PMG: Multi-core Metabolite Identification [1]

Mohammad Mahdi Jaghoori[a,b,c,2], Sung-Shik T.Q. Jongmans[b],
Frank de Boer[b], Julio Peironcely[c], Jean-Loup Faulon[d],
Theo Reijmers[c] and Thomas Hankemeier[c]

[a] *Academic Medical Center (AMC), Amsterdam, The Netherlands*

[b] *Center for Mathematics and Computer Science (CWI), Amsterdam, The Netherlands*

[c] *Leiden Academic Center for Drug Research (LACDR), Leiden, The Netherlands*

[d] *Institute of Systems and Synthetic Biology (iSSB), Evry, France*

Abstract

Distributed computing has been considered for decades as a promising way of speeding up software execution, resulting in a valuable collection of safe and efficient concurrent algorithms. With the pervasion of multi-core processors, parallelization has moved to the center of attention with new challenges, especially regarding scalability to tens or even hundreds of parallel cores. In this paper, we present a scalable multi-core tool for the metabolomics community. This tool addresses the problem of metabolite identification which is currently a bottleneck in metabolomics pipeline.

Keywords: multi-core, Java concurrency, fork-join, scalability, metabolomics

1 Introduction

Metabolomics is the study of the intermediate molecules and the final products of *metabolism*: the chemical reactions in living organisms. For example, in the early 2000s, Gavaghan et al. showed that one can distinguish white mice from black mice by chemically analyzing their urine [2]. Metabolomics requires well-designed and efficient software tools to analyze the data obtained during lab experiments. Currently, a major bottleneck in the pipeline is metabolite identification: having determined which particular chemical *elements* (i.e., atoms) exist in a particular sample (of cells, body fluids, etc.), it is nontrivial to come up with the structure of the corresponding chemical *compounds* (i.e., molecules), especially when dealing

---

[2] Email (corresponding author): m.jaghoori@amc.uva.nl

with a so far unknown metabolite. Often, researchers must propose candidate structures for a metabolite themselves. This requires solving a daunting combinatorial problem with a potentially huge search space.

In this paper, we describe the Java-based implementation of a multi-core *Computer Assisted Structure Elucidation* (CASE) tool, called *Parallel Molecule Generator* (PMG), for automatic molecular structure generation. PMG is an evolution of the open-source CASE tool, called *Open Molecule Generator* (OMG) [8]. A popular approach to implementing CASE tools is to view the molecular structure as a graph. The problem of structure elucidation can then be mapped to isomorph-free exhaustive graph generation [5, Section 4]. In Section 2, we explain the main idea behind the algorithms involved. In Section 3, we describe two parallel implementations of PMG using contemporary concurrency tools from Java. These high-level concurrency tools both make programming less error-prone, and have efficient and optimized implementations. Based on experimental evidence, we then discuss their scalability and the resulting speedup in Section 4.

The contribution of the paper has two sides. To *metabolomics*, we provide a parallel, scalable, open-source CASE tool. The open-source nature of the tool allows for further optimizations and addition of many other relevant features. Our contribution to *computer science* is an evaluation and comparison of concurrency tools in Java. We have not seen similar studies before, particularly on the possible merits of the fork/join framework [7]. Additionally, PMG can be seen as a success story in developing scalable parallel programs for the multi-core era.

To give a rough indication of PMG's performance, in its current state, given around ten cores, PMG can compete with and beat the fastest commercial alternative, MolGen [3] [4,6], as shown in Figure 1. MolGen outperforms PMG in the leftmost and in the middle case. However, when provided a known fragment of the structure, [4] such as in the rightmost case, PMG outperforms MolGen.
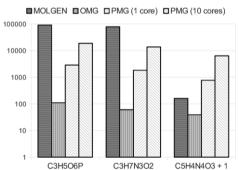

Fig. 1: Molecular structures generated per second for MolGen, OMG, PMG.

2 Molecular Structure Generation

Computer Assisted Structure Elucidation refers to software tools that given an elemental composition (something like C4H5N3O) generate every chemically possible molecular structure with those atoms. These tools generally use graphs as a natural representation of molecules, where atoms and bonds—connections between

---

[3] MolGen exists more than 20 years. See http://www.molgen.de.

[4] Sometimes, the lab experiments give extra information about small fragments of the whole molecular structure, which CASE tools should use to prune the search space.

Listing 1. Implementation of OMG – abstract code.

```
class SearchTree {
  Molecule molecule; // data structure representing the current molecule
  void expand() {
    List<SearchTree> extended = addOneBond();
    for (SearchTree em : extended)
      em.expand();
} }
```

atoms—translate into vertices and edges. Double and triple chemical bonds are then translated to two and three parallel edges. Furthermore, the maximum degree of each vertex is set to the *valence* of its corresponding atom: the maximum number of bonds the atom can form (e.g., a carbon atom can form at most four bonds).

Essentially, structure generation starts from a collection of unconnected vertices (atoms) and iteratively adds edges (bonds) between them. Thus, the algorithm constructs and explores a search tree, similar to the one in Figure 2, in which every leaf node represents a completed molecular structure. After having generated the entire search tree, the algorithm collects and returns the molecular structures in the leaves as output. However, as depicted in Figure 2, the naive construction of graphs by incrementally adding edges can result in duplicate graphs (e.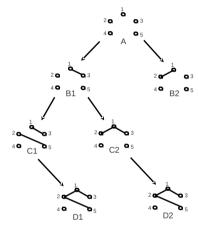g., in nodes D1 and D2) and isomorphic graphs (e.g., in nodes B1 and B2).[5] Structure generation algorithms should filter those

Fig. 2: A sample search tree.

out. Otherwise, we may end up with millions of duplicates. One approach to do such filtering is called *orderly generation*: first, we define a total order relation on edges and graphs. This algorithm adds an edge e to a graph G in the current node only if e is bigger than or equal to the edges in G. Colbourn and Read [1] have shown that in this setting, expanding only nodes containing a minimal graph guarantees generation of all possible graphs without duplicates.

Listing 1 shows an abstract of the sequential structure generation algorithm used in OMG (which we parallelized—see Section 3). It starts off with a set of atoms without bonds. In each node of the search tree, OMG uses the method addOneBond to find all the possibilities for adding one bond to the current molecular structure. The objects in the returned list serve as new nodes in the search tree, which OMG subsequently expands in a DFS fashion.

3 Two Implementations of PMG

Generally, there are two approaches to make the source code in Listing 1 parallel. The first approach is to decide at the beginning of the program on dividing the

---

[5] Isomorphic graphs represent the same molecule, and hence, graph isomorphism provides a sufficiently strong notion of equality for structure generation algorithms.

search tree in n parts (assuming n parallel cores). This approach has very low overhead and is used mainly on grids of multiple computers. It is, however, not optimal on a multi-core and in particular for this problem because the search tree is not balanced: some of the parts may be small, leaving the corresponding cores underutilized. Another approach is to break down the work into millions of small *tasks*, avoiding the risk of underutilization. However, if tasks are too small, the overhead of managing tasks outweighs the actual computation. In our case, the computation in each node is a proper task size.

Currently, Java offers roughly two high-level frameworks on top of basic Java threads for writing concurrent programs: the *fixed-size* (FS) executor service framework [3] and the *fork/join* (FJ) framework [7]. Because we were not sure about which of those two frameworks would suit our application best in terms of scalability, we wrote two implementations of PMG: one using the FS executor services, referred to as PMG_{FS}, and the other using the FJ framework, referred to as PMG_{FJ} [6]. We refer to Appendix A for a short primer on executor services and fork/join concurrency in Java.

Listing 2. Implementation of PMG_{FS} (left) and PMG_{FJ} (right) – abstract code.

```
class SearchTree implements Runnable {          class SearchTree extends RecursiveAction {
  Molecule molecule;                              Molecule molecule;
  void run() {                                     void compute() {
    List<SearchTree> extended = addOneBond();        List<SearchTree> extended = addOneBond();
    for (SearchTree em : extended)                   for (SearchTree em : extended)
      if (task queue is full) em.run();                em.fork();
      else executor.submit(em);                    }
} }                                              }
```

PMG_{FS}: Fixed-Size-Executor-Service Implementation

At startup on a machine with n cores, PMG_{FS} creates a basic thread pool executor service with n worker threads. Fixing the number of worker threads to the number of cores enables the utilization of all available cores while eliminating the overhead of dynamically adjusting the size of the thread pool. (Having more than n worker threads results in context switching overhead.)

Listing 2 (left) shows an abstract of the source code for the parallel structure generation algorithm used in PMG_{FS}. Compared to the sequential source code in Listing 1, every node in the search tree now corresponds to a potentially parallel task in PMG_{FS} (`SearchTree` implements `Runnable`—see Appendix A). When processing a node, a worker thread in PMG_{FS} either submits its child nodes to the executor service as new tasks (making them available for parallel execution by other worker threads) or processes them directly. This choice depends on the *task queue* of the executor service, which holds pending tasks: if sufficiently full (meaning that the queue contains enough work for other worker threads upon completing their current tasks), worker threads in PMG_{FS} avoid task submission to reduce contention over the shared task queue.

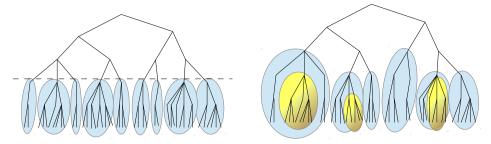[6] Available on git://git.code.sf.net/p/pmgcoordination/pmgcoordination.

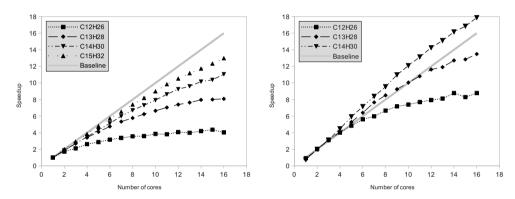Fig. 3. Task sizes: (static) threshold vs. dynamically sized.

Fig. 4. Speedup of PMG$_{\text{FS}}$.

Fig. 5. Speedup of PMG$_{\text{FJ}}$.

PMG$_{\text{FJ}}$: Fork/Join-based Implementation

At startup on a machine with n cores, PMG$_{\text{FJ}}$ creates an FJ executor service with roughly n worker threads (due to its internal workings, the FJ framework may create a few more than n worker threads). Listing 2 (right) shows an abstract of the source code for the parallel structure generation algorithm used in PMG$_{\text{FJ}}$. Instead of calling `submit` on an executor service, PMG$_{\text{FJ}}$ calls `fork` on an FJ task (`SearchTree` extends `RecursiveAction` which extends `ForkJoinTask`—see Appendix A). Because every worker thread has its own (more-or-less) private task queue in the FJ framework, worker threads in PMG$_{\text{FJ}}$ do not need to check the fullness of task queues to reduce contention.

In order to avoid too small tasks, the recommended way to use fork/join is to use a threshold to stop creating parallel tasks (Figure 3 left). In unbalanced search trees, the threshold must be dynamically adapted based on the queue size. In our implementation, we further optimized this technique by keeping the possibility of spawning new tasks even after the threshold is reached (Figure 3 right). By deciding on-the-fly whether to spawn a new task or continue sequentially, we are actually changing the 'task size' dynamically (which was discussed up at the beginning of this section).

4   Scalability: Experiments and Results

To investigate the scalability of our two implementations of PMG we ran them on a machine with two Intel Xeon E5-2650L processors, yielding a total of sixteen cores. We initially carried out three experiments. In every experiment, we ran PMG_{FS} and PMG_{FJ} for $1 \leq i \leq 16$ cores (using the `taskset` command on Linux to restrict the number of cores) and averaged our results over eight runs for each i: we called them on a small molecule consisting of twelve carbon atoms and 26 hydrogen atoms (C12H26) in the first experiment, a medium molecule (C13H28) in the second, and a big molecule (C14H30) in the third. Figures 4 and 5 shows our experimental results. The gray baseline represents linear speedup (i.e., optimal scalability).

To make Figure 5 more meaningful, we computed speedup of PMG_{FJ} relative to two cores instead of one. Otherwise, we would observe considerable more-than-linear—*superlinear*—speedup (we still have some superlinear speedup for the C14H30 runs). Superlinear speedup results from the relatively poor performance of the FJ framework on few cores, especially for $n = 1$, caused by bookkeeping. As the number of cores (or task size) increases, however, the bookkeeping overhead becomes negligible. Therefore, computing speedup relative to two cores yields more meaningful figures.

PMG_{FS} scales worse than PMG_{FJ}; all measurements for PMG_{FS} fall below the baseline. The improvement between C12H26, C13H28, and C14H30, however, suggests that the scalability of PMG_{FS} improves as the number of tasks or the average task size increases. To further investigate this trend, we did an additional experiment in which we ran PMG_{FS} on the even bigger molecule C15H32 (on the same machine with the same configuration for $1 \leq i \leq 16$ cores using `taskset`). Again, we observe an improvement in speedup, albeit less than before. This suggests that the speedup observed in the C15H32 runs approaches the limit of the inherent scalability of PMG_{FS}: for sufficiently large molecules, it gets roughly 80% faster every time we double the number of cores. Although a decent result, the inherent scalability of PMG_{FJ} seems to approach optimal speedup. For our application, thus, the FJ framework outperforms the ES framework.

5   Conclusion

The contribution of this work can be seen from two sides: Computer Science and Metabolomics. *Firstly*, we provide the metabolomics community a multi-core, scalable, open-source metabolite identification tool. We explained the workings of the main algorithm involved and described two implementations using different concurrency frameworks from Java. Such scalable tools are necessary in the multi-core era enabling faster execution of the programs by buying newer hardware, just as in the old times before processor speeds stopped increasing.

Secondly, we provide the computer science community an evaluation and comparison of concurrency tools in Java on a nontrivial real-world case. Based on a number of experiments, we conclude that the recently introduced fork/join frame-

work outperforms older Java technology (as expected). As future work, we are making a library that hides the complexity of handling dynamically-sized tasks from the programmer, thus keeping the simplicity of fork/join programming. Nonetheless, if this approach is taken up in the default implementation of fork/join, one can benefit from under-the-hood optimizations.

References

[1] Colbourn, C. and R. Read, *Orderly algorithms for graph generation*, International Journal of Computer Mathematics **7** (1979), pp. 167–172.

[2] Gavaghan, C., E. Holmes, E. Lenz, I. Wilson and J. Nicholson, *An NMR-based metabonomic approach to investigate the biochemical consequences of genetic strain differences*, FEBS Letters **484** (2000), pp. 169–174.

[3] Goetz, B., T. Peierls, J. Bloch, J. Bowbeer, D. Holmes and D. Lea, "Java Concurrency in Practice," Addison-Wesley, 2006.

[4] Gugisch, R., A. Kerber, A. Kohnert, R. Laue, M. Meringer, C. Rücker and A. Wassermann, *MOLGEN 5.0, a molecular structure generator*, Bentham Science Publishers Ltd (2012), submitted.

[5] Kaski, P. and P. Östergård, "Classification Algorithms for Codes and Designs," Algorithms and Computation in Mathematics, Springer, 2006.

[6] Kerber, A., R. Laue, T. Grüner and M. Meringer, *MOLGEN 4.0*, MATCH Commun. Math. Comput. Chem **37** (1998), pp. 205–208.

[7] Lea, D., *A Java Fork/Join Framework*, in: D. Gannon and P. Mehrotra, editors, *Proceedings of JAVA '00*, 2000, pp. 36–43.

[8] Peironcely, J., M. Rojas-Cherto, D. Fichera, T. Reijmers, L. Coulier, J.-L. Faulon and T. Hankemeier, *OMG: open molecule generator*, J. Cheminformatics **4** (2012), pp. 167–172.

A   Appendix: Java Concurrency Tools

Executor Services

Already in its early days, Java had basic support for thread-based concurrency. However, programming directly with threads is difficult. To ease this, Java 5 extended this support with new higher-level concurrency utilities in the form of the `java.util.concurrent` package: classes and interfaces aimed at simplifying concurrent (multi-core) programming in Java. This library includes a framework for *executor services*, which add a form of asynchronous message passing to Java.

Executor services accept asynchronous method calls, materialized as *submissions* of conceptual tasks. One can submit any object implementing Java's `Runnable` interface (the task returns nothing) or `Callable` interface (the task returns a result) as a task to an executor service. Immediately after such a submission, the executor service involved returns a `Future` object to the submitter (i.e., the caller of the asynchronous method). The submitter can then use this object to poll the completion of the submitted task (i.e., termination of the called method) or to get the task's result (i.e., the method's return value). Internally, an executor service assigns to each submitted task a *worker thread* that subsequently executes that task. To avoid the overhead of creating a new worker thread for every task, executor services maintain *thread pools*. A thread pool may contain a static or dynamic number of

worker threads. As long as no one has submitted a task to the executor service, the worker threads in the thread pool are idle. As soon as a task becomes available, the executor service selects one of the idle worker threads for executing that task. Once the selected thread completes the task, it becomes idle again and available for executing the next task.

The number of submitted tasks can exceed the number of worker threads in the thread pool. To handle those situations, executor services have an internal *task queue*. Every submitted task first gets offered to the task queue. Worker threads in the thread pool obtain new tasks by polling the queue.

Fork/Join Framework

Java 7 extends the executor service framework of Java 5 with high-level and highly optimized support for *fork/join* (FJ) algorithms [7], the concurrent variant of classical divide-and-conquer. Essentially, this extension consists of a special FJ executor service to which threads can submit special FJ tasks (i.e., extensions of the abstract `ForkJoinTask` class instead of implementations of the `Runnable`/`Callable` interface). The FJ framework adds a layer of abstraction on top of the ordinary submission facilities of executor services: instead of "submitting" tasks, threads "fork" tasks. [7] Likewise, instead of receiving a `Future` object to await the result of a task, threads "join" earlier forked tasks.

The internal workings of the FJ executor service differ significantly from the basic thread pool executor service: the FJ executor service maintains multiple task queues—conceptually one per worker thread in the thread pool—and applies a *work stealing* algorithm to prevent worker threads from idling if their own task queue has emptied while another queue has not. The combination of private task queues for worker threads and work stealing should give the FJ framework significantly better performance than the basic thread pool executor service, at least for those class of problems that fit the FJ abstraction (i.e., algorithms involving recursive decomposition of a problem into independent subproblems).

---

[7] Technically, one can still submit tasks to the FJ executor service directly, instead of forking them, but this is generally not the intended use of the FJ framework.