

Task Pool Teams Implementation of the Master Equation Approach for Random Sierpinski Carpets

K.H. Hoffmann¹, M. Hofmann², G. Runger², and S. Seeger¹

¹ Department of Physics, Chemnitz University of Technology

² Department of Computer Science, Chemnitz University of Technology

Abstract. We consider the use of task pool teams in implementation of the master equation on random Sierpinski carpets. Though the basic idea of dynamic storage of the probability density reported earlier applies straightforward to random carpets, the randomized construction breaks up most of the simplifications possible for regular carpets. In addition, parallel implementations show highly irregular communication patterns. We compare four implementations on three different Beowulf-Cluster architectures, mainly differing in throughput and latency of their interconnection networks. It appears that task pool teams provide a powerful programming paradigm for handling the irregular communication patterns that arise in our application and show a promising approach to efficiently handle the problems that appear with such randomized structures. This will allow for highly improved modelling of anomalous diffusion in porous media, taking the random structure of real materials into account.

1 Introduction

Random fractal structures are used to model the random structural properties found in many real materials such as aerogels, porous rocks or cements. There we find a fractal structure on certain length scales (spanning about two or three decades) [1], while on larger scales the structure looks rather homogeneous. One feature of these materials is that diffusion is anomalous and the behaviour is very well modeled by random walk processes on regular fractals like Sierpinski carpets [2]. But, these regular fractal structures do not exhibit the transition to normal diffusion found in the real materials. This transition could be captured by performing the Sierpinski carpet construction only to some finite stage and repeating the resulting structure, thereby obtaining a crystal like structure with fractal unit cells. However, this does not capture the randomness of the local fractal structure present in real materials, which has quite an influence on the diffusion properties [3]. This randomness in local structure can be modeled by using newly generated carpets instead of repeating one randomly generated unit cell. While in regular (crystal) structures added disorder usually leads to a decrease in diffusion or transport properties, we find here that disorder can also

enhance diffusion on these structures. This is also observed in experiments on ionic conduction in solid electrolytes [4].

In this paper we report how a master equation approach to simulating random walks on random Sierpinski carpet structures may be implemented efficiently. This method is an elegant way to calculate the evolution of the probability densities of random walkers on such structures from a given initial distribution. This initial distribution is assumed to have finite support, usually chosen to be a delta distribution. Though we can apply some concepts developed for regular Sierpinski carpets [2] in a straightforward manner, the randomness of the resulting structures poses some challenges not apparent when considering the simpler regular case. This article describes strategies that can be used for an efficient parallel implementation. An important problem that needs to be solved in order to obtain an efficient parallelization is to develop a strategy for handling the irregular communication patterns that arise due to the random, dynamic growth of the carpet structure covered by the probability distribution. We show that the concept of *task pool teams* [5] provides a suitable framework for handling these issues.

2 Random Sierpinski Carpets

Given a set of $M \times M$ black-and-white patterns (the generators) and a probability distribution for the choice among these patterns, the algorithm to construct random Sierpinski Carpets described by Reis [6] and ben-Avraham [7] is as follows:

1. start from a square (level 0).
2. divide each square into $M \times M$ subsquares.
3. choose a generator pattern at random (according to its probability)
4. remove the subsquares corresponding to white markings in the selected generator
5. for the next level, repeat steps 2 – 4 for each remaining subsquare.

Figure 1 shows an example of the first two refinement steps for a set of three different generators. Note that with just a single generator we obtain regular Sierpinski carpets as a subset of random Sierpinski carpets. The construction procedure can be repeated ad infinitum, where the resulting structure is a random Sierpinski carpet [6]. If we stop at some level l , the resulting pre-carpet pattern of size $M^l \times M^l$ is referred to as an *iterator* of level l .

These pre-carpet structures give a good model for the (in a statistical sense) self-similar micro-structure of porous materials. We therefore use iterators as basic unit in our algorithm to build larger structures by connecting single iterators. For instance, repeating a given iterator in all directions, we obtain a ‘crystal’ with random unit cell. Extending the carpet in all directions by appending newly created random iterators, we obtain a structure with the same properties as real porous materials. The last method is certainly the most difficult to implement, as virtually no savings can be made in the description of

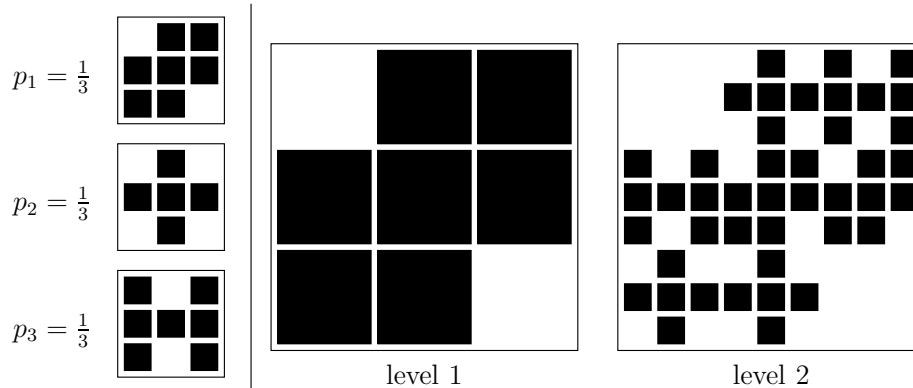


Fig. 1. The first two construction steps for a random Sierpinski carpet constructed from three different generators with equal probability.

the structure. We therefore discuss an algorithm that allows efficient simulation for the last, most demanding case. However, it can easily be modified to handle other cases well.

In order to iterate the master equation on the resulting structure, we introduce the following terms and notations: Consider a random walker is allowed to hop between the midpoints of the *tiles* (black subsquares) in a carpet. In one discrete time step, the walker can move to one of the neighbouring tiles. De Gennes [8] introduced the analogy of a random walker as an “ant in a labyrinth” and with the so called *myopic ant* or *blind ant* algorithms, we obtain the probabilities W_{ij} for a walker to arrive at tile i coming from tile j . Given some probability $p(t, i)$ to find a walker on tile i at time t , we can calculate the probability $p(t + 1, i)$ by accounting for the gain and loss of probability by walkers crossing the boundaries as

$$p(t + 1, i) = (1 - L_i) p(t, i) + \sum_{j \in \langle i \rangle} G_{ij} p(t, j). \quad (1)$$

The sum is over the set of all neighbours $\langle i \rangle$ of tile i , $G_{ij} = W_{ij}$ are the gain factors and $L_i = \sum_{j \in \langle i \rangle} W_{ji}$ is the overall loss of tile i . By iterating the master equation (1) starting with a delta distribution at the starting point we obtain a new distribution for every time t . This distribution determines the mean square displacement accurately, free of the fluctuations pertinent to direct simulation methods. From this, not only the random walk dimension of the fractal can be determined, but also can this probability distribution be compared with theoretical descriptions of anomalous diffusion, e.g. by fractional diffusion equations.

Iteration of the master equation, however, requires a large amount of computer RAM, as for every point in the carpet that can be reached by a random walk in the considered time t , memory to store two probability values need to be

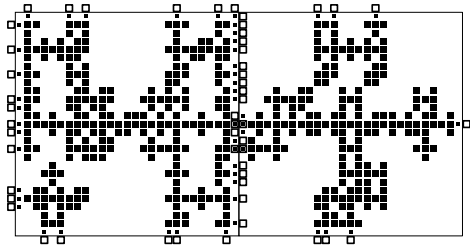


Fig. 2. Two adjacent iterators of level 3 with their body tiles (squares), boundary tiles (small squares) and halo tiles (outlines).

allocated. This memory requirement grows considerably with simulation time t , thus an efficient way of storing and updating these probability values is needed.

3 Data structures and Implementation

In [2] we have already reported on an efficient algorithm for regular carpets. Though the basic idea of dynamic storage of $p(t, i)$ applies straightforward to random carpets, the randomized construction breaks up most of the simplifications possible for regular carpets. For instance, with a regular carpet an iterator pattern of some level determines the whole carpet structure. This is not the case for randomized carpets where each iterator is different. Also, with a dynamically growing data structure the connections to neighbour iterators cannot be predicted in advance of the simulation from analysis of the iterator. Instead, it can only be determined once the carpet has actually been constructed and all neighbour iterators are known.

Our basic unit of processing remains an iterator of level l . We start with one iterator that contains the tile with the non-zero part of the initial delta distribution. The carpet is described by a linked list of iterator descriptions, that store topological information and the probability values at the current and next time step. In every time step this list is traversed once in order to calculate the probability values for the next time step. Within an iterator we have to distinguish the following types of black tiles: *body tiles* are inside the iterator but not adjacent to a boundary, *boundary tiles* are inside the iterator and adjacent to a boundary, *halo tiles* are outside the iterator adjacent to a boundary tile. Figure 2 illustrates this situation showing two iterators with their body (■), boundary (●), and halo (□) tiles.

For body tiles we can perform the update calculation without any additional information other than that stored for the iterator required. For boundary tiles, we do not know the surrounding carpet topology in the beginning. Furthermore we need to know the probability value(s) at the adjacent halo tile(s) in order to perform the update for a given tile. Fortunately, the corresponding terms in (1) vanish initially because we have zero probability that walkers are at those positions. Only as soon as the master equation predicts a non-zero probability value at a boundary tile for the next iteration step we need to make sure the neighbouring iterators are present and the data structures are consistent. Halo tiles are not updated according to (1) but by copying the values after updating the

corresponding boundary tiles from the neighbour iterator. Doing so allows the task of iterating the master equation to be distributed among multiple processes by distributing the iterators.

4 Parallel Implementation

For the parallel implementation we use a master-worker scheme. The master is responsible of overall program control as well as to keep track of the global carpet topology. With the data structures described above, the workers receive a number of iterator descriptions for the iterators they have to process. Thus the global list of iterators to process is split among the workers and each worker has its local list. A load balancing mechanism is implemented by assigning new iterators to the least-busy workers where load is determined by the number of tiles that need to be updated per iteration.

The processing of one time step is organized in three phases:

1. The master informs all workers to start processing their local list of iterators for updating the probability values of the body and boundary tiles. However, it may happen that workers arrive with non-zero probability at boundary tiles, thereby making a carpet extension necessary. If this happens, the worker reports this event to the master and processes the next iterator until it has finished traversing its local list. The master collects messages about carpet extensions necessary.
2. After all workers have finished processing their local iterator list, the master extends the carpet as necessary by assigning newly created iterators to the workers and notifying the workers of the changed carpet topology.
3. Finally, as the last phase in every iteration the boundary values are exchanged. Once this has been finished, results may be collected or a new iteration is started.

For the simulation of about 32000 time steps, the runtimes of the three phases for a straightforward implementation are shown in Figure 3. The carpet increases up to about 2300 iterators, each of size $5^3 \times 5^3$. The implementation uses MPI to send the various control and data messages. The master and every worker process is assigned to a single cluster node. Measurements have been performed on three different Beowulf-type clusters: (A) the Chemnitzer Linux Cluster CLiC with 512 nodes with single Pentium III/800MHz CPUs, 512MB RAM and FastEthernet interconnect and a Xeon cluster with dual Xeon/2GHz CPUs, 1GB RAM and either (B) GigabitEthernet or (C) SCI interconnects. For the Fast- and GigabitEthernet interconnects, the LAM-MPI implementation and for SCI interconnect the optimized SCAMPI implementation has been used.

As can be seen from Figure 3, the amount of wall-time spent in the first phase decreases as the number of nodes is increased. The carpet extension phase has a fairly constant and rather small amount of execution time, because the carpet extension is handled by the master only. The longest time, however, is spent in the third phase performing the boundary update. While with the SCI

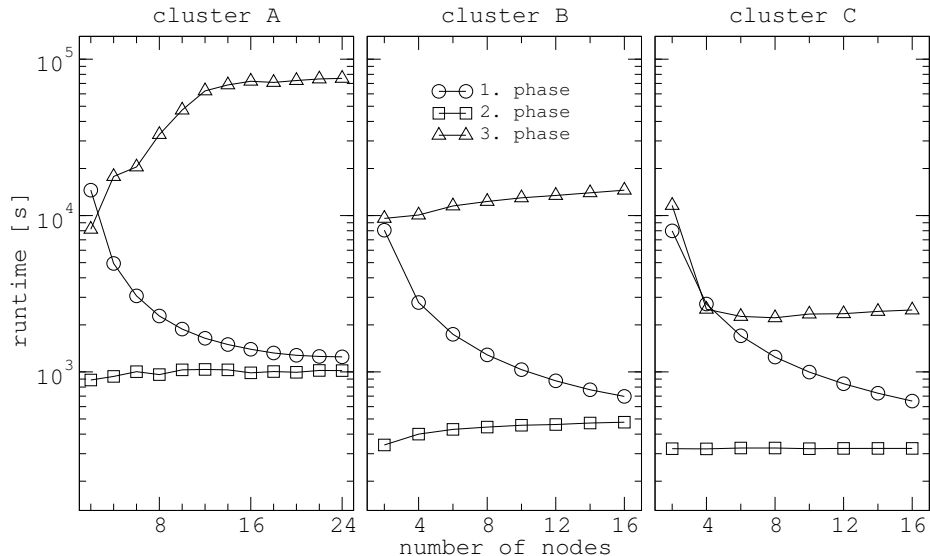


Fig. 3. Runtimes (in wall-time seconds) of the three phases of the main loop: 1) iteration of (1) for each iterator, 2) carpet extension and fixup where new iterators are appended, 3) update of boundary values (possibly between processors).

interconnect (cluster C) a slight speedup can be observed, the amount of time spent in boundary update remains fairly constant for GigabitEthernet (cluster B). For FastEthernet (cluster A) communication time actually increases with the number of nodes. This is because the communication scheme used in the reference implementation results in many short messages, resulting in high latency times adding up. Another drawback is the highly irregular communication scheme arising from sending and receiving the boundary updates. Because for a single-threaded worker the resulting irregular communication protocol cannot be proven deadlock-free, the third phase is serialized: each worker either sends messages to other workers or waits for incoming messages. The best improvement can therefore be achieved with a better implementation of the boundary update phase.

4.1 Optimized Boundary Updates

For parallelizing the boundary update phase by handling the irregular communication we use *task pool teams*. The *task pool* concept uses a decomposition of the computational work into tasks. A task pool stores the tasks and threads are responsible for the execution of tasks. *Task pool teams* are an approach for extending the idea of task pools to the use of parallel platforms with distributed memory. They combine task pools on single cluster nodes with explicit communication. We use the implementation of task pool teams for SMP cluster presented in [5], which uses Pthreads for SMPs and MPI for communication between SMP

nodes. A specific communication thread and a number of worker threads run on each SMP node. Thus, each worker of the master-worker scheme is now actually realized as a collection of internal worker threads and one communication thread. An advantage of task pool teams is to support irregular communication requirements.

In order to speedup the boundary updates we focused on three additional implementations:

- As a first implication from the strong impact of the latency, we start to collect single boundary update messages for each worker until a sufficiently large message can be sent. This avoids many small messages in favour of larger messages thereby reducing the high impact of the latency to start communication. We will refer to this as the boundary collect mechanism.
- To achieve a parallel update with task pool teams we use the communication thread to handle update requests from other workers. At the same time a worker thread is able to process the local iterator list performing the boundary updates. The messages are sent using the specific asynchronous communication which is mapped to MPI operations by the task pool teams implementation. This provides individual point-to-point communication between pairs of workers whenever messages need to be transferred. We will refer to this as the asynchronous parallel update.
- Another method for parallel update with task pool teams uses the specific communication for notifying the workers to perform a boundary update. After this notification all workers participate in sending their messages synchronous by all-to-all communication operation. We refer to this method as the synchronous parallel update.

Both parallel update methods use the boundary collect mechanism for sending larger messages instead of many small ones.

As can be seen from Figure 4, the boundary collect mechanism provides a saving in runtime of about an order of magnitude. This is caused by avoiding many small messages between nodes handling adjacent iterators. Especially for the high latency Fast- and GigabitEthernet (on clusters A and B) this provides the most substantial savings. The additional use of the parallel update scheme leads to different results with the different architectures. For the uniprocessor cluster (A) using only a small number of nodes the runtimes remain fairly unchanged. However, with an increasing number of nodes there is a slight saving in runtime. These rather fair improvements can be attributed to the use of multithreaded programming on uniprocessor architectures. Using the asynchronous and synchronous method makes no difference. Much better results are obtained with the SMP cluster (B and C). Using the parallel update we observe a gain of another order of magnitude in execution time. This is achieved by using the task pool teams concept for handling the irregular communication. Additional benefits are achieved by overlapping of communication and computation through the parallel execution of communication and worker thread. The results for the asynchronous parallel update are shown only for the SCI interconnect (cluster

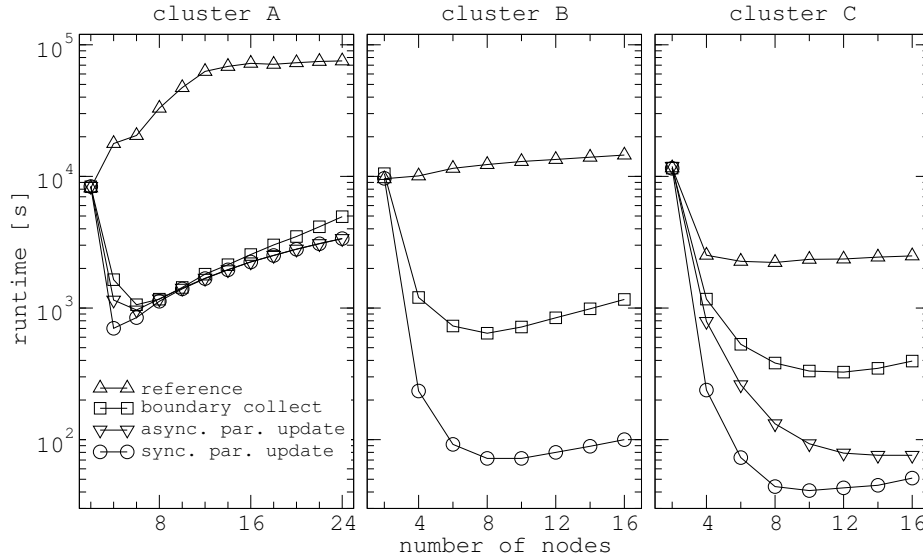


Fig. 4. Runtimes of various implementations of the boundary update phase: reference implementation, boundary collect mechanism and task pool teams with asynchronous and synchronous parallel boundary updates.

C). In comparison with the synchronous method the savings in execution time are rather small. The runtime results for the GigabitEthernet interconnect with the LAM-MPI implementation in the multi-threaded environment are diverse and not shown in the diagram.

4.2 Optimized Iterator Updates and Overall Runtimes

Due to the good results using the task pool teams with the SMP cluster, we extend their usage to another computationally expensive part of the simulation. The processing of the local iterator list in the first phase can easily be split into independent tasks. These tasks are executed in parallel by different worker threads.

As can be seen from Figure 5, the multi-threaded implementation of the task pool teams leads to another saving in runtime. For the multiprocessor architecture this is the expected behaviour. However, for the uniprocessor cluster there appears also a slight decrease in execution time as the number of nodes increases. On the multiprocessor cluster no additional benefits are achieved using more worker threads than CPUs per node available.

Finally, in Figure 6 we compare the overall runtimes of the optimizations using task pool teams and the reference implementation. On three different clusters the reference implementation shows a different behaviour in parallel execution. Savings in runtime with an increasing number of nodes are only achieved with the SCI interconnect (cluster C) while using Fast- and GigabitEthernet (cluster

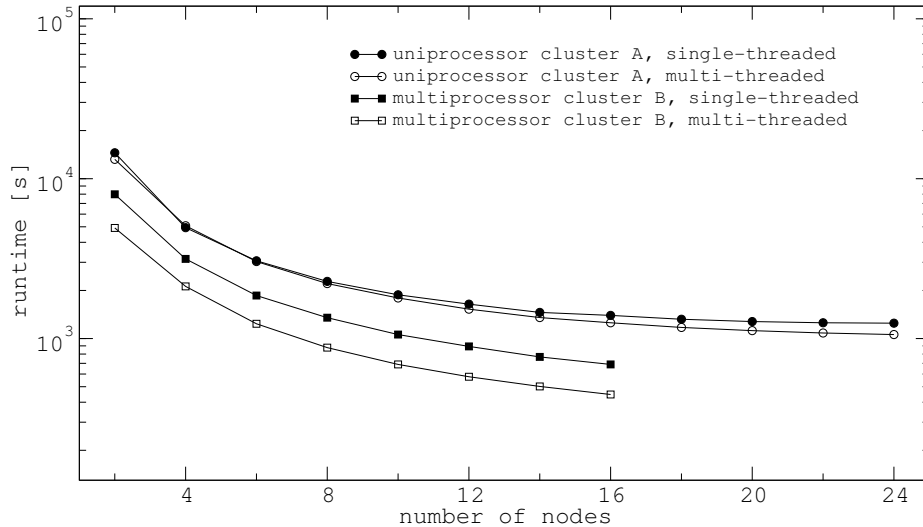


Fig. 5. Runtimes of the first phase updating the iterators according to (1) with single-threaded implementation and multi-threaded using task pool teams with 2 worker threads.

A and B) the runtimes increase or remain constant. With the optimizations this behaviour completely change and first of all becomes more independent from the different interconnects. The most significant results are achieved using task pool teams on multiprocessor clusters (B and C). For those an increased number of nodes still leads to a decrease in execution time.

5 Conclusions

We have considered an implementation of the master equation approach to simulating diffusion on random Sierpinski carpets. As iterating the master equation requires a huge amount of computer RAM, we have favoured a parallel implementation. However, due to the randomness in the construction of the structures, a parallel implementation shows highly irregular communication patterns that demand adequate strategies for implementing efficient boundary updates. In comparison with a reference implementation that uses MPI communication operation directly, we have analyzed four implementations. The first introduces the boundary collect strategy, collecting small messages and sending them as one large MPI message. The second two use the concept of task pool teams together with synchronous and asynchronous communication operations. The last extends the use of task pool teams to a more computational expensive part of the algorithm. We observe that on high latency communication networks, such as Fast- and GigabitEthernet, the savings due to the boundary collect strategy are most important. However, with an increasing number of nodes and taking SMP

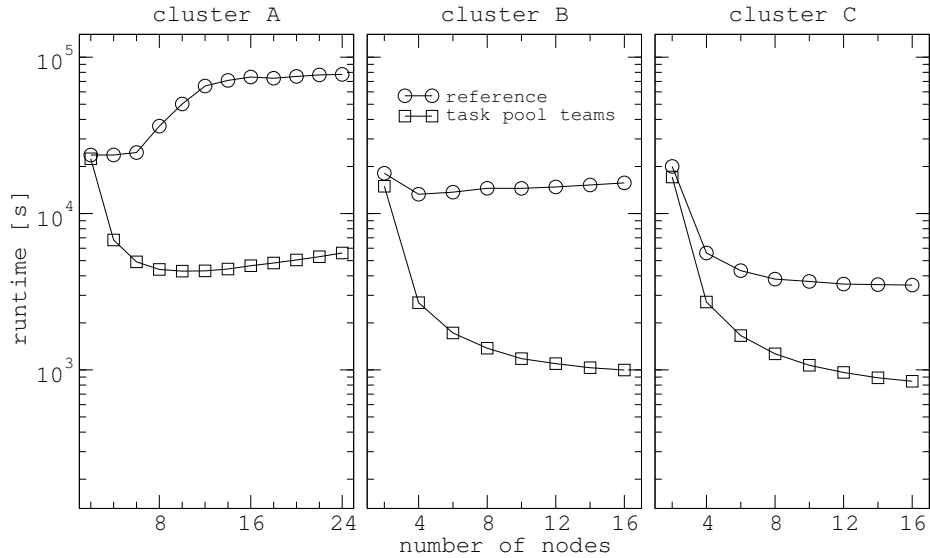


Fig. 6. Overall runtimes of the reference implementation and with the optimizations using task pool teams.

clusters into account, the use of task pool teams can result in a further reduction of the boundary update time of about an order of magnitude. Altogether, using the task pool teams concept we achieved a highly efficient implementation for the utilization of multiprocessor clusters.

References

1. Mandelbrot, B.B.: *Fractals - Form, Chance and Dimension*. W. H. Freeman, San Francisco (1977)
2. Franz, A., Schulzky, C., Seeger, S., Hoffmann, K.: An efficient implementation of the exact enumeration method for random walks on Sierpinski carpets. *Fractals* **8**(2) (2000) 155–161
3. Anh, D.H.N., Hoffmann, K.H., Seeger, S., Tarafdar, S.: Diffusion in disordered fractals. accepted by *Europhys. Lett.* (2005)
4. Chandra, S.: *Superionic Solids: Principles and Applications*. North-Holland, Amsterdam (1981)
5. Hippold, J., Runger, G.: Task pool teams: A hybrid programming environment for irregular algorithms on smp clusters. to appear in: *Concurrency and Computation: Practice and Experience* (2006)
6. Reis, F.D.A.A.: Diffusion on regular random fractals. *J. Phys. A: Math. Gen.* **29**(24) (1996) 7803–7810
7. ben Avraham, D., Havlin, S.: *Diffusion and Reactions in Fractals and Disordered Systems*. Cambridge University Press, Cambridge, UK (2000)
8. de Gennes, P.G.: La percolation: Un concept unificateur. *La Recherche* **7**(72) (1976) 919–927