Hotspot: a Framework to Support Performance Optimization on Multiprocessors

Fernando G. Tinetti[1], **Andres More**[2]
[1]III-LIDI, Fac. De Informática, UNLP
Comisión de Inv. Científicas Prov. de Bs. As.
1900, La Plata, Argentina
[2]Intel Corp., Argentina Software Design Center
5000, Córdoba, Argentina

Abstract— *High Performance Computing programs are usually developed by domain specialists, and they are not always experts on performance optimization. Those experts usually have to rely on a systematic and unattended (and/or partially guided) method to support performance analysis and optimization. This paper presents a framework that simplifies the job, including several case studies as a validation step and as a proof of concept. The main objective is to make the data recollection task straight-forward, allowing to focus on experimentation and optimization. The framework gathers context information about the program and the underlying system, detailing scaling behavior, execution profile, resource utilization, and bottlenecks. In a nutshell, this work contributes with a performance report generator for OpenMP programs.*

Keywords: Computer Aided Analysis, Performance Analysis, High Performance Computing, Parallel Processing

1. Introduction

High Performance Computing (HPC) optimized code can run several orders of magnitude faster than a naïve implementation [1]. This implies that investing on program parallelization and optimization can lead to large gains in productivity for processing intensive programs, which usually require multiple runs to simulate or solve a problem.

Parallel programming is used to fully take advantage of available computing power; incrementing both complexity and the required effort when implementing, debugging and optimizing any program [2]. The optimization and parallelization tasks are usually performed ad-hoc, without knowledge of available tools and their capabilities. Lacking the use of quantitative information to direct optimizations is another (not minor) problem underlying such parallelization and optimization work. Many times, well-known algorithms are (re)implemented instead of using already heavily optimized libraries, with proven correctness and a supporting community.

This paper is focused on simplifying the tedious and error-prone task of performance analysis. The methodology implies repetitive execution of multiple executions under different input conditions, taking advantage of the expertise on different tools to instrument behavior and understand resource utilization internals. The framework runs benchmarks on the system, apply profiling tools, and graphically show results into a final detailed report with statistical data on scaling and bottlenecks.

The rest of this paper is structured as follows: in Section 2 we introduce performance analysis background and related work. Section 3 the problem we aim to solve is reviewed, and in Section 4 we cover our proposed solution and the implemented framework. Finally, in Section 5, we apply the framework to several well-known computing kernels before concluding remarks are discussed.

1.1 Related Work

Performance optimization is relevant to almost any discipline involving computing processing. An introduction can be found in [12], a general revision in [13], just to mention a few. A proposal of the direction of the state-of-the-art in [14]. Useful reus-able design patterns in [15]. Several ideas for gathering relevant performance information can be found in [16], and an effort on automatic optimization in [17]. Further details on the state-of-the-art can be found in [18].

2. Performance Analysis

Performance is characterized by a metric which must be quantifiable units of achieved work in contrast with utilized time and resources. Metrics allows the relative comparison of systems, and also understand a reconfigured system behavior.

There are several laws that provide some guidance for estimating performance gains when incrementing computing resources. The so called Amdahl Law [7] pro-vides some insight in potential speedup according to the serial and parallel parts of an program. Gustafson [8] establishes something similar, but taking into account how many times we can compute the same problem and, also, the fact that increasing processing facilities implies increasing problem size (thus, also increasing processing requirements). A *raw* representation on how parallelism ratios impact speedup when doubling processing units is shown in Fig. 1.

	0.1	0.3	0.5	0.8	0.9	0.95
1	1.00	1.00	1.00	1.00	1.00	1.00
2	1.05	1.14	1.33	1.60	1.82	1.90
4	1.08	1.23	1.60	2.29	3.08	3.48
8	1.10	1.28	1.78	2.91	4.71	5.93
16	1.10	1.31	1.88	3.37	6.40	9.14
32	1.11	1.32	1.94	3.66	7.80	12.55
64	1.11	1.33	1.97	3.82	8.77	15.42
128	1.11	1.33	1.98	3.91	9.34	17.41
256	1.11	1.33	1.99	3.95	9.66	18.62
512	1.11	1.33	2.00	3.98	9.83	19.28
1024	1.11	1.33	2.00	3.99	9.91	19.64
2048	1.11	1.33	2.00	3.99	9.96	19.82
4096	1.11	1.33	2.00	4.00	9.98	19.91
8192	1.11	1.33	2.00	4.00	9.99	19.95
16384	1.11	1.33	2.00	4.00	9.99	19.98
32768	1.11	1.33	2.00	4.00	10.00	19.99
65536	1.11	1.33	2.00	4.00	10.00	19.99

	0.1	0.25	0.5	0.75	0.9	0.95
1	1	1	1	1	1	1
2	1	1	2	2	2	2
4	1	2	3	3	4	4
8	2	3	5	6	7	8
16	3	5	9	12	15	15
32	4	9	17	24	29	30
64	7	17	33	48	58	61
128	14	33	65	96	115	122
256	27	65	129	192	231	243
512	52	129	257	384	461	486
1024	103	257	513	768	922	973
2048	206	513	1025	1536	1843	1946
4096	411	1025	2049	3072	3687	3891
8192	820	2049	4097	6144	7373	7782
16384	1639	4097	8193	12288	14746	15565
32768	3278	8193	16385	24576	29491	31130
65536	6555	16385	32769	49152	58983	62259

Fig. 1: Speedup Limits According to Amdahl (left) and Gustafson (right)

The procedure to perform optimizations involves cycles of measurement, bottleneck identification, optimization, and gain comparison and analysis. Every decision should be made using strongly meaningful data in order to focus the work on maximizing impact. In the case the measurement of a metric has deviations, it is required to take several samples and use an average to avoid transient noise. Usually, a longer execution time of a problem will help to stabilize results. If the system has dynamic configuration, then results reproduction is non-trivial. On this case, it is advisable to run together old and new versions of the program for a direct comparison.

2.1 Performance Tools

There are many available tools for performance analysis [19]. Tools work at different levels of abstraction: from hardware-based event counters, to resource monitoring inside operating system kernels, code and binary instrumentation, up to the simple utilization of runtime as a reference metric.

Benchmarks are specially-built synthetic programs, or even a predefined set of real-world programs that provide a reference figure to compare performance. The required features for a benchmark are portability, simplicity, stability and results reproduction. It is also required that execution time is reasonable, and problem size can be adjusted to keep applicability over time and the evolution of technologies. High Performance Computing Challenge (HPCC) [22] is a package that includes several HPC benchmarks and provides multiple metrics at once.

An incremental approach is proposed in this paper, applying tools in the sequence shown in Table 1, every step adding more information and detail to the analysis. The overall system capacity extracted using the HPCC benchmarks allows to set a practical limit on performance of the system being used. The timing of workload executions allows to understand their runtime deviation. The execution profile will show call flows and bottlenecks at function, source code line and assembly levels. The program scaling behavior shows speedup trends when incrementing either problem size or processing units. A system profile show impact on available system resources. At last, low-level reports on vectorization details and hardware counter status can lead to fine-tuning needs at CPU instruction level.

Table 1: Performance Tools Incremental Application.

Information	Tools
Overall System Capacity	HPCC Benchmark
Workload Execution	time, gettimeofday()
Execution Profile	gprof, perf
Program Scaling	gprof, perf
System Profile	Perf
Vectorization	Compilers
Hardware Counters	Perf

3. Optimization Problems

The approach in this paper identifies three different dimensions in which the problems have to be addressed: performance analysis, optimization methods implementation, and supporting infrastructure.

3.1 Performance Analysis

The performance analysis has multiple challenges:

- Human interaction is always a source of involuntary mistakes. Miss investing valuable time in tasks that can be automated. Usually, one person should run tests, gather results, and draw charts in a performance report to rely on during subsequent analysis. Absolute discipline is mandatory as the main time-consuming task is to execute the same program under different configurations to record its behavior.
- Required expertise in tools. Learning about the proper use of the multiple tools takes considerable time. However, those tools povide high quality information necessry to take data-driven decisions at the time of optimization efforts. The analysis requires the correct use of statistics to average results, discard noise and outliers, and understand limits on potential improvements.
- Data gathering and representation of results. The analysis requires the gathering of supporting data about both the program and the system being analyzed. Idetifying metrics, how to get them and even how to represent them is non-trivial and requires time and expertise.
- Early optimization. An optimization made in the wrong place implies wasted effort and potentially little impact on the overall execution runtime of the program.
- Naïve implementation of algorithms. Direct implementation of an algorithm may guarantee correctness but not efficiency without first understanding underlying low-level details of the computer architecture. Reuse of portable math libraries guarantee both correctness and

performance with only a minimum effort on learning how to apply them.

And many of them (maybe all) are interraleted, so they cannot be treated indepently of each other.

3.2 Optimization Methods Implementation

The usual optimization methods target different aspects of a given program:

- Code. The source code is analyzed to improve jump prediction by pre-fetching units, in-lining frequently used routines to avoid unnecessary returns, align code and unroll loop to make program steps easy to map into vectorised instructions, among many other aspects.
- Execution. The generated assembly level code is inspected to verify the usage of lightweight and vectorised instructions, trying also to reduce the number of used registers.
- Memory. Program structures are analyzed to minimize cache failures, align inner components and improve the locality of data to enhance the memory hierarchy performance.
- Prefetching. Use of pre-fetching instructions are analyzed to help prediction unit to be more effective.
- Floating point arithmetic. Given a problem it can be considered to relax representation assumptions following standards, in order to exchange accumulated error and final precision by processing speed.

And, again (as in the case of the performance analysis explained above), many of them (maybe all) are interraleted, so they cannot be treated indepently of each other.

3.3 Supporting Infrastructure

Clearly, performance analysis needs (semi)automated support, the following are some requirements that any framework should include:

- Reusability. The framework should be applicable to a wide range of programs, without requiring its modification. Its deployment should only depend on the same tools that any user may need to gather performance-related information manually.
- Configuration. The framework should be configurable and parameters should include how to compile and execute the program, the input range to consider, and the number of repetitions to check for stability, among other details.
- Portability. The framework should be implemented in a portable (and maybe interpreted) language, to allow easy review and the expansion with new tools, new data gathering experiments or the modification on how things are being done.
- Extensibility. The framework should be designed for extension from scratch, incorporating new tools, charts o sections on a final report should be an almost trivial task given the user already knows how to gather the metric manually.
- Simplicity. The framework should reuse the same tools available at system level to a regular user. It should generate log files of all the issued commands and their output so any user can check them if required. It should be possible to use the framework to run overnight and allow incremental optimization.

4. Proposed Solution

We propose an approach that combines a generic procedure on top of an automated framework that takes care of running the program multiple times, applying performance analysis tools, identifying bottlenecks, gathering metrics and representing results graphically into a performance report. This allow the user to solely focus on optimization of the identified bottlenecks, freeing him of performance-related data recollection work.

4.1 Method

The performance analysis procedure needs to be done incrementally to guarantee progress no matter the allowed time for the task. We propose then a process which starts analyzing system computing power. The computing power is taken into account to follow a sequence of improvement cycles of execution of the program for gathering performance metrics and revealing bottlenecks. Thus, optimization efforts can be focused on those identified bottlenecks, in order to maximize overall impact. The following steps reflects how the process should be done:

1) Run the HPCC benchmark to get insights on overall system performance.
2) Run the program multiple times using the same workload to check deviations.
3) Establish a baseline using geometric mean to avoid measurement noise.
4) Run the program scaling problem size to dimension its behavior.
5) Run the program scaling computing resources to dimension its behavior.
6) Extract execution profile of the program
7) Extract system profile while running the program.

4.2 Framework

A framework named hotspot (https://github.com/more-andres/hotspot) was built, implementing the previous procedure. The automation behaves exactly like a regular user running commands and checking their output. It executes tools like gcc, make, prof, gprof, pidstat (among others) to gather relevant information and finally using the Latex typesetting environment to compile a human-friendly report including the data and associated charts.

Currently, only GNU/Linux systems with kernels above 2.6.32 and OpenMP threaded programs are supported out-of-the-box. Old kernel versions do not properly support the tools used to identify bottlenecks at assembly level. OpenMP threaded programs easily let to change the number of processing units in order to understand scaling behavior.

At a lower abstraction level, the framework is designed with two simple object class hierarchies as shown in Fig. 2. The first one keeps the main engine which runs the second one as independent sections that gather information and report back metrics. The latter one can be extended to add more sections to the final report. The implementation

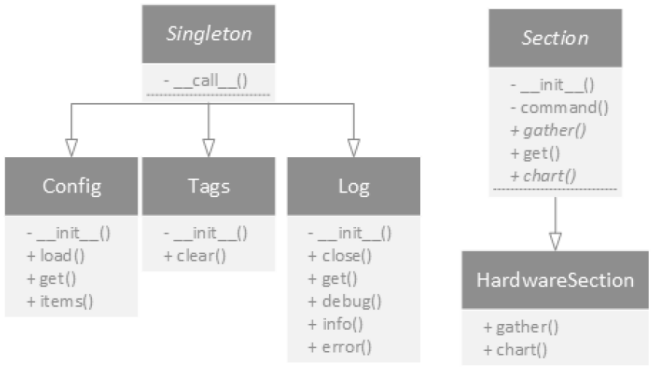

Fig. 2: Speedup Limits According to Amdahl (left) and Gustafson (right)

defines multiple objects as part of these class hierarchies. The components of the framework engine are:

- Singleton: extended by other object to guarantee unique instance.
- Tags: storage of keywords to be replaced at the report template.
- Log: manager used to structure output messages.
- Config: manager used to read and share configuration attributes.

And the components that hold the responsibility and knowledge of running experiments and gathering relevant metrics are the following:

- Section: extended by sections to be included in the report.
- HardwareSection: hardware-related information such as used CPU and Memory.
- ProgramSection: program-related information such as used problem input range.
- SoftwareSection: program-related information such as used compiler and libraries.
- SanitySection: basic check on that the workload can run without issues.
- BenchmarkSection: runs HPCC and gather relevant metrics.
- WorkloadSection: reports on program footprint and its stability.
- ScalingSection: reports on problem size scalability.
- ThreadsSection: reports on computing scalability.
- OptimizationSection: reports on program behavior under compiler optimizations.
- ProfileSection: reports on program execution profile, call flow and bottlenecks.
- ResourcesSection: reports on the use of system resources while the program runs.
- AnnotatedSection: reports annotated bottlenecks mapping code to assembly.
- VectorizationSection: reports loops being vectorised or not and the impediments
- CountersSection: reports hardware counter status.
- ConfigSection: reports used configuration for the overall framework execution.

The framework uses a per-program hidden directory to keep record on executions, classified per timestamp and caching the different results to avoid long waiting times.

A configuration file is used to define the framework parameters to handle any program. It is important to note that the framework needs to know how to run, compile and instrument each program as part of the configuration. The framwork will rely on variables that hold problem input size, numbers of OpenMP threads to use and even compiler flags to instrument binaries.

4.3 Operation and Report

At high level the design mimics how a user manually interacts with the system. The framework depends on tools available on the system, the program, and its matching configuration plus the LaTeX typesetting system.

After reading the configuration, the workload is executed multiple times and its wall time is checked for stability by charting a histogram using as a baseline a normally distributed curve. The geometric mean is extract to be used as a reference in future executions. As second step, the program is executed over the full range of input size and available computing resources, with this scaling charts are generated, plotting ideal scaling as a comparison approach as well. Using this scaling information the potential speedups are computed according to Amdahl and Gustafson laws. Then using expanded debugging information bottlenecks are identified on logical, source and assembly program levels. One last execution is done while at the same time resource usage is recorded to understand system bottlenecks. After all these executions the gathered information is used to generate a detailed PDF report to support the performance optimization task.

The general considerations followed when building the report include:

- Format similar to a scientific paper.
- Hyperlinks to full logs of tools output.
- Brief explanation of each section and chart objective.
- Inclusion of ideal trends and behavior in charts.

- References to base bibliography.

Several pieces of information are given, such as: a) Abstract: a brief summary introducing the framework and the location of the supporting output for detailed review, b) Content: reduced table of content with direct links to the information, c) Program: details of the program under analysis, timestamp of the analysis and input parameters used during tests, d) System computing poer: beyond specifics about hardware and software on the system, the framework includes the metric reported by an HPCC execution, e) Workload: working set and in-memory structures, histogram of execution to understand its deviation and the geometric mean used as a baseline. Also a chart checking how the compiler optimization levels improve time.

5. Case Study - Examples

This section will showcase the hints that the framework can lead to, but will not include solutions to those as they are out-of-scope. The framework was used to analyze three well-known compute kernels: a) Mtrix Multiplication: a naïve implementation of dense matrix multiplication., b) 2D Heat Distribution: a naïve implementation of iterative heat distribution, and c) Mandelbrot Set: a naïve implementation of a recursive fractal algorithm.

The HPCC benchmarks executed and its multiple reference metrics are included to be used as top reference of system capabilities, shown in Table 2.

Table 2: Performance Metrics Reported by HPCC

Benchmark	Value	Unit
hpl	0.00385977 TFlops	tflops
dgemm	1.03413 GFlops	mflops
ptrans	0.997656 GBs	MB/s
random	0.0274034 GUPs	MB/s
stream	5.19608 MBs	MB/s
fft	1.2658 GFlops	MB/s

5.1 Examples

Fig. 3 shows the initial performance and metrics report provided by our tool. As a first step towards optimization,

Execution time:
(a) problem size range: `1024 - 2048`
(b) geomean: `4.61694` seconds
(c) average: `4.61757` seconds
(d) stddev: `0.07678`
(e) min: `4.52168` seconds
(f) max: `4.73660` seconds
(g) repetitions: `8` times

Fig. 3: Initial Matrix Multiplication Report

the different compiler optimization options are also reported, shown in Fig. 4. We have seen almost the same *pattern*

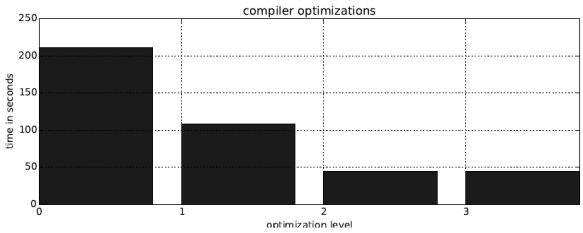

Fig. 4: Compiler Optimizations on Matrix Multiplication

for the different compiler optimization levels on several programs, e.g. a huge improvement for -O1 and -O2, and relatively small improvement for -O3. Fig. 5 shows the information about serial and parallel fractions as well as the limits of parallelization according to de the so called Amdhal and Gustafson laws.

1. Parallel Fraction: `0.52260`.
   Portion of the program doing parallel work.
2. Serial: `0.47740`.
   Portion of the program doing serial work.

Optimization limits can be estimated using scaling laws.

1. Amdhal Law for 1024 procs: `2.09466 times`.
   Optimizations are limited up to this point when scaling problem size. [2]
2. Gustafson Law for 1024 procs: `535.61564 times`.
   Optimizations are limited up to this point when not scaling problem size. [3]

Fig. 5: Matrix Multiplication Parallelization Reports

The tool on the naïve implementation of heat distribution in 2 dimensions identifies:

- Workload is stable, although deviation is greater than in the previous case.
- Compiler optimizations have little impact.
- Execution time does not grow monotonically when scaling input size; it neither decreases when adding more computing units.
- Parallelism only reaches 65%, therefore speedup has a limit on 2.9x.
- There are two main bottlenecks with 30% and 15% of overall execution time.
- CPU utilization is not constant, and loops are not vectorised.

On the third example, the naïve implementation of Mandelbrot sets [23], the tool reports:

- Structures used to represent complex numbers are aligned, without holes that may consume cache memory.
- Workload is stable.
- 50% of the time is spent on the same line of code.
- There are already optimized cycles using the movss and adds vectorised in-structions.

6. Conclusions and Further Work

The performance optimization process is not trivial and requires disciplined analysis of used resources and gathering of metrics characterizing program behavior. This works reviews the development of a supporting framework that streamlines the process by running in unattended mode and generating a report to direct optimization efforts. The generated report combines multiple tools to identify bottlenecks at function, source, and assembly level (e.g. vectorization). It also records system configuration and resource usage. All of this is offered as an unattended automated task before undergoing the program optimization analysis. We hence propose a systematic procedure supported with an automated framework that is suitable for both newcomers and experts. It also allows the exchange of standardized performance reports between research and development groups.

Extension possibilities on this framework are straightforward considering new sections can be added to the report including the application of new tools, charts or contextual information. The application of this framework to a well-known open source program will size its usefulness and provide feedback on both the procedure and the generated report. At last, it may be interesting to move from a static report format to a dynamic one offering pivot tables and charts that can be reconfigured easily. This may be better achieved using HTML5 technologies over a browser for instance.

References

[1] A. More. A Case Study on High Performance Matrix Multiplication. Technical Report, Universidad Nacional de La Plata, 2008. Available at http://mm-matrixmultiplicationtool.googlecode.com/files/mm.pdf

[2] P. E. McKenney. Is Parallel Programming Hard, And, If So, What Can You Do About It? kernel.org, 2010.

[3] Intel Corporation. Intel Math Kernel Library. Reference Manual. Intel Corporation, 2009.

[4] R. Garabato, V. Rosales, A. More. Optimizing Latency in Beo-wulf Clusters. CLEI Electron. J., 15(3), 2012.

[5] J. Jeffers, J. Reinders. Intel Xeon Phi Coprocessor High Performance Programming. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

[6] A. More. Lessons Learned from Contrasting BLAS Kernel Implementations. In XVIII Congreso Argentino de Ciencias de la Computación, 2013.

[7] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In Proceedings of the April 18-20, 1967, spring joint computer conference, AFIPS '67 (Spring), pages 483-485, New York, NY, USA, 1967. ACM.

[8] J. L. Gustafson. Reevaluating Amdahl's Law. Communications of the ACM, 31:532-533, 1988.

[9] A. H. Karp, H. P. Flatt. Measuring parallel processor performance. Commun. ACM, 33(5):539-543, May 1990.

[10] P. J. Fleming, J. J. Wallace. How not to lie with statistics: the correct way to summarize benchmark results. Commun. ACM, 29(3):218-221, March 1986.

[11] K. Atkinson. An Introduction to Numerical Analysis. Wiley, 2 edition, 1989.

[12] C. U. Smith. Introduction to software performance engineering: origins and outstanding problems. In Proceedings of the 7th international conference on Formal methods for performance evaluation, SFM'07, pages 395-428, Berlin, Heidelberg, 2007. Springer-Verlag.

[13] J. C. Browne. A critical overview of computer performance evaluation. In Proceedings of the 2nd international conference on Software engineering, ICSE '76, pages 138-145, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[14] M. Woodside, G. Franks, D. C. Petriu. The future of software performance engineering. In 2007 Future of Software Engineering, FOSE '07, pages 171-187, Washington, DC, USA, 2007. IEEE Computer Society.

[15] T. Mattson, B. Sanders, B. Massingill. Patterns for parallel programming. Addison-Wesley Professional, First edition, 2004.

[16] K. A. Huck, O. Hernandez, V. Bui, S. Chandrasekaran, B. Chapman, A. D. Malony, L. Curfman McInnes, B. Norris. Capturing performance knowledge for automated analysis. In Proceedings of the 2008 ACM/IEEE conference on Supercomputing, SC '08, pages 49:1-49:10, Piscataway, NJ, USA, 2008. IEEE Press.

[17] T. Margalef, J. Jorba, O. Morajko, A. Morajko, E. Luque. Performance analysis and grid computing. In V. Getov, M. Gerndt, A. Hoisie, A. Malony, B. Miller, editors, Performance analysis and grid computing, chapter Different approaches to automatic performance analysis of distributed applications, pages 3-19. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[18] F. Wolf B. Mohr. Automatic performance analysis of hybrid mpi/openmp applications. J. Syst. Archit., 49(10-11):421-439, November 2003.

[19] B. Gregg. Linux Performance Analysis and Tools. Technical report, Joyent, February 2013.

[20] J. D. McCalpin. Memory bandwidth and machine balance in current high perfor-mance computers. IEEE Technical Committee on Computer Architecture (TCCA) Newsletter, Dec 1995.

[21] U. Drepper. What Every Programmer Should Know About Memory. Tech-nical report, Red Hat, November 2007.

[22] P. Luszczek, J. J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. Mccalpin, D. Bailey, D. Takahashi. Introduction to the HPC Challenge Benchmark Suite. Technical report, 2005.

[23] B. B. Mandelbrot, D. E. Passoja, editors. Fractal aspects of materials: metal and catalyst surfaces, powders and aggregates: extended abstracts, volume E-4 of Materials Research Society extended abstracts, Pittsburgh, PA, USA, 1984. Materials Research Society.