UNITED STATES PATENT AND TRADEMARK OFFICE

In the Ex Parte Reexamination of

U.S. Patent No. 7,257,582

LOAD BALANCING WITH SHARED DATA

Filing Date: February 27, 2003
Issue Date: August 14, 2007

Reexamination Control Number 90/015,337

DECLARATION OF DR. DANIEL ABADI IN SUPPORT OF PATENT OWNER

TABLE OF CONTENTS

I. INTRODUCTION AND QUALIFICATIONS 1

II. RELEVANT LEGAL STANDARDS 5

III. THE '582 PATENT 8

IV. PRIOR ART RELIED ON IN THE REXAMINATION 10

A. IYERS 11

B. REEVE 13

V. THE '582 PATENT AND REEVE ARE DIRECTED TO DIFFERENT FIELDS OF ENDEAVOR 14

VI. GROUND 1: CLAIMS 1-14 ARE NOT OBVIOUS IN VIEW OF IYERS AND REEVE 27

A. I. IYER SIZES AND COUNTS WORK PARTITIONS TO MATCH PROCESSOR COUNT, I.E., NO OVER-SUBSCRIPTION AND NO DYNAMIC "REPEAT/CLAIM" POOL 27

B. THE OFFICE ACTION'S GRAB-BASED INFERENCE FROM REEVE DOES NOT "CARRY OVER" TO IYER WITHOUT RE-ARCHITECTING IYER'S WORK-DISTRIBUTION MODEL 31

C. THE '582 PATENT EXPRESSLY TEACHES AN OVER-SUBSCRIPTION + REPEATED ALLOCATION MODEL THAT IS ABSENT FROM IYER'S SORTING FLOW 35

D. IYER'S "ONLY TWO PROCESSORS ARE NEEDED" PASSAGE CONCERNS PARTITION-BOUNDARY DETERMINATION, NOT REPEATED PROCESSING OF ADDITIONAL UNPROCESSED PARTITIONS 37

E. CLAIM 1 STEP (A) REQUIRES FILE ALLOCATION BASED PARTITIONING OF FILE RECORDS, AND NEITHER IYER NOR REEVE TEACHES THAT ELEMENT 40

F. A POSITA WOULD NOT HAVE BEEN MOTIVATED TO COMBINE IYER AND REEVE TO ARRIVE AT THE CLAIMED SYSTEM 43

G. DEPENDENT CLAIMS 2, 5,6, 7, 9, 10, 11 AND 12 RECITE ADDITIONAL, NARROWING TECHNICAL CONSTRAINTS THAT ARE NOT TAUGHT OR SUGGESTED BY IYER IN VIEW OF REEVE 49

(a) Claim 5. "An accumulation of output records from said subtasks in an arbitrary order" is an output ordering requirement, not a statement about asynchronous completion 50

(b) Claims 6 and 7. A disk bus connects a processor to a local disk, not to a storage area network or network attached storage. 53
(c) Claims 9 and 10. A post run cleanup routine that emits statistics does not teach that the computer executable process is a statistical analysis or report creating process....53
(d) Claim 11. Generic disk storage or bus architecture is not a disclosure of a database query 55
(e) Claims 2 and 12. Processor 1 in Iyer does both partitioning and subtask processing. Placing it on a separate mainframe would violate Iyer's homogeneous processor assumption.....57

VII. GROUND 2: CLAIMS 1-14 ARE NOT OBVIOUS IN VIEW OF REEVE ...59

A. DIFFERENT OBJECT BEING PARTITIONED. REEVE PARTITIONS AN ITERATION SPACE INTO TILES, NOT A PREEXISTING FILE'S RECORDS INTO FILE RANGE PARTITIONS BASED ON FILE ALLOCATION 59
B. DIFFERENT PARTITIONING TARGET. THE 582 PATENT DIVIDES DATA, WHILE REEVE DIVIDES AND SCHEDULES COMPUTATION 61
C. DIFFERENT MEANING OF FIRST COME FIRST SERVED. REEVE'S FCFS ASSIGNS ITERATION SPACE TILES TO THREADS, WHILE CLAIM 1 USES FCFS FOR REPEATED ALLOCATION OF UNALLOCATED FILE PARTITIONS AFTER COMPLETION....63
D. REEVE DOES NOT DISCLOSE THE 582 PATENT'S OUTPUT MODEL, WHICH COMBINES DISCRETE SUBTASK OUTPUTS INTO A RESULT OUTPUT FILE OR FILES....65
E. THE OFFICE ACTION'S REEVE ALONE MAPPING IS STRUCTURALLY FLAWED BECAUSE IT SUBSTITUTES GENERIC PARALLELISM FOR THE SPECIFIC FILE PARTITION LOAD BALANCING ARCHITECTURE OF CLAIM 1 67
F. DEPENDENT CLAIMS 3, 4, 5, 9, 10, AND 11 ARE NOT TAUGHT OR SUGGESTED BY REEVE ALONE....72
(a) A. Claim 3. Reeve does not teach the additional step of merging subtask outputs in step (e) 73
(b) Claim 4. Reeve does not teach a one-to-one correspondence between input records and output records....75
(c) Claim 5. Reeve does not teach an accumulation of output records from subtasks in an arbitrary order....76

(d) Claims 9 and 10. A post run cleanup routine that emits statistics does not teach that the computer executable process is a statistical analysis or report creating process.....78
(e) Claim 11. Reeve does not teach that the computer executable process includes a database query.....79
VIII. CONCLUSION 81
IX. DECLARATION.....84

I. INTRODUCTION AND QUALIFICATIONS

1. I, Daniel Abadi, submit this declaration on behalf of Intellectual Ventures I LLC, ("Patent Owner") to provide my opinions concerning the validity of the claims of U.S. Patent No. 7,257,582 (the "'582 patent," Ex.1001), in support of Patent Owner's Response to the Non-Final Office Action issued in Re-examination Control Number 90/015,337.

2. I am more than eighteen years of age, and I am a citizen of the United States, currently residing in the State of Maryland.

3. In formulating my opinions, I have relied upon my knowledge, training, and experience in relevant art. My qualifications, along with a more detailed summary of my background and publications, are stated more fully in my curriculum vitae, which has been provided as Exhibit 2012. Here, I provide a concise summary of my qualifications.

4. I am a Professor in the Department of Computer Science at the University of Maryland in College Park. I was hired as the Darnell-Kanal Chair, so technically I am the Darnell-Kanal Professor of Computer Science at the University of Maryland, College Park.

5. In addition to my two decades in academia, I have also been involved in several start-ups. My PhD thesis was commercialized by a company called Vertica for which I served as a consultant to help with the technology transfer. In addition, the

HadoopDB project which was developed in my lab at Yale University (back when I was on the computer science faculty there) was commercialized by a company called Hadapt. I served as co-founder and Chief Scientist at Hadapt during the company's entire existence until it was acquired by Teradata.

6. Vertica built one of the first commercially successful column-stores in history. Column-stores store database tables on disk in columns instead of rows. This makes read analysis much faster (since irrelevant columns can be ignored) at the cost of increasing the time it takes to insert records (rows) into tables. My work on C-Store introduced mechanisms for decreasing the write overhead while further accelerating the read benefits through compression, introduction of operators that know how to directly process column-oriented compressed data, delaying the stitching of columns into rows and other new operators including a faster column-oriented join algorithm.

7. My work on C-Store and Vertica made tremendous impact in the industry. Today, almost every commercial database system vendor implements column-store options for their customers. Consequently, this work won the test of time award at VLDB --- the most prestigious award at one of the two leading yearly database community conferences.

8. The HadoopDB project introduced the first high-performance SQL layer on top of data stored in Hadoop (it was concurrent with Hive that had a competing

goal but was significantly slower). It used a combination of traditional database operators and MapReduce operations to process database queries.

9. The HadoopDB project had a similar impact as the C-Store project mentioned above. Now there are dozens of companies that offer SQL layers to data stored in Hadoop. Consequently, this work won yet another test of time award at VLDB.

10. My work on streaming database systems --- including Aurora and Borealis also has been impactful. Today many companies use the streaming technology to process data, as it generated using ideas introduced by the Aurora and Borealis projects. The Borealis project won yet another test of time award --- this time at CIDR --- 20 years after the seminal paper was published there.

11. I am unaware of any other member of my technical community that has won three test of time awards (though the data of who has won these awards does not appear to be maintained in a central location, so I have not been able to verify whether I am the only one or not). At the very least, this is an extremely rare feat.

12. I was named an ACM Fellow in 2020, an elite recognition of outstanding science and scholarship that is bestowed upon less than one percent of the organization's 100,000 members.

13. I have also won several other awards for my research and contributions to the database systems community. Perhaps the four most notable of these awards

are the VLDB 2007 best paper award, the 2008 Jim Gray Doctoral Dissertation Award, the 2013 VLDB Early Career Researcher Award, and the Yale Provost's Teaching Prize.

14. I have consistently advised undergraduate students, masters students, PhD students, and postdocs throughout my career. Two of my PhD students won the Jim Gray Doctoral Dissertation Award --- the highest award for a PhD dissertation given by the database systems community. The names of these two students are Alexander Thomson and Jose Faleiro.

15. My research publications have received 25000+ citations with an h-index of 55 (based on Google Scholar).

16. I am a named inventor on several patents corresponding to the areas of my professional work. Accordingly, I have a general understanding of the U.S. patent prosecution process and of the novelty and non-obviousness requirements for patentability. Example patents include:

- US Patent 10,868,865 System and apparatus to manage data using a peer-to-peer network and the blockchain (2020)
- US Patent 9,934,276 Systems and methods for fault tolerant, adaptive execution of arbitrary queries at low latency (2018)
- US Patent 9,495,427 Processing of data using a database system in communication with a data processing framework (2016)

- US Patent 9,471,711 Schema-less access to stored data (2016)
- US Patent 9,336,263 Data loading systems and methods (2016)
- US Patent 8,935,232 Query execution systems and methods (2015)
- US Patent 8,886,631 Query execution systems and methods (2014)
- US Patent 8,700,563 Deterministic database systems (2014)

17. I believe that my extensive industry experience and educational background qualify me as an expert in the relevant field of parallel and distributed data processing systems. Almost the entirety of my career has been spent working in this area, including the Borealis, C-Store, and HadoopDB projects mentioned above.

II. RELEVANT LEGAL STANDARDS

18. The opinions I am expressing in this report involve the application of my knowledge and experience to evaluate certain references with respect to the '582 patent. My formal knowledge of patent law is no different than that of any lay person. Therefore, I have requested the attorneys from Volpe Koenig, who represent Intellectual Ventures, to provide me with guidance as to the applicable patent law in this matter. The paragraphs below express my understanding of how I must apply current principles related to patent validity to my analysis.

19. I understand that a re-examination is a proceeding before the USPTO for evaluating the patentability of an issued patent's claims based on prior-art patents and printed publications.

20. I understand that the claims of a patent are reviewed from the point of view of a hypothetical person of ordinary skill in the art ("POSITA") at the time of the filing of the patent. The "art" is the field of technology to which a patent relates.

21. I understand that in determining whether a patent claim is anticipated or obvious in view of the prior art, the Patent Office must apply the *Phillips* standard to construe these claims since the '582 Patent has expired.

22. I understand that there are two ways in which prior art may render a patent claim unpatentable. First, the prior art can "anticipate" the claim. Second, the prior art can render the claim "obvious" to a POSITA. I understand that for an invention claimed in a patent to be patentable, it must not be anticipated and must not be obvious based on what was known before the invention was made.

23. For purposes of this Declaration only, I have applied the following definition of a POSITA for the '582 patent. A POSITA at the time of the invention would have had at least a bachelor's degree in computer science, computer engineering, electrical engineering, or an equivalent discipline, and approximately two to four years of practical experience designing or implementing file-processing and parallel or distributed execution systems. Such experience would include familiarity with partitioning large input files or datasets for parallel processing (for example, dividing data by byte ranges or record boundaries), distributing and scheduling work across multiple processors or networked "helping systems,"

coordinating and synchronizing subtasks, and aggregating or merging subtask outputs into a final output. A person with a master's degree in a relevant field and correspondingly less work experience could also qualify as a person of ordinary skill in the art.

24. Both myself and the Requestor's expert, Dr. Weissman exceed the requirements of a POSITA.

25. I understand that a patent claim may be invalid as "anticipated" under 35 U.S.C. § 102 if each and every limitation of the claim as construed by the court is found, either expressly or inherently, in a single prior art reference, such as a publication or patent, that predates the claimed subject matter. I also understand that a patent claim may be invalid as obvious under 35 U.S.C. § 103 if the differences between the subject matter claimed and the prior art are such that the claimed subject matter as a whole would have been obvious to a POSITA at the time the invention was made. I also understand that several factual inquiries underlie a determination of obviousness. These inquiries include (1) the scope and content of the prior art, (2) the level of ordinary skill in the field of the invention, (3) the differences between the claimed invention and the prior art, and (4) any objective evidence of non-obviousness (i.e., "secondary considerations").

26. I also understand that, when a party alleges obviousness based on a combination of references, that party should identify a reason why a POSITA would

have been motivated to combine the asserted references in the manner recited in the claims and to explain why a POSITA would have had a reasonable expectation of success in making such combinations.

27. I understand that an obviousness analysis permits the application of "common sense" in examining whether a claimed invention would have been obvious to a POSITA. For example, I understand that combining familiar elements according to known methods and in a predictable way may suggest obviousness when such a combination would yield nothing more than predictable results. I also understand a party asserting obviousness should still provide a specific motivation to combine the references as recited in the claims and explain why one would have reasonably expected to succeed in doing so.

28. I understand that one indicator of non-obviousness is when a prior art reference leads in a different direction or discourages that combination, recommends steps that would not likely lead to the patent's result, or otherwise indicates that a seemingly inoperative device would be produced.

III. THE '582 PATENT

29. The '582 patent discusses processing records in a file. It discusses several different examples of why a practitioner would want to process records in a file, including data sorting, batch data processing, and querying of data. All of these

examples involved reading through (potentially many) records, performing processing actions on them, and then returning results or report outputs.

30. Fundamentally, the '582 patent deals with data-centric processing. The system is designed to be able to read and process large amounts of data, while maximizing the ability for a set of computers running in parallel to share in the effort of doing this reading and processing.

31. Another important contribution is that the '582 patent performs dynamic load balancing across all of the computers performing data processing. The claims require dividing an input file into many different partitions (more partitions than the available computers to process it).

32. Only after these partitions have been defined and identified, data processing can begin ("When the split step terminates, the various Sub Tasks can be activated." '582, 3:64). Each computer can run a "Sub Task" that takes responsibility for processing a partition that has been previously identified.

33. Different computers maintain load balance by only accepting responsibility for one partition at a time on a first-come first-served basis. Only after it completes processing of one partition does it take responsibility for the next one. Since there are more partitions than there are computers, the load is balanced naturally. The faster computers will end up requesting responsibility for more partitions over time as they process them at a fast rate, while the slower computers will end up

requesting responsibility for fewer partitions. All computers are constantly working on their data processing tasks until there are no more partitions to process.

34. The patent also discusses how to partition records in a file so that records do not get split across partition boundaries, how to define partition ranges using byte ranges, and how to perform this partitioning process without actually having to perform an extra scan of the input file while partitioning it. It also discusses that the partitioning process may take into account various additional properties of the computers that will be participating in the data processing effort.

35. The end of the patent discusses several special case embodiments for particular use cases, including data sorting, tasks for which the expected output size is small, and tasks for which data output order does not matter. In each of these cases, optimizations can be made to the basic description described earlier in the patent in order to achieve better performance.

IV. PRIOR ART RELIED ON IN THE REXAMINATION

36. The Office Action discusses two other patents that are used in its obviousness claims with respect to the '582 patent. In addition, the Requestor submitted a declaration by Dr. Weissman. In this section I will describe each one, in turn.

A. Iyers

37. I have reviewed U.S. Patent No. 5,179,699 to Iyer ("Iyer"). Iyer is directed to parallel sorting on a multiprocessor system and, in particular, to partitioning and merging techniques intended to produce a final globally sorted output while keeping the work assigned to each processor balanced.

38. A central feature of Iyer is a deliberate one processor to one work partition model. Iyer states that the goal is to perform the sort in the shortest period of time and that this is achieved by dividing the work as equally as possible among multiple processors. (Iyer, 4:21–24). Consistent with that objective, Iyer teaches that the initial list is divided into multiple equal sized lists, and critically, that "[t]he number of such lists should be equal to the number of processors available." (Iyer, 4:25–33). In other words, Iyer starts from the premise that the number of top level work units is chosen to match the number of processors, not to exceed it.

39. Iyer then repeats the same one to one assignment principle at the partition stage. Iyer states that the second step in the sorting process is to divide the lists into partitions and that "[t]here should be one partition of elements for each available processor." (Iyer, 4:36–40). Iyer describes a static mapping in which each processor receives a dedicated partition and is responsible for processing that partition, rather than a regime where processors repeatedly request and receive additional unassigned partitions after completing initial work.

40. Iyer's merge phase is described in the same one processor to one partition manner. Iyer explains that "each of the P processors merges the lists in its partition," producing P sorted lists, and that those lists are then strung together to form the final sorted lists. (Iyer, 3:44–50). Iyer emphasizes that "each partition is nearly the same size so no processor load imbalance occurs," reflecting that Iyer's balancing technique is to assign each processor a single partition designed to contain approximately equal work. (Iyer, 2:45–49).

41. Iyer also discusses determining partition boundaries as part of defining those one to one partitions. For example, Iyer notes that only a limited number of partition boundaries may need to be determined and therefore fewer processors may be sufficient for that boundary determination computation even when more processors are available for the overall sort. (Iyer, 4:60–63). In context, this discussion concerns how the one processor to one partition mapping is defined, not a disclosure that processors execute multiple partitions sequentially or repeatedly claim additional unprocessed partitions.

42. In summary, Iyer's parallel sorting and merge technique is built around selecting the number of work units to equal the number of processors and then assigning each processor one partition to process, followed by an ordered concatenation to form the final sorted output.

B. Reeve

43. I have reviewed U.S. Patent No. 5,535,393 to Reeve ("Reeve"). Reeve is directed to a compiler and runtime framework for executing parallel program constructs by decomposing an iteration space into tiles and scheduling those tiles for execution by threads according to selectable runtime strategies.

44. At a high level, Reeve operates in the compute domain rather than the file domain (I will go into more detail on this point below). Reeve describes how a compiler and associated runtime library manage execution of a "parallel construct" by selecting and coordinating a set of threads, including selecting "more or less threads than available processors," and relying on the operating system scheduler as part of load balancing and execution. (Reeve, 31:7–13).

45. Reeve's core decomposition mechanism is iteration space tiling. Reeve describes defining regions of work using directives that specify an "AFFINITY REGION" using an index and bounds, together with a "tilesize," and a selectable execution strategy, for example "strategy={GRAB, MOD, WAVE, SLICE}." (Reeve, 32:10–16). In this framework, the unit of parallel work is a tile, meaning a bounded subregion of an iteration index space, rather than a partition of a preexisting data file.

46. Reeve also describes runtime scheduling policies for distributing tiles to threads. In particular, Reeve explains that under the GRAB strategy, "tiles are assigned to threads with a pure first come, first serve algorithm." (Reeve, 39:4–6). In

context, the FCFS rule is a tile to thread assignment policy within a runtime tiling framework for executing a parallel construct.

47. Finally, Reeve includes discussion of program lifecycle and runtime support functions, including that a routine executed when a program finishes can perform cleanup and "produce statistics and reports." (Reeve, 30:13–14). From an HPC standpoint, this is consistent with runtime telemetry or execution reporting about program performance, rather than a disclosure about the substantive output of the application being executed.

48. In summary, Reeve teaches compile time and runtime mechanisms for parallelizing and scheduling computation by decomposing an iteration space into tiles and assigning those tiles to threads under selectable strategies such as GRAB, with execution management influenced by a runtime library and the operating system scheduler. (Reeve, 31:7–13; Reeve, 32:10–16; Reeve, 39:4–6; Reeve, 39:15–17; Reeve, 30:13–14).

V. THE '582 PATENT AND REEVE ARE DIRECTED TO DIFFERENT FIELDS OF ENDEAVOR

49. I have reviewed the '582 Patent and Reeve from the perspective of a person of ordinary skill in the art at the relevant time. In my opinion, the two references are directed to different fields of endeavor, because they state so explicitly, and they also address different classes of problems, operate at different levels of the computing stack, and use fundamentally different decomposition targets.

50. The '582 Patent describes a data processing solution for distributed systems. This is indicated by the first sentence of the "FIELD OF THE INVENTION" section which states (1:10-12): "This invention relates to the field of sharing data and workload between possibly heterogeneous computer systems." Immediately after this section, the patent discusses the background of the invention where it clarifies that it is targeted for multi-computer settings:

> *"The load imposed on the **various computers** in these sites is normally the result of the activities directed specifically to these computers and the fact that one of them is especially loaded while others are relatively idle does not affect the way work is **distributed between them**. As a result, in contemporary environments, much can be gained by the parallelization of some processes and their **distributed execution** across all available computing resources in a way that both speeds the execution of the whole process and balances the load the various computers are subjected to. Recent developments introduced the concepts of Storage Area Networks (SAN) and Network Attached Storage (NAS) devices to enable efficient usage of storage resources. These technologies increase the symmetry of storage access in a **multiple computer environment**, increasing the benefits that can be expected from Such a parallelization."*

51. Even when the patent discusses that "the plurality of processors can all be parts of a single multiprocessor" (2:36-37), it still indicates that at least one of them is on "at least one other computer" (2:36).

52. In contrast, Reeve discusses processing solutions for single (non-distributed) multiprocessor systems. E.g., it states explicitly that that the object of the

patent is to: "provide a computer run-time mechanism for parallelizing and executing a computer program on **a multiprocessor system**." (1:65-67). Furthermore, when it states that another object is to "provide an improved mechanism for executing programs on parallel processing computers, including those which are massively parallel" (1:59-62) --- even though the word computer is written in plural form, this is a reference to the "computers with multiple processors" mentioned earlier in the section which are single (non-distributed) multi-processor systems.

53. HPC-style parallel processing on multi-processor systems (where Reeve's contribution lies) and distributed systems (where the '582 patent lies) are different fields within computer science, consist of different communities, have different sets of conferences and journals, and do not interact frequently with each other.

54. Even if you ignore the fact that these contributions are targeted towards different fields within computer science, the actual techniques introduced by these patents are also attacking different classes of problems. The '582 Patent is directed to distributed, batch style processing of a preexisting input file stored on shared storage, where the system improves throughput by partitioning the input file into well defined ranges using file allocation and offset information, distributing partition descriptions, and then having subtasks read and process allocated partitions, repeatedly allocating additional unallocated partitions to balance load. For example, the '582 Patent

describes defining partitions as consecutive ranges by relative byte address or relative track address, and it emphasizes that partition definition does not rely on actual reading of the file because actual reading is reserved for subtasks that read the allocated partitions. (582, 3:36–41; 582, 3:47–53). This is characteristic of parallel IO and batch processing systems, where the core engineering problem is how to subdivide a shared file into independent work units and allocate those work units across multiple systems efficiently.

55. Reeve, by contrast, is directed to compiler and runtime techniques for parallelizing loop based computation by partitioning an iteration space into tiles based on directives that specify indices, bounds, and tile sizes. Reeve describes defining regions such as an affinity region using index and bounds information and a tile size, and then scheduling those tiles to requesting processors. (Reeve, 32:10–16). Reeve also emphasizes that its runtime makes decisions based on the iteration space rather than the data space, which is consistent with a compiler and runtime scheduling reference concerned with loop execution, locality, and thread scheduling for compute kernels, not with file allocation metadata and partitioning of a preexisting input file. (Reeve, 39:15–17).

56. I will now explain this difference in more detail.

57. In early days of computing, computers had a single processor. When a program would be submitted to the computer for execution, the single processor would execute each instruction of the program sequentially, one after the other.

58. Over time, computers started being built with multiple processors. Furthermore, multiple computers (each with one or more processors) started becoming connected over fast networks such that they could work together to perform complex computations as a team.

59. Either way, it was no longer necessary to execute programs sequentially. On the contrary, it became highly desirable to utilize all the processors available to process a program in parallel. This often would lead to far better performance than processing the same program sequentially.

60. As the computer scientists started working on the concept of parallel computing, the research community grappled with the question of how best to get a program (or task) to execute in parallel on all the available processors while still achieving the same (correct) result as if it ran sequentially. At a high level, this task would have to be divided ("partitioned") into pieces ("partitions"), with each partition running on a different processor.

61. It became clear early on that there were two orthogonal approaches to how to partition programs to accomplish this goal. One option was to divide the code ("instructions") of the program into partitions. For example, one particular set of

instructions from the original program would be executed by processor 1, and a different set of instructions would be executed by processor 2.

62. The other option was to divide the data on which the code operates into partitions. For example, processor 1 would execute a particular set of instructions on one part of the data, and then processor 2 would execute the same set of instructions on a different part of the data.

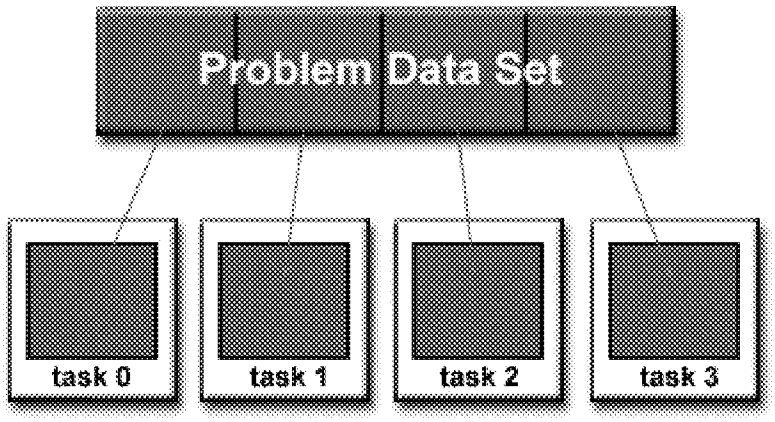
63. In the early days, the first option was often called functional decomposition, and the second option was called domain decomposition. See for example, Chapter 2 from Ian Foster's textbook "Designing and Building Parallel Programs" (https://wotug.org/parallel/books/addison-wesley/dbpp/), and also: https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial. See also Chapter 3 of the textbook" "Introduction to Parallel Computing" which further subdivides these two high level options.

64. I copy and paste here the relevant text and figures from the second reference I cited above which help explain this concept with helpful figures:

- *One of the first steps in designing a parallel program is to break the problem into discrete "chunks" of work that can be distributed to multiple tasks. This is known as decomposition or partitioning.*
- *There are two basic ways to partition computational work among parallel tasks: domain decomposition and functional decomposition.*

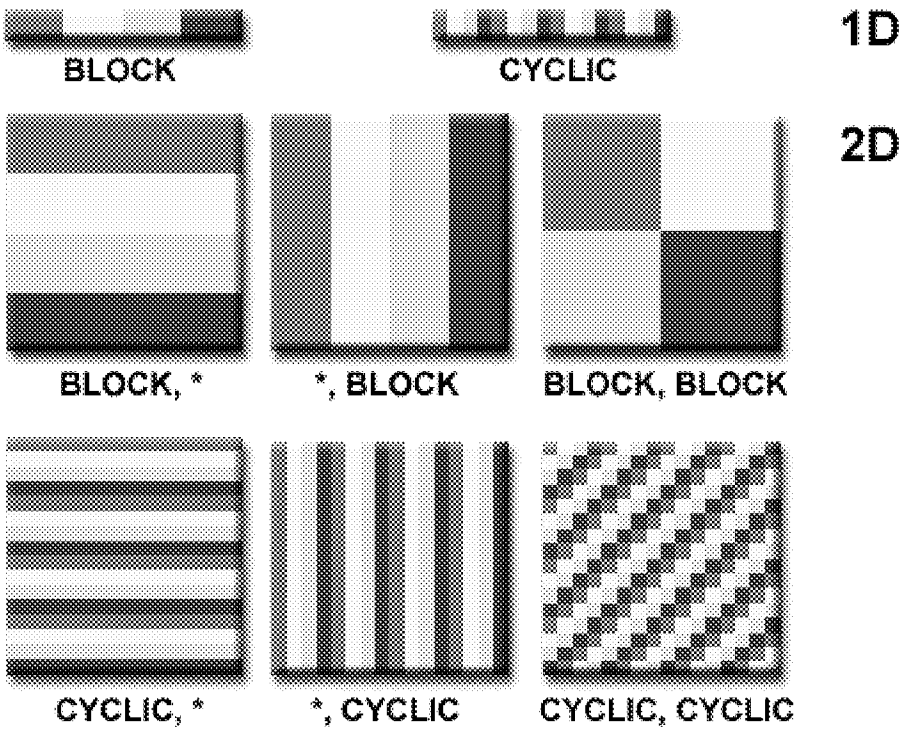
Domain Decomposition

In this type of partitioning, the data associated with a problem is decomposed. Each parallel task then works on a portion of the data.



Domain decomposition

There are different ways to partition data:

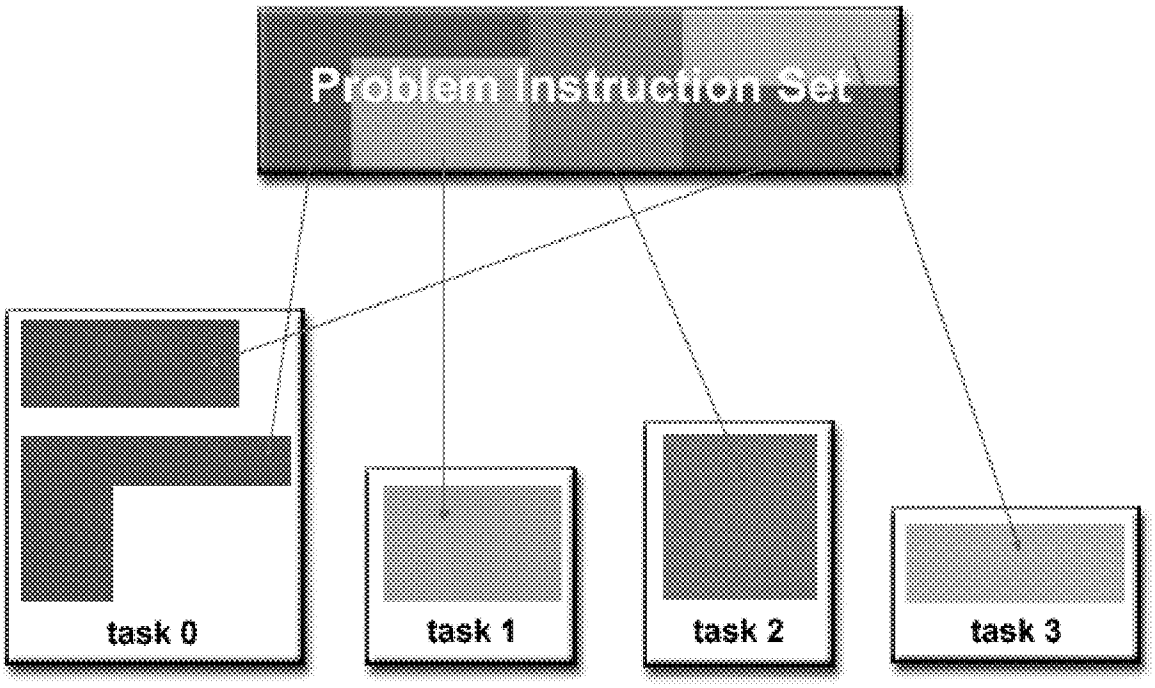


Partitioning examples

Functional Decomposition

In this approach, the focus is on the computation that is to be performed rather than on the data manipulated by the computation. The problem is decomposed

according to the work that must be done. Each task then performs a portion of the overall work.



Functional decomposition

65. The '582 patent explicitly uses a domain (data) decomposition approach. It discusses taking an input data file and partitioning it. E.g. "subdividing records of [an] input file into a plurality of partitions" (claim 1a). A record is a well-known term of the art to describe a row of data. It is synonymous with the term "tuple" which is a row of data in a database. It is clear that the patent uses the term according to this standard term of the art when it uses the term to talk about the output (e.g. "output records" in claim 5).

66. In the '582 Patent, the subtasks generally execute the same code. The only difference is that they are executing this code on different datasets. See for

example Figure 4 in which both subtasks (subtask 1 and subtask 2) sort their subset of data.

67. The Iyer patent also explicitly uses a domain (data) decomposition approach. Claim 1 states that it is a "method of partitioning P sorted lists" and discusses accessing "storage" to obtain data from these sorted lists.

68. Furthermore, Iyer is explicit that each processor is running the same code for different data. For example, claim 1 discusses "the accessing step being performed by each of the processors." In other words, each processor is doing the same accessing, but they will each read different data from the sorted lists input. See also 3:34: "p-1 processors perform this operation in parallel" --- i.e. this same operation.

69. In contrast the Reeve patent explicitly uses a functional decomposition approach. This is clear from the summary of the invention (2:29-23): "The improvement is characterized, in part, by storing a tiled sequence of instructions, representing the iterative sequence, in memory. Further, each processor signals its availability to execute a portion of the tiled sequence." The patent is clear that a tile is a "sequence of instructions", i.e. code corresponding to an iterative loop, and the processors are assigned these "tiles" of code.

70. The functional decomposition approach has several disadvantages relative to the domain (data) decomposition approach (which although might not avoid these disadvantages entirely, will nonetheless not experience them with the

same degree of severity). (see also slide 6 from https://www-users.york.ac.uk/~mijp1/teaching/4th_year_HPC/lecture_notes/Parallel_II_Designs.pdf).

71. These disadvantages are:

> (1) There may be dependencies between the different code partitions ("tiles" in the language of Reeve). These dependencies may be in the form of data dependencies in which one code partition (tile) requires data from another partition (tile) as part of its execution task. Alternatively, they may be in the form of semantic dependencies --- code partition (tile) 1 must occur after code partition (tile) as part of the semantic correctness of a task.
>
> (2) Code partitions are generally harder to load balance than data partitions. If each processor is running the same code on different data partitions, it is generally expected that as long as the data partition sizes are the same, each processor will complete its task in approximately the same time. Indeed, this is an implicit assumption throughout the Iyer patent. (See e.g. 3:43-53 "Since the number of elements in each resulting partition differs from the others by at most 1 element, work is divided evenly between the processors for the final merge phase … maximum use of processors is obtained, and hence a near linear improvement in sort time is obtained when adding further processors"). In contrast, if each processor is running a different code partition, it is common that they will complete at different times --- it is well-known that some instructions are more resource-consuming than other instructions and take longer to execute. This is in fact the motivation of the grab strategy cited by the examiner: "It facilitates load-balancing between the processors." 4:6).
>
> (3) If there are more processors than partitions (e.g. n

processors and m partitions and n > m) then (n-m) processors will not be able to contribute to the parallel computation. This limits the scalability of the parallelism. Which this is true for data partitioning as well, since data is generally larger than code, this scalability bottleneck is less likely to be encountered for domain (data) partitioning.

(4) Some code constructs are impossible to partition. The Reeve patent discusses this in column 10. For example, it states: at 10:8-10 "Moreover, some statements are not tilable. These can include a goto out of the loopnest and a subroutine call."

72. The Reeve patent makes innovative contributions to alleviating the first two disadvantages. It does not attempt to alleviate the second two disadvantages.

73. Over time different communities developed around these different approaches. Almost all innovative work regarding functional decomposition approach can be found in the various publications of the HPC community (such as SC and ISC). In contrast, innovative approaches regarding the domain decomposition approach are generally published in that database systems community (such as SIGMOD or VLDB).

74. As the communities have diverged, they have become aware of the progress made in the other community. Members of the database community (such as myself) are expected to stay on top of the papers published in SIGMOD or VLDB, but not SC or ISC. Members of the HPC community are expected to stay on top of the papers published in SC or ISC, but not SIGMOD or VLDB.

75. I can personally testify that while I regularly publish in and/or attend SIGMOD and VLDB, I have never published in or attended an HPC conference in my career. This is true for most people in my community.

76. The Reeve patent is a classic HPC type of innovation. Indeed, this is why they repeatedly refer to Wolfe, Optimizing Supercompilers for Supercomputers (The MIT Press, 1989) and other of Wolfe's HPC publications.

77. A practitioner working in the space of data partitioning (domain decomposition) would generally not look to the HPC community for solutions to problems in the space. The fields of code portioning and data partitioning have diverged too much. Instead, they would look for solutions within the data partitioning / domain decomposition space.

78. To summarize: the '582 Patent is in the field of data parallel batch processing over shared storage, with partitioning defined by file allocation and offsets, and with load balancing achieved through repeated allocation of unallocated file partitions. Reeve is in the field of compiler guided and runtime managed parallelization of loop nests, with partitioning defined over an iteration index space and with scheduling policies for distributing compute tiles to processors or threads. These are distinct fields of endeavor. A POSITA working in the '582 Patent's field would typically look to references addressing parallel file processing, partition manifests, work queue allocation over file ranges, and parallel IO behavior. A

POSITA working in Reeve's field would typically look to compiler directives, runtime libraries for loop scheduling, cache locality, and iteration space decomposition.

79. Because the two references solve different problems using different abstractions, a POSITA would not naturally treat them as belonging to the same field of endeavor. The fact that both references use generic terms such as partition, tile, or processor does not place them in the same field, because the technical meaning and use of those concepts differs. In the '582 Patent, a partition is a well defined range of a file's records described by byte or track offsets and allocated to subtasks without reading file contents during partition definition. In Reeve, a tile is a subset of an iteration space defined by index bounds and tile size and scheduled for execution as part of a loop based computation. (582, 3:36–41; 582, 3:47–53; Reeve, 32:10–16; Reeve, 39:15–17).

80. For these reasons, in my opinion, the '582 Patent and Reeve are directed to different fields of endeavor, and Reeve would not have been a natural reference a POSITA would consult when addressing the specific shared file partition allocation and repeated allocation load balancing techniques described and claimed in the 582 Patent.

VI. GROUND 1: CLAIMS 1-14 ARE NOT OBVIOUS IN VIEW OF IYERS AND REEVE

A. I. Iyer sizes and counts work partitions to match processor count, i.e., no over-subscription and no dynamic "repeat/claim" pool

81. Iyer addresses parallel sorting using multiple processors, and frames the design objective as minimizing overall elapsed time by balancing work across processors. In particular, Iyer states that "the goal is to perform the sort in the shortest period of time," which is achieved "by dividing the work as equally as possible between the multiple processors." (Iyer, 4:21–24.)

82. Iyer does not describe a scheduling scheme in which processors repeatedly request additional unassigned work from a larger pool after completing initial work. Rather, Iyer describes a static equalization approach in which the data is partitioned such that the number of work units is chosen to match the number of processors. For example, Iyer explains that the "first step is to divide the list into multiple equal sized lists," and then expressly states that "[t]he number of such lists should be equal to the number of processors available." (Iyer, 4:25–33.)

83. Iyer then repeats the same one-to-one assignment principle in the next phase of its sorting flow. After the equal-sized lists are formed and sorted, Iyer states: "The second step in the sorting process is to divide the lists into partitions," and "[t]here should be one partition of elements for each available processor." (Iyer, 4:36–40.) This is a deliberate design choice to make the work unit count (lists; then

partitions) coincide with the processor count. That is the opposite of an "over-subscribed" work-queue approach where the system intentionally creates more work units than processors and relies on dynamic allocation to balance heterogeneous or unpredictable execution times.

84. On the contrary, Iyer assumes a homogeneous processor architecture where each processor executes instructions at the same speed as other processors. This is why they divide the input data into "equal sized lists", because if the list size is equal, then the processing time will be equal due to the homogeneous nature of the processors.

85. Iyer's subsequent description of the merge phase further confirms that Iyer's architecture is premised on each processor receiving a predetermined, approximately equal share of the total work, rather than repeatedly acquiring additional work after completion. Specifically, Iyer explains that "each of the P processors merges the lists in its partition," after which the resulting lists are "simply strung together to form the final sorted lists." (Iyer, 3:44–50.) Iyer emphasizes that the approach achieves balance because "each partition is nearly the same size so no processor load imbalance occurs." (Iyer, 2:45–49.)

86. Iyer is thus describing a static partitioning regime in which load imbalance is avoided up front by constructing partitions of approximately equal size and assigning them in a one-to-one manner to processors. (Iyer, 4:25–33; Iyer, 4:36–

40; Iyer, 2:45–49.) This is technically distinct from a dynamic "self-scheduling" or "work stealing" model, where processors repeatedly request and are assigned additional unprocessed work items from a shared pool when they finish earlier than others.

87. Accordingly, Iyer's express teachings are inconsistent with any premise that Iyer contemplates a "more partitions than processors" configuration in which processors complete an initial partition and then thereafter repeat execution on another unprocessed partition by claiming additional work from a pool. Instead, Iyer teaches choosing the number of lists and the number of partitions to match the number of processors, and achieving short completion time by making each processor's assigned partition "nearly the same size" to avoid load imbalance. (Iyer, 4:21–24; Iyer, 4:25–33; Iyer, 4:36–40; Iyer, 2:45–49.)

88. The distinction between Iyer's approach and a dynamic "repeat/claim from a pool" approach can be illustrated with a simple operational analogy. Consider a kitchen with multiple chefs preparing a set of tasks.

89. Under Iyer's approach, the head chef first divides the total prep work into exactly as many plates as there are chefs, with each plate intentionally sized to be about the same effort. Each chef is assigned one plate and completes that plate. When a chef finishes, the chef's assigned work is complete because there are not additional unassigned plates waiting to be claimed. This corresponds to Iyer's express teaching

that, in the first step, "the list [is divided] into multiple equal sized lists," and "[t]he number of such lists should be equal to the number of processors available," followed by "one partition of elements for each available processor." (Iyer, 4:25–33; Iyer, 4:36–40.)

90. By contrast, a dynamic "repeat/claim" scheduling model is like a kitchen that intentionally creates many more work items than chefs, such as a rail of numerous small order tickets. Each chef takes one ticket, completes it, and then returns to the rail to take the next available ticket. Faster chefs naturally complete more tickets, and slower chefs complete fewer, so the kitchen balances workload during execution through repeated claiming of additional unassigned tasks. In computer science terms, that is a self-scheduling or work-queue model characterized by over-subscription and repeated task acquisition.

91. Iyer does not describe the "order ticket rail" model. Instead, Iyer's text aligns with the "one plate per chef" model because it directs that the number of initial lists should match the number of processors, and that there should be one partition for each processor, with the explicit goal of avoiding load imbalance by making partitions "nearly the same size so no processor load imbalance occurs." (Iyer, 2:45–49; Iyer, 4:25–33; Iyer, 4:36–40.) In other words, Iyer's solution is to avoid imbalance up front through static equalization, not to address imbalance by repeatedly allocating

additional unprocessed work items after a processor completes its initial assignment. Such an approach is only practical in homogeneous processor architectures.

92. Accordingly, to the extent an analysis assumes that Iyer operates on a premise of "more partitions than processors" such that processors complete one partition and then thereafter repeat processing on another unprocessed partition by claiming from an unassigned pool, that premise is inconsistent with Iyer's described architecture and stated design objective. (Iyer, 4:21–24; Iyer, 4:25–33; Iyer, 4:36–40.)

B. The Office Action's GRAB-based inference from Reeve does not "carry over" to Iyer without re-architecting Iyer's work-distribution model

93. I have reviewed the Non-Final Office Action in this reexamination proceeding (the "Office Action"), including the Office Action's discussion of the combination of Iyer and Reeve with respect to claim 1. The Office Action expressly acknowledges that "Iyer … does not explicitly teach repeating step (c) … each with another unprocessed partition on a first-come/first-served basis," and then relies on Reeve for that feature. (Office Action, p. 4.) The Office Action further reasons that, because Reeve teaches "more tiles (partitions) than processors," "the execution of step (c) may be repeated," and concludes it would have been obvious to "modify Iyer's processors 1–P to implement Reeve's GRAB strategy to improve efficiency of Iyer's sort method." (Office Action, p. 5.)

94. The Office Action's reasoning assumes a specific predicate: a work-queue or shared pool model in which (i) there are more work units than processors (over-subscription), (ii) some work units remain unprocessed/unallocated while processors are executing, and (iii) processors repeatedly request and acquire additional work units after completing previously assigned work. That predicate is characteristic of dynamic self-scheduling techniques (e.g., work queues, ticket queues, or work stealing), where load balance is achieved during execution by repeatedly allocating "next" work items as processors become available.

95. Iyer does not disclose that predicate. To the contrary, Iyer's approach is explicitly designed to avoid dynamic reallocation by selecting the number and size of work units to match processor resources at the outset. Iyer states that the list is divided into "multiple equal sized lists," and that "[t]he number of such lists should be equal to the number of processors available." (Iyer, 4:25–33.) Iyer likewise states that, when dividing the lists into partitions, "[t]here should be one partition of elements for each available processor." (Iyer, 4:36–40.) Iyer further emphasizes that "each partition is nearly the same size so no processor load imbalance occurs," reflecting a design objective of up-front equalization rather than runtime reallocation. (Iyer, 2:45–49.)

96. Because Iyer's architecture ties the number of lists/partitions to the number of processors, Iyer does not describe a scenario in which processors complete an initial assignment and then "thereafter" obtain "another unprocessed partition"

from a remaining pool. Under Iyer's stated design, there is no remaining pool of unprocessed partitions to be repeatedly claimed; instead, each processor is assigned a partition in a one-to-one manner, and the merge phase proceeds on that predetermined allocation. (Iyer, 3:44–50; Iyer, 4:25–33; Iyer, 4:36–40.)

97. As a result, the Office Action's inference, drawn from Reeve, that "more tiles than processors" implies that execution "may be repeated" via GRAB does not transfer to Iyer without changing Iyer's foundational work-distribution regime. Stated differently, adopting GRAB in Iyer would not be a simple "efficiency" tweak to Iyer's existing parallel sort flow; it would require a redesign with several interdependent changes, including at least the following:

98. a. Intentional over-partitioning. Iyer would need to depart from its express instruction that the number of lists and partitions "should be equal to the number of processors," and instead create more partitions than processors so that a pool of unassigned work exists. (Iyer, 4:25–33; Iyer, 4:36–40; Office Action, p. 5.)

99. b. Shared allocation state. A GRAB-style approach requires maintaining shared state representing which work units remain available, and ensuring that each work unit is claimed by only one processor. In practical implementations, this entails a shared queue, shared marker structure, or equivalent synchronization mechanism to coordinate concurrent requests. That is not part of Iyer's described flow, which is based on a predetermined one-to-one allocation. (Iyer, 4:36–40; Office Action, p. 5.)

100. c. Repeated claim/dispatch loop. Iyer's processors would need to be modified to include a control loop to repeatedly request and receive "another unprocessed partition" after completing work on a prior partition. This repeated acquisition behavior is characteristic of dynamic scheduling, but it is not taught by Iyer's described two-step division followed by per-processor merge. (Iyer, 3:44–50; Iyer, 4:25–33; Iyer, 4:36–40.)

101. d. Reconciliation with Iyer's merge/concatenation assumptions. Iyer's described merge and concatenation steps presume that each processor's partition is predetermined and "nearly the same size" to avoid imbalance. Converting to a dynamic assignment model changes those premises; the system must ensure correctness and ordering under a different allocation regime and may require additional coordination regarding intermediate outputs. (Iyer, 2:45–49; Iyer, 3:44–50.)

102. In my opinion, these changes amount to re-architecting Iyer from a static equalization model into a different class of parallel scheduling system—namely, an over-subscribed, dynamically allocated work-queue model. The necessity of these structural changes is inconsistent with characterizing the modification as a straightforward adjustment "to improve efficiency" while otherwise retaining Iyer's disclosed design. (Office Action, p. 5; Iyer, 4:25–33; Iyer, 4:36–40; Iyer, 2:45–49.)

C. The '582 Patent expressly teaches an over-subscription + repeated allocation model that is absent from Iyer's sorting flow

103. The '582 Patent expressly contemplates a work distribution regime in which there are more work partitions than available processing resources, and in which each worker repeatedly obtains new work after completing prior work.

104. In particular, the '582 Patent explains that "the number of logical input partitions will largely exceed the number of helping systems." ('582, 3:47–50.) This describes an over-subscription approach in which the total work is divided into a number of independent work units that is intentionally greater than the number of participating processors or systems. This design choice is a recognized technique for achieving robust load balance in distributed and/or heterogeneous environments, because it allows faster systems to complete more work units and slower systems to complete fewer.

105. The '582 Patent further states that each subtask "repeatedly tries to allocate for itself and then process, an input partition that has not been allocated yet." ('582, 4:1–4.) This is the defining operational mechanism of a dynamic work-queue model: (i) the system maintains a set of partitions that remain unallocated/unprocessed, and (ii) each worker, upon completing a partition, returns to that set to claim "another" partition until no unallocated partitions remain.

106. The combination of these disclosures—"partitions … largely exceed" the number of helping systems and subtasks that "repeatedly" allocate an "input

partition that has not been allocated yet"—describes the same load-balancing regime reflected in claim 1's "thereafter repeating" limitation: a worker performs its processing step and then thereafter performs the step again with another unprocessed partition, with work being claimed/allocated dynamically as workers become available. ('582, 3:47–50; '582, 4:1–4.)

107. Iyer's sorting embodiment discloses the opposite architectural approach. Iyer instructs that the list is divided into "multiple equal sized lists," and that "[t]he number of such lists should be equal to the number of processors available." (Iyer, 4:25–33.) Iyer likewise states that, after the lists are divided into partitions, "[t]here should be one partition of elements for each available processor." (Iyer, 4:36–40.) Iyer further explains that the purpose of this approach is to ensure that "each partition is nearly the same size so no processor load imbalance occurs." (Iyer, 2:45–49.) As I stated above, Iyer is a static equalization approach—work is balanced in advance by selecting partition counts and sizes to match processor resources—rather than a dynamic repeated-allocation approach.

108. The Office Action acknowledges that "Iyer … does not explicitly teach repeating step (c) … each with another unprocessed partition on a first-come/first-served basis," and then relies on Reeve for the premise that where there are "more tiles (partitions) than processors," execution "may be repeated," and thus suggests modifying Iyer to implement Reeve's GRAB strategy. (Office Action, p. 4; Office

Action, p. 5.) From a technical standpoint, however, the over-subscription predicate that makes such "repeat allocation" operative is expressly taught by the '582 Patent ('582, 3:47–50; '582, 4:1–4) and is expressly not taught by Iyer, which directs that the number of lists/partitions should equal the number of processors to avoid imbalance. (Iyer, 4:25–33; Iyer, 4:36–40; Iyer, 2:45–49.)

109. Accordingly, because Iyer does not disclose the over-subscription condition and corresponding pool of unallocated partitions that the '582 Patent identifies as central to its load-balancing operation, the Office Action's reliance on Reeve's "more tiles than processors" premise does not cure Iyer's acknowledged deficiency without departing from Iyer's described architecture and redesigning Iyer into a different scheduling regime.

D. Iyer's "only two processors are needed" passage concerns partition-boundary determination, not repeated processing of additional unprocessed partitions

110. I have reviewed Iyer's discussion regarding the number of processors used during partition formation, including the passage stating that "it only requires two processors to perform the partitioning when three processors are available to perform the entire sort." (Iyer, 4:60–63.) From a technical perspective, that passage does not describe processors repeatedly performing merge processing, or other step (c) type processing, on a succession of "unprocessed partitions." Rather, it addresses

a different preliminary task: determining partition boundaries, meaning the cut points that define the partitions.

111. Specifically, the cited passage explains that "only two partition boundaries need to be determined," and therefore "it only requires two processors to perform the partitioning when three processors are available to perform the entire sort." (Iyer, 4:60–63.) In parallel sorting systems, "partitioning" in this sense refers to computing the boundary values or indices that delimit each processor's assigned partition. This is the domain decomposition process that I discussed above. Iyer is performing a boundary determination computation step in order to achieve this decomposition. This step precedes the subsequent merge execution. It is not a disclosure that the unused processor later "grabs" additional partitions to merge after completing earlier work.

112. Iyer's subsequent description of the merge phase confirms the one to one, predetermined nature of Iyer's execution model. Iyer states that "each of the P processors merges the lists in its partition," after which the resulting lists are "simply strung together to form the final sorted lists." (Iyer, 3:44–50.) This is a fixed assignment model. After the partitions are established, and their boundaries defined, each processor processes the (singular) partition assigned to it, and the outputs are concatenated. Iyer's merge flow does not describe a mechanism in which a processor

finishes a first partition and thereafter selects another unprocessed partition for further merge processing.

113. Even if one accepts that, in Iyer, the preliminary boundary determination computation may be performed by fewer than all available processors, because only a limited number of boundaries must be computed, that does not imply or disclose the separate runtime scheduling feature at issue. The feature at issue is a system in which processors repeatedly execute the processing step each with another unprocessed partition, for example by claiming work from a pool, on a first come, first served basis. (Office Action, p. 4.) A boundary computation that does not fully occupy all processors is a common preprocessing characteristic in parallel algorithms that use domain decomposition and does not establish the existence of more partitions than processors, a shared pool of unprocessed partitions remaining after initial assignments, or repeated allocation of additional unprocessed partitions to processors after completing prior work.

114. In other words, Iyer's "only two processors" passage describes a limited, preparatory boundary determination task and does not provide a technical basis to infer that Iyer uses a dynamic, repeated allocation scheduling regime for its merge execution or any other processing step. (Iyer, 4:60–63; Iyer, 3:44–50.) On the contrary, it prepares an execution plan in which processor processes exactly one partition.

E. Claim 1 step (a) requires file allocation based partitioning of file records, and neither Iyer nor Reeve teaches that element

115. I have reviewed claim 1 of the '582 Patent, including step (a), which recites "automatically determining file allocation and logically subdividing records of said input file into a plurality of partitions." In my opinion as a computer scientist specializing in parallel data processing, this is not generic "partitioning." The claim language ties the partitioning operation to a preexisting input file, and allocating its records to partitions without physically scanning and rewriting them. That wording reflects an engineering approach in which partitions are defined via offsets of an existing file, rather than by parsing or scanning record contents to discover boundaries.

116. The 582 Patent specification confirms this understanding. It explains that a "logical partition" is a "well defined part of the input," and describes a "very simple way" to define partitions as "consecutive ranges on the input" ranging from one relative byte address to another, or from one relative track address to another. (582, 3:36–41.) The specification then states an explicit constraint: "the logical partition does not rely on actual reading of the file," and "[t]he actual reading of the file is reserved to the subtasks which read the partitions allocated to them." (582, 3:47–53.) This describes a metadata driven partition definition step that can be performed without an O(N) serial pass over the file contents which is particularly slow in the common case where the file is stored on slower storage mediums than RAM memory.

117. In contrast, Iyer cannot perform its partitioning without reading the input data. It is important to note that there are two basic partitioning steps in Iyer. The first partitioning step involves dividing a list into "multiple equal sized lists," with the number of such lists equal to the number of available processors. After these initial partitions are sorted, the next partitioned step invovles dividing the (now sorted) lists into partitions with one partition per processor. (Iyer, 4:25–33; Iyer, 4:36–40.)

118. The first partitioning step requires reading the original list. If it does not read the original list, it will not be able to successfully produce its own stated requirement that each partition has the same number of elements unless it knows a priori where each element begins and the size of each element. Although in some scenarios this possible (e.g. if the input list is an array of fixed sized elements), Iyer does not constrain its algorithm to work only for such limited scenarios.

119. It should also be noted that the input for this first partitioning step does not appear to be data in a file, but rather data in an in-memory list data structure. The very process of creating this data structure in memory would require a scan of the raw data on storage.

120. The second partitioning step explicitly requires reading the output from the first step (after each list is sorted). The code presented in figures 2F and 2G clearly show that this output is read to carefully specify the partition boundaries such that each processor will have an equal amount of work in this second step.

121. In short, Iyer does not disclose "automatically determining file allocation", and it does not teach the 582 Patent's constraint that partitions are defined without reading the file contents prior to subtask execution. On the contrary, if the algorithm in Iyer was attempted to be run without first reading the data contents, it would fail at it primary load balancing goal.

122. Reeve is even further removed from claim 1 step (a). Reeve's "partitioning" is iteration space tiling, defined by indices and bounds via directives such as "AFFINITY REGION (<index and bounds>)" with "tilesize." (Reeve, 32:10–16.) This is task decomposition over an iteration space, not "automatically determining file allocation" and subdividing the records of a preexisting input file into file partitions. Its tiling decisions are not driven by file allocation metadata and are not directed to defining file range partitions of a shared input file. Thus, even if Reeve uses the vocabulary of "tiles" or "partitions," it does not teach claim 1 step (a)'s file allocation based partitioning of file records, and it does not teach the 582 Patent's explicit constraint that this partitioning is performed without reading the file contents. (Reeve, 32:10–16; Reeve, 39:15–17.)

123. Accordingly, in my opinion, the combination of Iyer and Reeve does not teach claim 1 step (a) as written and as explained by the 582 Patent. The Office Action's reliance on generic "partitioning" concepts from Iyer, which concerns list segmentation for sorting, and from Reeve, which concerns iteration space tiling, does

not satisfy the requirement of "automatically determining file allocation" and logically subdividing records of an input file into partitions in a manner consistent with the specification's constraint that partitions are defined without reading the file contents prior to subtask processing.

F. A POSITA would not have been motivated to combine Iyer and Reeve to arrive at the claimed system

124. I have reviewed Iyer and Reeve from the perspective of a person of ordinary skill in the art in the relevant timeframe, with training and experience in parallel processing and high performance computing. In my opinion, a POSITA would not have been motivated to combine Iyer's parallel sorting and merging technique with Reeve's iteration space tiling and GRAB scheduling in the manner suggested, because the two references address different computational problems, operate on different decomposition targets, and embody materially different execution models.

125. Iyer is directed to parallelizing a sorting workflow by dividing a list into equal sized lists and then dividing the lists into partitions, with the express objective of dividing work "as equally as possible" and selecting the number of lists and partitions to match the number of processors so that load imbalance is avoided by construction. (Iyer, 4:21–24; Iyer, 4:25–33; Iyer, 4:36–40; Iyer, 2:45–49.) To reiterate

what I stated above, Iyer is a static workload equalization design. It anticipates a fixed, one to one mapping of work partitions to processors, followed by a merge phase where each processor performs the merge for its assigned partition and the results are concatenated.

126. Reeve, by contrast, is directed to decomposing an iteration space for compute kernels into tiles using compiler directives and runtime scheduling policies, including a GRAB approach in which requesting processors are assigned tiles on a first come, first served basis. (Reeve, 32:10–16.) Reeve's scheduling is designed for loop based computation over an index space and is motivated by issues such as locality, granularity control, and distributing iteration space work to threads.

127. In short, Iyer partitions data (along the lines of the domain decomposition philosophy I discussed above) while Reeve partitions computation (along the lines of the functional decomposition philosophy I discussed above). These two philosophies are considered as alternatives to each other. If for some reason data composition was not working, one would switch to functional decomposition, not attempt to combine them.

128. Furthermore, a POSITA seeking to implement or improve Iyer's sorting flow would not look to Reeve for the specific feature the Office Action attributes to the combination, namely repeated processing of "another unprocessed partition" after completion of prior work. The reason is that Iyer's core design objective is to eliminate

the need for dynamic rescheduling by selecting the number and size of partitions to avoid processor load imbalance. (Iyer, 2:45–49; Iyer, 4:25–33; Iyer, 4:36–40.) Introducing Reeve's GRAB mechanism would undermine Iyer's express design choice and would require deliberately departing from Iyer's one to one partitioning scheme, including creating more work units than processors and introducing shared state to manage repeated allocation. A POSITA would recognize that this is not a routine "efficiency improvement" within Iyer's framework, but instead a shift to a different class of scheduling regime, with different correctness and coordination considerations.

129. Furthermore, Iyer assumes a homogeneous architecture where each processor runs at the same speed (which is why load is balanced if each processor gets lists of the same size). Introducing Reeve's GRAB mechanism and shared pool and scheduling mechanisms would actually hurt performance in that context. When workload is predictable, it is better to avoid paying the overhead of dynamic scheduling complexity. Instead, it is far better to schedule statically in advance, as proposed by Iyer. Adding the GRAB mechanism would consume CPU cycles to context switch between different partitions on the same processor, to coordinate with other processors to ensure that no partition is processed twice, and track scheduling state and progress status.

130. Moreover, the type of "work unit" in Reeve is not aligned with the type of "work unit" in Iyer. Reeve's tiles are portions of an iteration space, defined by index bounds and tilesize directives, and are executed as computational tiles within a loop nest. (Reeve, 32:10–16.) Iyer's partitions are conceptual partitions lists for sorting, merging, and concatenation. (Iyer, 3:44–50; Iyer, 4:36–40.) A POSITA would not view Reeve's GRAB scheduling of iteration space tiles as a direct solution to a perceived deficiency in Iyer's merge partitioning, particularly where Iyer expressly teaches that its partitions are "nearly the same size" so that "no processor load imbalance occurs." (Iyer, 2:45–49.) In other words, Reeve addresses a circumstance where dynamic scheduling is beneficial, whereas Iyer is engineered to avoid that circumstance.

131. For these reasons, in my opinion, a POSITA would not have been motivated to combine Iyer and Reeve in the manner suggested to achieve a repeated allocation model. The combination would require departing from Iyer's stated partition sizing and assignment approach, repurposing Reeve's iteration space scheduling concepts into a different computational pipeline, and adding shared allocation mechanisms that Iyer neither needs nor contemplates. (Iyer, 2:45–49; Iyer, 4:25–33; Iyer, 4:36–40; Reeve, 32:10–16; Reeve, 39:15–17.)

132. I have reviewed the Office Action's rationale for combining Iyer with Reeve, including the statement that it would have been obvious to "modify Iyer's

processors 1 to P to implement Reeve's GRAB strategy to improve efficiency of Iyer's sort method." (Office Action, p. 5.) From an engineering perspective, once one departs from Iyer's disclosed one to one partitioning model and restructures Iyer into a pool based repeated claiming regime, additional technical incompatibilities and design constraints arise that are not addressed in the Office Action.

133. First, a pool based repeated claiming regime requires that processors have access to a representation of the available work units so that they can repeatedly obtain another unprocessed partition after completing prior work. In practice, this requires generating and distributing "descriptions of all partitions" in a form that can be queried or claimed by processors, and it requires coordination so that each partition is claimed once. The Office Action does not explain how such partition descriptions and coordination would be implemented in Iyer, which instead teaches dividing the list into equal sized lists and selecting the number of lists and partitions to match the number of processors. (Iyer, 4:25–33; Iyer, 4:36–40.) In other words, Iyer's disclosed flow does not present a shared work description structure because the one to one assignment makes such a structure unnecessary.

134. Second, converting Iyer's merge execution into a regime where processors sequentially execute multiple partitions introduces correctness questions for Iyer's merge and concatenation logic that are not addressed by the Office Action. Iyer describes a merge phase in which "each of the P processors merges the lists in its

partition," followed by the merged lists being "simply strung together to form the final sorted lists." (Iyer, 3:44–50.) That description presumes that each processor has a predetermined partition and that the resulting merged outputs can be concatenated in a known order to produce a correct globally sorted output. If processors instead execute multiple partitions sequentially by repeatedly claiming additional partitions, the system must specify how those outputs are to be organized and concatenated to preserve correctness, including maintaining the proper ordering relationships among merged partitions. The Office Action does not identify where Iyer or Reeve teaches these integration details, and it does not explain how the correctness invariants (i.e. that the output be globally sorted) of Iyer's merge and concatenation would be preserved under repeated dynamic assignment.

135. Third, introducing over subscription and dynamic claiming is in tension with Iyer's stated design objective of dividing work "as equally as possible" to complete sorting in the shortest time. (Iyer, 4:21–24.) Iyer's approach achieves that objective by selecting the number of lists and partitions to match the number of processors and by making each partition nearly the same size so that "no processor load imbalance occurs." (Iyer, 2:45–49; Iyer, 4:25–33; Iyer, 4:36–40.) If the system intentionally creates more partitions than processors and relies on dynamic claiming, the design goal shifts from static equalization to runtime balancing. A POSITA would recognize that this shift changes the tradeoffs. Dynamic claiming introduces overhead

such as synchronization, coordination, duplicating start-up costs, and possibly increased merge complexity, and it no longer preserves Iyer's stated premise that balance is achieved by constructing partitions that are nearly the same size. The Office Action does not address these tradeoffs or explain why a POSITA would expect the redesigned system to maintain Iyer's stated equal division goal.

136. In my opinion, these unaddressed issues illustrate that the proposed modification is not a straightforward optimization of Iyer. Instead, it is a substantial redesign that raises multiple engineering questions concerning work description distribution, allocation coordination, and preservation of correctness and performance objectives. The Office Action's conclusory statement that the modification would "improve efficiency" does not address these incompatibilities or provide a technically grounded explanation for how the redesigned system would operate consistently with Iyer's disclosed sorting and merge flow. (Office Action, p. 5; Iyer, 4:21–24; Iyer, 4:25–33; Iyer, 4:36–40.) Furthermore, the modification would likely decrease efficiency instead of improving it in the context of Iyer's homogeneous processor architecture.

G. Dependent claims 2, 5,6, 7, 9, 10, 11 and 12 recite additional, narrowing technical constraints that are not taught or suggested by Iyer in view of Reeve

137. I have reviewed dependent claims 5, 10, and 11 of the 582 Patent, as well as the Office Action's mappings for those claims. In my opinion as a computer

scientist specializing in high performance computing, each of claims 5, 10, and 11 introduces an additional technical limitation that further constrains the claimed run time partition based processing architecture of claim 1. Even assuming arguendo that claim 1 were taught or suggested by the cited references, I do not believe the Office Action identifies teachings in Iyer and Reeve that satisfy these additional limitations.

(a) Claim 5. "An accumulation of output records from said subtasks in an arbitrary order" is an output ordering requirement, not a statement about asynchronous completion

138. Claim 5 requires that "the output in step (e) is an accumulation of output records from said subtasks in an arbitrary order." From a technical perspective, this limitation concerns the ordering characteristics of the output records that are produced or assembled as part of step (e). It is not satisfied merely because parallel work may complete at different times. Parallel tasks can complete asynchronously while still producing an output that is assembled in a defined order, for example by collecting partial outputs and concatenating them according to a specified order, or by performing an order preserving merge.

139. The Office Action asserts that claim 5 is taught because Iyer includes a "waiting" step which purportedly confirms that processors complete their subtasks independently and at varying times, and the Office Action equates that to "arbitrary order." (Office Action, p. 6, citing Iyer, 11:23–30.) In my opinion, that reasoning does

not establish claim 5's output ordering requirement for at least four independent technical reasons.

140. First, completion time is not output order. A parallel algorithm can have processors complete intermediate tasks at different times, and yet still produce an output that is assembled according to a defined ordering rule. Thus, an inference that processors finish at "varying times" does not disclose that the output itself is an "accumulation of output records" in an "arbitrary order." (Office Action, p. 6; Iyer, 11:23–30.)

141. Second, the type of "waiting" referenced in parallel computing is typically a synchronization mechanism used to ensure correctness, such as a barrier or dependency wait that prevents later operations from proceeding until required prior operations complete. As an engineering matter, such waiting reflects structured coordination and ordering constraints. It is not an instruction that output records are accumulated without order. (Office Action, p. 6; Iyer, 11:23–30.)

142. Third, in the context of Iyer at the point that the Office Action cites, the waiting is precisely for a preceding processor. It is giving additional information about the process described earlier in the patent at 4:62-5:1: "The first processor determining the first partition boundary will then pass the boundary definition onto the second processor determining the second boundary. This will clearly define the second partition to the second processor when the second processor has determined the

second boundary. The second processor further passes on the second boundary definition to the third processor to define the third partition." The patent thus states at point that was cited in the Office Action: "The processors will wait for the partition boundary information from the preceding processor." In other words, the point of the wait in the context of Iyer is precisely to add order across processors instead of remove it!

143. Fourth, the Office Action's inference cannot be reconciled with Iyer's own description of forming the final sorted output. Iyer's sorting embodiment is expressly directed to producing globally sorted results, and it describes a defined output formation procedure: "each of the P processors merges the lists in its partition to form P sorted lists" and then "the lists are then simply strung together to form the final sorted lists." (Iyer, 3:45–49.) That "stringing together" to form the "final sorted lists" presupposes a prescribed concatenation order that preserves global sortedness. This is inconsistent with an "accumulation" of output records "in an arbitrary order" as claim 5 requires. (Iyer, 3:45–49.)

144. Accordingly, even if Iyer discloses waiting and asynchronous completion of intermediate tasks, that does not establish that step (e) produces an accumulation of output records in an arbitrary order. In my opinion, the Office Action has not identified a teaching in Iyer or the proposed Iyer plus Reeve combination that

satisfies claim 5's narrowing output ordering limitation. (Office Action, p. 6; Iyer, 11:23–30; Iyer, 3:45–49.)

(b) Claims 6 and 7. A disk bus connects a processor to a local disk, not to a storage area network or network attached storage.

145. Claim 6 recites that "method defined in claim 1 wherein said input file resides on a storage area network and is derived therefrom." and claim 7 recites that "method defined in claim 1 wherein said input file resides on a network-attached storage and is derived therefrom". The Office Action maps these claims to Figure 1, elements 26 and 28 and the description of this figure at 4:7-13. However, these citations are discussing accessing one or more local disks over an I/O Bus. The term I/O Bus is a term of the art that describes accessing data on a local disk (as shown in that same figure). It is not used in the context of accessing data in a storage are network or network attached storage.

(c) Claims 9 and 10. A post run cleanup routine that emits statistics does not teach that the computer executable process is a statistical analysis or report creating process

146. Claim 9 recites that "the method defined in claim 1 wherein said computer executable process is a statistical analysis process." and claim 10 recites that "the method defined in claim 1 wherein said computer executable process is a report creating process." Technically, these limitations concern the nature of the process that

is being executed under the claim 1 framework, meaning the process that is partitioned into subtasks and whose outputs are combined. A POSITA would understand a "statistical analysis process" and "report creating process" to be a process whose substantive purpose and output is the creation of a statistical analysis or report, not merely a process that logs execution metrics or emits diagnostic information about the run.

147. The Office Action maps claims 9 and 10 to Reeve's statement that "pr_end is executed once when the program finished. In addition to general cleanup, it produces statistics and reports." (Office Action, p. 6, citing Reeve, 30:13–14.) In my opinion, this does not teach claims 9 and 10. The cited passage describes a termination routine executed once after the program completes its substantive work, which performs cleanup and produces statistics and reports about the execution. These are reports about program operation and performance, not the substantive report output that defines the purpose of a statistical analysis process or report creating process.

148. In HPC and distributed computing systems, it is routine for programs to emit end of run statistics, logs, or summaries, including counters, timing metrics, and diagnostic output. Treating such routine telemetry as sufficient to satisfy a "statistical analysis process" or "report creating process" limitation would effectively eliminate the narrowing effect of claims 9 and 10, because virtually any program that logs

runtime statistics could be characterized as producing "statistics" or "reports." Claims 9 and 10 instead require that the underlying computer executable process itself is a statistical analysis or report creating process, meaning the process is directed to producing a statistical analysis or report as its substantive output. Reeve's pr_end disclosure does not teach that. (Office Action, p. 6; Reeve, 30:13–14.)

149. Accordingly, in my opinion, the Office Action has not identified a teaching in Reeve, or the proposed Iyer plus Reeve combination, that the computer executable process executed under claim 1 is itself a statistical analysis or report creating process, as opposed to merely including a post execution cleanup routine that emits diagnostic statistics and reports. (Office Action, p. 6; Reeve, 30:13–14.)

(d) Claim 11. Generic disk storage or bus architecture is not a disclosure of a database query

150. Claim 11 requires that the computer executable process includes a database query. From a technical perspective, a "database query" connotes execution of a query operation against a database system (such as Oracle, PostgreSQL, MySQL, SQLServer, Snowflake, etc), for example issuing a query statement or predicate, evaluating selection criteria, and retrieving records or fields in response to the query. It is not disclosed merely by describing that data resides on disk or that processors can access storage through an input output bus.

151. The Office Action asserts that Iyer teaches claim 11 and cites Iyer, 4:11–16. (Office Action, p. 7.) The cited passage in Iyer describes a multiprocessor system architecture and storage connectivity, including main storage access through an input output bus and data storage in direct access storage devices. (Iyer, 4:11–16.) In my opinion, that disclosure does not teach a database query. It does not describe querying a database, issuing a query, applying query predicates, interacting with a database management system, or retrieving records in response to a database query operation. It describes hardware and storage access.

152. Equating the existence of disk resident data or storage devices with a "database query" would read the words "database" and "query" out of claim 11. Almost any computer system includes disk storage and data access, but that does not mean the process includes a database query. The Office Action does not identify where Iyer teaches the specific query operation required by claim 11, and Iyer's focus is parallel sorting and merging rather than database retrieval. (Office Action, p. 7; Iyer, 4:11–16.)

153. Accordingly, in my opinion, the Office Action's citation to Iyer, 4:11–16 is insufficient to establish claim 11's database query limitation, and the proposed Iyer plus Reeve combination does not teach or suggest that narrowing limitation. (Office Action, p. 7; Iyer, 4:11–16.)

154. For these reasons, dependent claims 5, 10, and 11 are not taught or suggested by the proposed Iyer plus Reeve combination, for the independent additional limitations discussed above, in addition to any patentability of claim 1 from which they depend.

(e) Claims 2 and 12. Processor 1 in Iyer does both partitioning and subtask processing. Placing it on a separate mainframe would violate Iyer's homogeneous processor assumption.

155. Claim 2 requires: "the automatic determination of file allocation and logical subdivision of records of said input file into said plurality of partitions in step (a) and the distribution of the description of all of said partitions in step (b) is carried out with at least one further processor in addition to the Subtask processors." It requires that at least one processor performs the action of partitioning and distributing the partitions that is distinct from the processors doing the subtask processing. Claim 12 takes this a step further, requiring this processor to be running on a separate mainframe computer, The office action cites to Figure 3A and 3B along with their descriptions at 10:66-11:40 in Iyer and states that processor 1 in the figure "corresponds to the claimed at least one further processor". With respect to claim 12 it states that "it would be obvious to place processor 1 on its own mainframe computer".

156. Figures 3A and 3B in Iyer do not show a distinct separation of duties as required by the claim. This is because the processors in Iyer perform both partitioning

and sorting/merging. Iyer states at 4:62-5:1: "The first processor determining the first partition boundary will then pass the boundary definition onto the second processor determining the second boundary. This will clearly define the second partition to the second processor when the second processor has determined the second boundary. The second processor further passes on the second boundary definition to the third processor to define the third partition." In other words, the entire algorithm described in Iyer involves a collaboration across processors to perform the partitioning. These same processors then perform the rest of the sort algorithm. There is no separation between the partition creation process and data processing process such that they can be performed on different machines. On the contrary, Figure 3a and 3b in Iyer show that same processor 1 performing both the partition creation ("find 1st partition boundary" and data processing ("sort elements 1 through N/P").

157. Furthermore, placing processor 1 on a mainframe would violate Iyer's homogeneous processor architecture assumption. Since the mainframe processor runs at a different speed as the other processors, the assumption that giving each processor the same amount of data to process will lead to balanced parallelism would no longer be true.

158. Placing processor 1 on a mainframe would also turn Iyer into a distributed system and thus require all the radical architecture changes necessary when moving from a single multi-processor system to a distributed system.

VII. GROUND 2: CLAIMS 1-14 ARE NOT OBVIOUS IN VIEW OF REEVE

A. Different object being partitioned. Reeve partitions an iteration space into tiles, not a preexisting file's records into file range partitions based on file allocation

159. As I described above, the '582 patent uses a domain decomposition philosophy for parallelizing tasks, and Claim 1 is anchored specifically to domain decomposition and requires partitioning work based on the input data to be processed. The input data to be processed is located inside a file in a file system. Claim 1 requires "automatically determining file allocation and logically subdividing records of said input file into a plurality of partitions." The 582 Patent specification describes that partitioning step as a file defined operation that uses operating system information about the size and location of the input file to create a control file specifying logical partitions, where a logical partition is a well defined part of the input. (582, 3:30–33; 582, 3:35–36.) The patent further provides a concrete definition of such partitions as consecutive ranges on the input defined from one relative byte address to another, or from one relative track address to another. (582, 3:37–41.) In other words, the claimed partitions are file range segments derived from file allocation information and offsets, with attention to record boundaries.

160. Reeve, by contrast, uses a functional decomposition approach (see Section V above where I explain the fundamental differences between domain

decomposition and functional decomposition and refer to reading materials where these alternative parallel processing approaches are explained in depth). While domain decomposition partitions work based on the input data, functional decomposition partitions work based on the code that processes this data. Specially, Reeve's "tiles" are compute domain subregions of an iteration space. Reeve's directive syntax makes this explicit by defining an affinity region using an index and bounds together with a tilesize and a selectable strategy such as GRAB, MOD, WAVE, or SLICE. (Reeve, 32:10–16.) In technical terms, those directives define bounded subregions of an iteration space for executing a parallel construct. They do not determine file allocation, and they do not logically subdivide the records of a preexisting input file into partitions defined by file offsets or track boundaries.

161. Tiles in Reeve do not execute the same code on different input data partitions. Rather, one tile can process the output from a previous tile. This is why the patent spends so much time discussing data dependencies between tiles. For example, when defining the notion of a data dependency, Reeve states at 2:53-55: "an instruction in the first tile writes a selected datum, which an instruction in the second tile subsequently reads." Such a dependency would not emerge from the type of partitioning discussed in the language of claim 1 in the '582 patent, since claim 1 only discusses partitioning a single input file, not a combination of input data and output data from previous steps of data processing.

162. This is not a semantic distinction. It goes to the mechanism and the object being partitioned. Claim 1 requires partitions of a preexisting file's records based on file allocation information. Reeve describes tiling of an iteration space based on indices, bounds, and a tile size for scheduling computation. (Reeve, 32:10–16; 582, 3:30–33; 582, 3:37–41.) A POSITA would understand these as different kinds of partitioning with different inputs, different artifacts, and different runtime consequences.

B. Different partitioning target. The 582 Patent divides data, while Reeve divides and schedules computation

163. The 582 Patent partitions the data set. The unit of parallelism is a slice of the input data itself, namely a portion of a preexisting input file. The specification explains that the split step uses operating system information about the size and location of the input file to create a control file specifying logical partitions, and it defines those partitions as consecutive file ranges specified by relative byte addresses or relative track addresses. (582, 3:30–33; 582, 3:37–41.) Each subtask then reads and processes its allocated data partition.

164. Reeve partitions the computation. It decomposes the execution of a parallel construct into tiles of an iteration space and schedules those tiles to threads during program execution. Reeve's syntax describes an affinity region defined by an index and bounds with a tilesize and a strategy such as GRAB. (Reeve, 32:10–16.)

Reeve also describes a runtime library influencing execution by choosing the number of threads for a parallel construct, including more or less threads than available processors, and it relies on the operating system scheduler for load balancing. (Reeve, 31:7–13.) This reinforces that Reeve is concerned with scheduling computation units during execution of a parallel program construct, not with subdividing a preexisting input file into file range partitions and ensuring all data partitions are processed.

165. This is also evident in the fact that in Reeve one tile can process the output from a previous tile which can cause dependencies across tiles. For example, when defining the notion of a data dependency, Reeve states at 2:53-55: "an instruction in the first tile writes a selected datum, which an instruction in the second tile subsequently reads." In other words, there are two sequential steps that need to happen: first a particular data item needs to be written, and then a separate computationally distinct step happens in which it gets read. As another example of a dependency, Reeve states at 2:57-59: "an instruction in the first tile reads a selected datum, which an instruction in the second tile that subsequently writes". Again, there are two sequential steps that need to happen: first a particular data item needs to be read, and then a separate computationally distinct step happens in which it gets written. Reeve places these in two separate tiles that will get scheduled separately. Such a dependency would not emerge from the type of partitioning discussed in the language of claim 1 in the '582 patent, since each subtask processes a separate

partition of the input data. Any output from data processing in claim 1 is not produced until step (e) of claim 1. If further processing of this output is necessary, this would be done with another iteration of all the steps of claim 1 or the merge step discussed in later claims of the patent.

166. This distinction matters. Claim 1's architecture assumes that processing is performed on partitions of the input file's data and that completion is achieved by covering all unprocessed data partitions, with the 582 Patent expressly contemplating that the number of logical input partitions exceeds the number of helping systems to enable dynamic allocation. (582, 3:47–50; 582, 4:1–4.) Reeve's tiling is a mechanism for decomposing and scheduling computational iterations among threads, which can include sequential steps of computation, not a mechanism for defining and allocating file data partitions until all file data partitions have been processed. (Reeve, 31:7–13; Reeve, 32:10–16.)

C. Different meaning of first come first served. Reeve's FCFS assigns iteration space tiles to threads, while claim 1 uses FCFS for repeated allocation of unallocated file partitions after completion

167. The Office Action relies on Reeve's GRAB strategy to satisfy claim 1's requirement that subtask processors thereafter repeat processing each with another unprocessed partition on a first come first served basis. (Office Action, pp. 7–8.) In my opinion, the phrase first come first served is being used in materially different ways in Reeve and in the 582 Patent.

168. Reeve's FCFS statement appears in its description of the GRAB strategy, stating that tiles are assigned to threads with a pure first come first served algorithm. (Reeve, 39:4–6.) In context, the work items are tiles of an iteration space and the workers are threads executing a tiled parallel construct. Reeve's runtime discussion similarly frames execution around allocating threads to a parallel construct and relying on the operating system scheduler for balancing. (Reeve, 31:7–13.) Thus, Reeve's FCFS is a policy for assigning pre defined compute tiles to threads within a runtime environment.

169. Claim 1's FCFS limitation, as used in the 582 Patent, is an application level allocation mechanism over a pool of unallocated data partitions. The 582 Patent teaches that the number of logical input partitions will largely exceed the number of helping systems and that each subtask repeatedly tries to allocate for itself and then process an input partition that has not been allocated yet. (582, 3:47–50; 582, 4:1–4.) This describes a system that tracks allocation status, supports claiming of unallocated file partitions, and repeatedly allocates another unprocessed partition after prior processing completes. In that context, first come first served is the allocation policy governing which subtask receives the next unallocated data partition when it requests additional work. (582, 4:1–4.)

170. Reeve does not disclose the predicate conditions that make claim 1's FCFS repetition meaningful in the 582 Patent. The 582 Patent's FCFS repetition

depends on partitions outnumbering helping systems, a pool of partitions that have not been allocated yet, and repeated allocation of an unallocated data partition to a subtask after completion of previous processing. (582, 3:47–50; 582, 4:1–4.) Reeve's FCFS tile assignment to threads within an iteration space tiling framework does not teach or suggest a pool of unallocated file data partitions that are repeatedly allocated to independent subtasks operating on partitions of a preexisting input file. (Reeve, 31:7–13; Reeve, 39:4–6.)

171. For these reasons, the mere presence of the words first come first served in Reeve does not establish that Reeve teaches claim 1's "thereafter repeating" limitation in the file partition sense used by the 582 Patent.

D. Reeve does not disclose the 582 Patent's output model, which combines discrete subtask outputs into a result output file or files

172. Claim 1 requires generating an output combining all of the subtask outputs. In the 582 Patent, this limitation has a concrete file processing meaning. Each subtask operates on an allocated partition of a preexisting input file and produces its own output, and those outputs are then further processed to produce an output equivalent to that expected from the original process. The 582 Patent explains that the output of the various subtasks has to be further processed to create an equivalent output and, for that reason, each subtask creates its output on dedicated output files. (582, 4:5–9.) The patent further describes a subsequent step that is scheduled after

subtasks complete and whose function is to combine or merge the outputs of the various subtasks into the result output file or files. (582, 3:14–20; 582, 4:57–60.)

173. Reeve, by contrast, is directed to runtime scheduling of tiles to threads during execution of a parallel construct. In that setting, the results of executing tiles are typically manifested as updates to program state or data structures associated with the parallel construct, not as separate per thread output files that must be combined to produce a result output file. Reeve's description of GRAB states that tiles are assigned to threads with a first come first served algorithm, but it does not disclose the 582 Patent's discrete subtask output and combination into a result output file model. (Reeve, 39:4–6; 582, 4:5–9; 582, 4:57–60.)

174. Accordingly, in my opinion, Reeve does not teach claim 1's file partition based architecture, including the step (a) file allocation driven partitioning of a preexisting input file, the repeated allocation of unallocated file partitions among subtasks, and the generation of an output by combining discrete subtask outputs into a result output file or files. The Office Action's mapping relies on treating Reeve's iteration space tiling and tile to thread scheduling as if it were equivalent to the 582 Patent's file partition job processing framework, which is not how a POSITA would understand these disclosures when claim 1 is interpreted in context.

E. The Office Action's Reeve alone mapping is structurally flawed because it substitutes generic parallelism for the specific file partition load balancing architecture of claim 1

175. I have reviewed the Office Action's rejection of claim 1 based on Reeve alone, including the assertion that Reeve discloses the claim 1 architecture. (Office Action, pp. 7–8.) In my opinion as a computer scientist specializing in parallel data processing, the Reeve alone mapping is structurally flawed because it treats Reeve as if it were directed to the same type of partitioning and load balancing described as claimed in the 582 Patent. Claim 1, read in context under the Phillips framework, is not directed to parallel execution in the abstract. It recites a specific architecture for accelerating a linear file processing job by partitioning a preexisting input file into file range partitions, dynamically allocating those partitions among workers, and producing a combined output equivalent to the output of the original process. The 582 Patent describes this architecture in concrete file centric terms that do not appear in Reeve.

176. First, claim 1 is anchored to data partitioning of a preexisting input file and to the mechanism by which those partitions are defined. The 582 Patent explains that the split step uses operating system information about the size and location of the input file to create a control file specifying a logical partition of the input. (582, 3:30–33.) It further explains that a logical partition is a well defined part of the input and describes a very simple way to define partitions as consecutive ranges on the input,

such as from one relative byte address to another or from one relative track address to another. (582, 3:35–41.) In other words, the unit of parallelism is a slice of an existing data object, the input file, and the partition definition is tied to file allocation and file offsets.

177. Second, claim 1 requires load balancing by repeated allocation of unallocated file partitions. The 582 Patent expressly contemplates over subscription of partitions relative to helping systems, stating that the number of logical input partitions will largely exceed the number of helping systems. (582, 3:47–50.) It then describes the operative mechanism, stating that each sub task repeatedly tries to allocate for itself and then process an input partition that has not been allocated yet. (582, 4:1–4.) This is an application level allocation scheme over a pool of partitions whose allocation status is tracked, and it requires repeated claiming of additional unallocated partitions until all partitions have been processed. It is not merely that multiple units exist or that processors may execute in parallel. It is a specific dynamic allocation protocol built around unallocated file data partitions. (582, 3:47–50; 582, 4:1–4.)

178. Third, claim 1's output limitation is grounded in the same file partition pipeline. The 582 Patent explains that the outputs of the sub tasks have to be further processed to create an output that is equivalent to the one expected from the original process, and it describes that each sub task can create output on dedicated output files,

followed by a step that combines the outputs of the various sub tasks into the result output file or files. (582, 4:5–9; 582, 4:57–60.) Thus, claim 1's output combining all subtask outputs corresponds to a specific workflow involving partition level outputs and combining or merging those outputs into a final result that is equivalent to the output of the original non partitioned process.

179. Reeve is structurally different. Reeve describes a compiler and runtime framework for executing parallel constructs using threads, where a runtime library influences execution by choosing the number of threads and where thread scheduling is largely delegated to the operating system scheduler. (Reeve, 31:7–13.) Reeve's tiling is defined through directives specifying an affinity region using an index and bounds together with a tilesize and a strategy such as GRAB, MOD, WAVE, or SLICE. (Reeve, 32:10–16.) The GRAB strategy then assigns tiles to threads using a first come first served algorithm. (Reeve, 39:4–6.) These disclosures describe decomposition and scheduling of computation over an iteration space defined by indices and bounds. They do not describe partitioning a preexisting input file into file range partitions derived from file allocation metadata, they do not describe maintaining a pool of unallocated file data partitions that are repeatedly allocated to subtasks on helping systems, and they do not describe producing discrete subtask output files that must be combined to yield a final output equivalent to an original file processing job. (Reeve, 31:7–13; Reeve, 32:10–16; Reeve, 39:4–6.)

180. The structural difference between the 582 Patent's claimed architecture and Reeve's disclosure can be illustrated with a simple analogy. Consider a scenario where multiple people are asked to complete work (such as to make French toast out of loaf of bread) faster by using parallel effort.

181. The 582 Patent is analogous to starting with a preexisting physical object, such as a loaf of bread that already exists on the counter, and accelerating the task by slicing that loaf into many slices based on high level information about the loaf (i.e. it is a foot long). A coordinator can prepare a list of slices without eating or inspecting the bread, but only using the knowledge that it was originally a foot long. For example, it can specify that it will be divided into 24 half-inch slides. Multiple helpers then repeatedly take the next unclaimed slice from the 24 slice pool, , further cut it into smaller pieces, dip each piece into french toast egg batter, fry the pieces, and place the finished pieces into their own trays . When all slices have been processed, the finished pieces from all trays are combined into one basket that is equivalent to the result that would have been produced if one person had made the French toast out of the entire loaf alone. This reflects the 582 Patent's file centric architecture, in which a preexisting input file is subdivided into file range partitions, those partitions are allocated to subtasks from a pool of unallocated partitions, and the outputs of the subtasks are later combined to form an output equivalent to the original process.

182. Reeve is analogous to a different type of parallelization. Instead of starting with an already existing loaf and dividing it into slices, one starts with a recipe and divides the cooking procedure into steps, such as "cut slices into pieces", "dip pieces into French toast egg batter," "fry the pieces", and "make the result look nice with powdered sugar and chocolate drizzle". The steps have dependencies (e.g. a particular piece can only be fried after it has been dipped into batter), but can still be parallelized in a pipeline fashion --- e.g. one person dips the next piece while the other person fries the previous piece. In short, this model partitions the procedure or task, not a preexisting physical object. This is analogous to Reeve's disclosure, which partitions an iteration space into tiles defined by index and bounds and tile size directives and schedules those tiles to threads under a strategy such as GRAB.

183. The analogy illustrates why the Office Action's Reeve alone mapping is structurally flawed. Claim 1 requires dividing an existing data object, the preexisting input file, into partitions tied to file allocation and offsets, dynamically allocating those partitions among workers, and then combining subtask outputs into a final output equivalent to the original process. Reeve instead describes dividing a computation into tiles of an iteration space and scheduling those tiles to threads during execution of a parallel construct. The fact that both approaches can be described as "parallel" does not make them equivalent. They operate on different objects, use

different partition definition mechanisms, and produce results through different execution and output models.

184. Accordingly, in my opinion, the Office Action's Reeve alone mapping fails at the architectural level because it substitutes generic concepts of parallelism and work scheduling for the specific file partition allocation and output combination architecture required by claim 1 and described in the 582 Patent. Treating Reeve's tile to thread scheduling for iteration space computation as if it disclosed the 582 Patent's file partition allocation system is not consistent with how a POSITA would understand these references when claim 1 is read in context.

185.

F. Dependent claims 3, 4, 5, 9, 10, and 11 are not taught or suggested by Reeve alone

186. I have reviewed the Office Action's Reeve only rejection as applied to dependent claims 3, 4, 5, 9, 10, and 11. In my opinion as a computer scientist specializing in parallel data processing, even apart from the deficiencies in the Office Action's mapping of independent claim 1, the Reeve only ground independently fails as applied to these dependent claims because each introduces an additional, distinct limitation that further narrows the scope of claim 1. Reeve's disclosures concern compile time tiling and runtime scheduling of tiles to threads over an iteration space. The Office Action's reliance on high level statements about combined results,

execution statistics reporting, and generalized capability statements does not establish the specific merging, record accumulation, report creating, or database query limitations recited in claims 3, 5, 10, and 11. (Office Action, p. 9.)

(a) A. Claim 3. Reeve does not teach the additional step of merging subtask outputs in step (e)

187. Claim 3 depends from claim 1 and adds the further requirement of "the step of merging said subtask outputs in step (e)." Technically, this limitation calls for a merge operation applied to subtask outputs as part of the step (e) output generation. It is not satisfied by a generic statement that results are combined at a high level.

188.

189. The Office Action maps claim 3 to Reeve by asserting that "the results from processed tiles are combined" and equates that to a merge, citing Reeve, 2:25–27, 8:9–11, and 19:38–42. (Office Action, p. 9.) In my opinion, those citations do not establish claim 3 for at least three reasons.

190. First, the Office Action's mapping treats the word combined as synonymous with merged without identifying a disclosure in Reeve of a merge step performed on subtask outputs. Reeve's cited passages are high level descriptions that execution of all tiles yields an overall result of the parallel construct. (Reeve, 2:25–27; Reeve, 8:9–11; Reeve, 19:38–42.) They do not describe a distinct merge operation applied to discrete outputs produced by subtasks.

191. Second, Reeve's execution model does not identify "subtask outputs" in the sense required by claim 3. Under Reeve's runtime tiling framework, tiles are executed as portions of a parallel construct, and the results of executing tiles are often realized as updates to shared program state during execution rather than as discrete, separable output artifacts per thread that are later merged. The Office Action does not identify where Reeve teaches that the work units corresponding to subtasks produce discrete subtask outputs that are subsequently merged. (Reeve, 19:38–42.)

192. Third, claim 3 requires an additional step of merging subtask outputs in step (e), beyond the step (e) requirement in claim 1 of generating an output combining all subtask outputs. The Office Action's Reeve citations do not identify a distinct additional step. Even if a high level combined results statement were treated as relevant to claim 1, it does not automatically disclose claim 3's further requirement of a separate merging step. (Office Action, p. 9; Reeve, 2:25–27; Reeve, 8:9–11; Reeve, 19:38–42.)

193. For these reasons, in my opinion, Reeve does not teach claim 3's additional limitation requiring the step of merging subtask outputs in step (e). (Office Action, p. 9.)

(b) Claim 4. Reeve does not teach a one-to-one correspondence between input records and output records

194. Claim 4 depends from claim 1 and adds the further requirement of "the output in step (e) is a succession of outputs from said subtasks in a one-to-one correspondence with said records of said input file." In other words, it requires that there be one output record per input record. The office action states that "Reeve's tiled program has the same results as the original program" and cites to 8:9-11 and 19.38-42. These passages indeed indicate that Reeve's tiled program has the same results as the original program, and therefore have a one-to-one correspondence to the output records of the original program. However, this is not what the claim requires. The claim requires a one-to-one correspondence between the input records and the output records, not a one-to-one correspondence between the output records of the original program and the output records of the parallelized version of it.

195. Going back to our French toast analogy. Claim 4 requires that the number of French toast slices have a one-to-one correspondence to the raw bread slices input into the process. The cited parts of Reeve only have a one-to-one correspondence between the French toast slices produced a parallel group of 10 chefs and the French toast slices produced by a single, individual chef.

(c) Claim 5. Reeve does not teach an accumulation of output records from subtasks in an arbitrary order

196. Claim 5 depends from claim 1 and further requires that "the output in step (e) is an accumulation of output records from said subtasks in an arbitrary order." This is a narrowing output formation limitation. It requires that the step (e) output is formed as an accumulation of output records produced by subtasks, and that the accumulation is in an arbitrary order.

197. The Office Action does not identify where Reeve discloses subtasks producing output records that are accumulated into a record based output. Reeve is directed to tiling an iteration space and scheduling tiles to threads for executing a parallel construct. Its disclosures concern how tiles are assigned and executed, not how an output is formed as an accumulation of record outputs. (Reeve, 39:3–6.) In many tiled computations, the results are expressed as updates to program state rather than as output records emitted by each worker and accumulated into an output stream. The Office Action does not identify a Reeve disclosure of output records produced by subtasks and accumulated into the step (e) output. (Office Action, p. 9; Reeve, 39:15–17.)

198. Nor does Reeve disclose that any output is accumulated in an arbitrary order. Reeve's GRAB discussion states that tiles are assigned to threads using a first come first served algorithm. (Reeve, 39:4–6.) That is a work issuance policy for tiles to threads. It is not a disclosure that the output itself is an accumulation of output

records in an arbitrary order. A system can assign work units first come first served and still produce an ordered output. Claim 5 speaks to the ordering characteristic of the output produced in step (e), whereas Reeve's first come first served language concerns scheduling of work units. (Reeve, 39:4–6.)

199. To the extent the Office Action relies on generalized statements in Reeve that results are combined after tile execution, such statements do not teach claim 5. High level "combined results" language does not disclose an accumulation of output records, and it does not disclose accumulation in an arbitrary order. (Reeve, 19:38–42; Reeve, 39:15–17.)

200. Moreover, Reeve explicitly excludes the possibility that results can be accumulated in arbitrary order. Reeve defines a "correct execution" as meaning that an execution produces "the same result as by running the same program serially" (Reeve, 19:39–42.). If the output order is different than running the same program serially (because it was accumulated in arbitrary order), this would not be a correct execution using this definition.

201. Accordingly, in my opinion, Reeve does not teach claim 5's additional limitation. (Office Action, p. 9.)

(d) Claims 9 and 10. A post run cleanup routine that emits statistics does not teach that the computer executable process is a statistical analysis or report creating process

202. Claim 9 recites that "the method defined in claim 1 wherein said computer executable process is a statistical analysis process." and claim 10 recites that "the method defined in claim 1 wherein said computer executable process is a report creating process." Technically, these limitations concern the nature of the process that is being executed under the claim 1 framework, meaning the process that is partitioned into subtasks and whose outputs are combined. A POSITA would understand a "statistical analysis process" and "report creating process" to be a process whose substantive purpose and output is the creation of a statistical analysis or report, not merely a process that logs execution metrics or emits diagnostic information about the run.

203. The Office Action maps claims 9 and 10 to Reeve 30:13-14 --- i.e. Reeve's statement that "pr_end is executed once when the program finished. In addition to general cleanup, it produces statistics and reports." In my opinion, this does not teach claims 9 and 10. The cited passage describes a termination routine executed once after the program completes its substantive work, which performs cleanup and produces statistics and reports about the execution. These are reports about program operation and performance, not the substantive report output that defines the purpose of a statistical analysis process or report creating process.

204. In HPC and distributed computing systems, it is routine for programs to emit end of run statistics, logs, or summaries, including counters, timing metrics, and diagnostic output. Treating such routine telemetry as sufficient to satisfy a "statistical analysis process" or "report creating process" limitation would effectively eliminate the narrowing effect of claims 9 and 10, because virtually any program that logs runtime statistics could be characterized as producing "statistics" or "reports." Claims 9 and 10 instead require that the underlying computer executable process itself is a statistical analysis or report creating process, meaning the process is directed to producing a statistical analysis or report as its substantive output. Reeve's pr_end disclosure does not teach that. (Reeve, 30:13–14.)

205. Accordingly, in my opinion, the Office Action has not identified a teaching in Reeve that the computer executable process executed under claim 1 is itself a statistical analysis or report creating process, as opposed to merely including a post execution cleanup routine that emits diagnostic statistics and reports.

(e) Claim 11. Reeve does not teach that the computer executable process includes a database query

206. Claim 11 depends from claim 1 and further requires that the computer executable process includes a database query. Technically, a database query denotes executing a query operation against a database system (such as Oracle, PostgreSQL, MySQL, SQLServer, Snowflake, etc.), such as issuing a query statement or predicate

and retrieving records responsive to that query. It is not satisfied merely by parallel execution, memory access, or generic data processing.

207. The Office Action does not identify any disclosure in Reeve of a database query. Reeve is focused on compile time tiling and runtime scheduling of tiles to threads for executing a parallel construct over an iteration space, including tile assignment strategies such as GRAB. (Reeve, 39:3–6.) Reeve does not describe database structures, query statements, query predicates, or database query execution. Its discussion is framed in terms of iteration space and scheduling, not application semantics such as querying a database. (Reeve, 39:15–17.)

208. General statements that a parallel execution framework could be used with different types of applications do not establish that the process includes a database query. Claim 11 requires the specific feature of a database query as part of the executed process, and the Office Action does not point to any Reeve passage that teaches that feature. (Office Action, p. 9.)

209. For these reasons, in my opinion, the Office Action does not meet its burden to show that Reeve alone teaches or suggests the additional limitations recited in dependent claims 3, 4, 5, 9, 10, and 11, and the Reeve only rejection fails as applied to those claims. (Office Action, p. 9.)

VIII. CONCLUSION

210. For the reasons set forth above, it is my opinion that the Office Action does not establish a prima facie case that claims 1–14 of the '582 Patent are unpatentable over the cited references.

211. With respect to the Iyer + Reeve ground, the Office Action acknowledges that Iyer does not explicitly teach claim 1's "thereafter repeating" limitation and attempts to supply that limitation by importing Reeve's GRAB strategy. In my opinion, that approach does not cure Iyer's deficiency without fundamentally departing from Iyer's disclosed one-processor-to-one-partition architecture. Iyer teaches selecting the number of lists and partitions to match the number of available processors and emphasizes avoiding processor load imbalance by construction. Converting Iyer into an over-subscribed pool-based repeated-claiming regime would require substantial redesign, including intentionally defining more partitions than processors, maintaining shared allocation state, distributing partition descriptions so processors can claim additional unprocessed partitions, and addressing correctness and output-formation constraints introduced by having processors execute multiple partitions sequentially. The Office Action does not address these incompatibilities or provide a technically grounded explanation for how this redesigned system would operate consistently with Iyer's described merge and concatenation flow.

212. With respect to the Reeve-only ground, it is my opinion that the Office Action's mapping relies on an over-abstract characterization of Reeve that substitutes generic "parallelism" for the specific file-partition load-balancing architecture required by claim 1. Claim 1, as explained in the '582 Patent, is anchored to file-allocation-based partitioning of records in a preexisting input file into file-range partitions that can be defined without reading file contents, followed by dynamic allocation of unallocated file partitions to subtasks until all partitions are processed, and generation of an output by combining or merging subtask outputs into a result output equivalent to that of the original process. Reeve, by contrast, discloses compile-time iteration-space tiling and runtime scheduling of tiles to threads for executing parallel constructs, where tiling is defined by indices, bounds, and tile sizes and where the runtime looks at the iteration space rather than the data space. In my opinion, Reeve does not teach the '582 Patent's file-partition mechanisms, including file-allocation-based partition definition, repeated allocation of unallocated file partitions to subtasks, or the discrete subtask-output and merge-into-result-output workflow described in the '582 Patent.

213. In addition, dependent claims 3, 4, 5, 9, 10, and 11 include further narrowing limitations that are not taught or suggested by Reeve, and dependent claims 5, 10, and 11 are not taught or suggested by the proposed Iyer + Reeve combination. In my opinion, the Office Action does not identify disclosures in the cited references

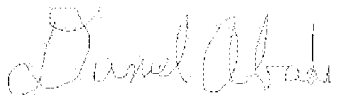
establishing the specific limitations of a merge step applied to subtask outputs, accumulation of output records in an arbitrary order, a report-creating process as the substantive executed process, or a computer-executable process that includes a database query.

214. Accordingly, it is my opinion that the rejections set forth in the Office Action should be withdrawn and that claims 1–14 of the '582 Patent are patentable over the references cited in the reexamination proceeding.

IX. DECLARATION

215. I, Dr. Daniel Abadi, declare that all statements made herein on my own knowledge are true, and all statements made on information and belief are believed to be true. Further, I am aware that these statements are made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under 18 U.S.C. § 1001. I declare under penalty of perjury that the foregoing is true and correct.

Executed on February 17, 2026, in College Park, Maryland.



Dr. Daniel Abadi