have no remaining pairs, then exit

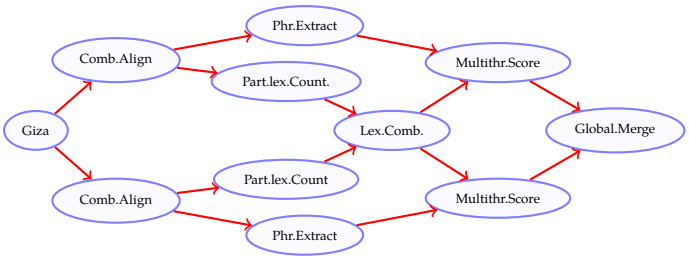

Figure 2. Distributed phrase table construction process (2 partitions)

5. Receive half of the remaining pairs from the node with the largest remaining number of pairs
6. Go to 1

The output is done in a similar manner to the previous system where all the nodes write to the same file concurrently. The position from which a node starts writing in the common output file is estimated based on the number of enteries in this node's vector.

3.3. Distributed translation model construction

This version is based on two complementary pieces: the aforementioned multithreaded scorer and a multithreaded merger. The objective of the latter is to merge streamed partial phrase tables produced by the scorer.

In fact, starting from partitioned Giza alignement files all the subsequent steps are run independently (typically on a cluster of machines). However, a slight modification is introduced in this pipeline in order to produce correct lexical dictionaries.The counts for aligned words are collected from each part independently and then globally combined in an additional step from all the collected counts. The sequencing and dependence between the different steps of this version is shown in Figure 2 (for a number of partitions equals to 2).

The global merger is very similar in design to the multithreaded scorer. The only difference is that the counts are not initialized to 0, but rather based on the saved counts in the partial phrase tables. Afterwards, it proceeds in the same steps as the multithreaded scorer.

3.4. Usage

All our tools show different options by specifying `-h` or `--help` flag.

Multithreaded phrase scorer The number of threads can be specified by setting the `OMP_NUM_THREADS` environment variable. If this variable is not set, it will be set to the number of physical cores available on the machine. The most important options and flags for this software are (all options can be printed using the `-h` flag):

`-e, -l, -L` are used to provide the extract, lexical dictionary source to target, lexical dictionary target to source files respectively.

`-b` with this option, the sorting internal memory per thread can be specified in Mega bytes.

`-w` is used to specify the different disks (paths) which will serve as parallel disks for STXXL sort. It is a comma-separated list.

Hybrid phrase scorer The binary in this case is called `pscore`. It accepts the same arguments as the previous one. Though, it needs to be started with `mpirun`.

Running the distributed version on a cluster The script which automates the partitioning the Giza alignment files and ensures the correct dependency between the jobs (as shown in Figure 2) is written in Python and uses specific commands for the *Slurm*[4] queue manager. We believe however that it can be easily adapted to other schedulers.

4. Experimental results

In this section, we show some performance comparisons between the different versions of our scoring tools. We compare them as well to Moses. The hardware environment where these experiments took place is a cluster consisting of 8 core machines with 32GiB of memory and 16 core machines with 64GiB memory.

All the machines have access to a RAID NFS shared space and dispose of a local disk of 1.7TiB. In all experiments the parallel scorers use two disks for the STXXL vectors (the local disk and NFS). The first set of experiments (in WMT2011) was held on the 8 core machines, while the others were held on the 16 core machines.

Experiments in the WMT2011 In this set of experiments, the Multithreaded version was run on a 16-core machine, whereas the hybrid was run on four different machines (using 4 cores out of 8 on each one). Table 1 compares the speed of different tools used in this experiment, whereas Table 2 shows the Bleu scores resulting from a system based on Moses phrase table and the hybrid balanced system (we kept only one phrase table since all the tables produced by our tools are identical). These phrase tables are trained based on three parallel corpora (merged into a single large corpus): EPPS, NC, and UN. The total number of parallel sentences is 13.8 millions. Clearly, the best choice here is the hybrid balanced version. It is 7.5 times faster than Moses

[4] `https://computing.llnl.gov/linux/slurm/`

scorer. However, explicitly handling the communication (for both versions) and the load balancing (for the balanced version) from whithin the scoring routines degrades readability, and thus maintaining this code became expensive. This was the reason why we next created the fully distributed version and we didn't report further tests with the hybrid version.

System	Time span
Moses	53h 34m
Multithreaded	28h 49m
Hyb. unbalanced	8h 45m
Hyb. balanced	7h 08m

Table 1. Phrase scoring time span in WMT2011

System	en-fr	fr-en
Moses	23.16	24.16
Parallel scoring	23.24	24.21

Table 2. Bleu scores in WMT2011

System	EPPS+NC	+UN
Moses	11h 23m/27.21	49h 34m/29.13
Multithr.	9h 34m/27.5	27h 44m/29.02

Table 3. Phrase scoring in IWSLT2011

System	Time	Bleu
Moses	92h 46m	29.77
Distributed	49h 20m	30.00

Table 4. Phrase scoring in WMT2012

Experiments in the IWSLT2011 Experiments in this context are shown for Moses vs. the multithreaded version for the same corpora as the previous. For every corpus and system, Table 3 gives the corresponding time span and Bleu score. As in the previous experiment, the amount of speed up becomes more and more appearent as the corpus size augments, while the translation model's performance in terms of Bleu scores is almost invariant. However, the slight difference (Table 3, column +UN compared to Table 1) is mainly due to a different set of disks.

Experiments in the WMT2012 This set of experiments is held between Moses and the distributed version. In addition to the EPPS, NC, and UN corpora, the training data here include the Giga corpus as well (resulting in 29.4 millions parallel sentences). The number of partitions here was 12 and the jobs were submitted independently to the cluster (some of them end up on the same node, which is not optimal). Table 4 records the time required for phrase scoring. It is shown here that the distributed version is almost 2 times faster than Moses.

It is noteworthy that relative frequency in Moses version here was also modified as in Foster et al. (2006). These experiments show that not only our tools are faster

than Moses, but they also produce in most cases slightly better results. We think the reason for that is due to the lexical score selection explained in Section 3.1, unlike Moses where the first occurring score is selected.

Surprisingly, the distributed version was not as fast as the hybrid version. This could be justified by the race condition which occurs during concurrent access to the the NFS space by so many processes.

5. Conclusion

In this paper, we presented three versions of a tool which makes the phrase scoring manageable for extra large corpora. This was achieved by exploiting multiple processing units and parallel disk IOs using the STXXL library for external memories. The first implementation can be run on a single machine. Whereas the other two can be executed in a multinode environment (typically on a cluster of nodes). The three implementations are freely available under the LGPL license and can be downloaded from `http://isl-wiki.ira.uka.de/~mmediani/fscorer`. All these tools depend on the STXXL and OpenMP libraries. In addition to that, the hybrid version assumes the existence of an MPI implementation and the DEMSort extension for the STXXL library.

Given that the bottleneck in this process is the slow disk speeds compared to internal memory, the amount of improvement strongly depends on the number of parallel disks. This could be shown by the experiment in Section 4, where the hybrid version performed better than other versions since it uses multiple nodes each of which uses its local disk as well as the NFS space. The distributed version is still being tested and optimized, therefore the speedup it brings is still low compared to the hybrid version even though they are somehow similar in spirit.

Since the objective of the experiments shown in this paper was to participate in the MT evaluation campaign, they were run on relatively powerful hardware. However, these tools would also work for less powerful architectures, since the memory consumption is bounded by design.

The main limitation of our tools is the disk space consumption. This is essentially due to the fact that our basic data structure uses a fixed size character string for the keys of our STXXL vectors. As a result, some very long pairs cannot be taken into account and shorter ones have to be filled with blank characters. This implies that a considerable amount of the space allocated for keys is not useful. A possible solution to this would be to use suffix arrays to index the phrases and use only the ID's in the STXXL vector keys.

Acknowledgements

This work was realized as part of the Quaero Programme, funded by OSEO, French State agency for innovation. The authors are also glad to Yuqi Zhang for her comments and suggestions on this paper.

Bibliography

Andreas, Beckmann, Meyer Ulrich, Sanders Peter, and Singler Johannes. Energy-efficient sorting using solid state disks. *Sustainable Computing: Informatics and Systems*, 1(2):151–163, 2011.

Arge, Lars, Octavian Procopiuc, and Jeffrey Scott Vitter. Implementing I/O-efficient data structures using TPIE. In *In Proc. European Symposium on Algorithms*, pages 88–100. Springer, 2002.

Beckmann, Andreas, Ulrich Meyer, Peter Sanders Johannes Singler, and Peter Sanders Johannes Singler. Energy-efficient fast sorting 2011, 2012. URL `http://sortbenchmark.org/demsort_2011.pdf`.

Chapman, Barbara, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

Crauser, Andreas and Kurt Mehlhorn. LEDA-SM extending LEDA to secondary memory. In *Proceedings of the 3rd International Workshop on Algorithm Engineering*, WAE '99, pages 228–242, London, UK, UK, 1999. Springer-Verlag. ISBN 3-540-66427-0.

Dean, Jeffrey and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008. ISSN 0001-0782.

Dementiev, R. and L. Kettner. STXXL: Standard template library for XXL data sets. In *In: Proc. of ESA 2005. Volume 3669 of LNCS*, pages 640–651. Springer, 2005.

Dementiev, R., L. Kettner, and P. Sanders. STXXL: standard template library for XXL data sets. *Softw. Pract. Exper.*, 38(6):589–637, May 2008. ISSN 0038-0644.

Foster, George F., Roland Kuhn, and Howard Johnson. Phrasetable smoothing for statistical machine translation. In *EMNLP*, pages 53–61, 2006.

Gao, Qin and Stephan Vogel. Training phrase-based machine translation models on the cloudopen source machine translation toolkit chaski. *Prague Bull. Math. Linguistics*, 93: 37–46, 2010.

Hardmeier, Christian. Fast and extensible phrase scoring for statistical machine translation. *Prague Bull. Math. Linguistics*, 93:87–96, 2010.

Koehn, Philipp, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondřej Bojar, Alexandra Constantin, and Evan Herbst. Moses: open source toolkit for statistical machine translation. In *Proceedings of the 45th Annual Meeting of the ACL on Interactive Poster and Demonstration Sessions*, ACL '07, pages 177–180, Stroudsburg, PA, USA, 2007. Association for Computational Linguistics.

Olson, Michael A., Keith Bostic, and Margo Seltzer. Berkeley DB. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '99, pages 43–43, Berkeley, CA, USA, 1999. USENIX Association.

Pacheco, Peter S. *Parallel programming with MPI*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996. ISBN 1-55860-339-5.

Rahn, Mirko, Peter S, Johannes Singler, and Tim Kieritz. DEMSort-distributed external memory sort, 2009. URL `http://sortbenchmark.org/demsort.pdf`.

Rahn, Mirko, Peter Sanders, and Johannes Singler. Scalable distributed-memory external sorting. In on Data Engineering (ICDE), International Conference, editor, *26th IEEE International Conference on Data Engineering, March 1-6, 2010, Long Beach, California, USA*, pages 685–688. IEEE Computer Society, März 2010.

Sanders, Peter and Roman Dementiev. Asynchronous parallel disk sorting. Research Report MPI-I-2003-1-001, Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany, February 2003.

Vitter, Jeffrey Scott. *Algorithms and Data Structures for External Memory*. Now Publishers Inc., Hanover, MA, USA, 2008. ISBN 1601981066, 9781601981066.

Address for correspondence:
Mohammed Mediani
`mohammed.mediani@kit.edu`
Karlsruhe Institute für Technology
Adenauerring 2,
Karlsruhe 76131, Germany