Parallel Computing in Chemical Reaction Metaphor with Tuple Space

Hong Lin linh@uhd.edu
Department of Computer and Mathematical Sciences
University of Houston-Downtown
1 Main Street, Houston, Texas 77002, USA

Jeremy Kemp
Department of Computer and Mathematical Sciences
University of Houston-Downtown
1 Main Street, Houston, Texas 77002, USA

Wilfredo Molina
Department of Computer and Mathematical Sciences
University of Houston-Downtown
1 Main Street, Houston, Texas 77002, USA

Abstract

Methodologies have been developed to allow parallel programming in a higher level. These include the Chemical Reaction Models, Linda, and Unity. We present the Chemical Reaction Models and its implementation in IBM Tuple Space. Sample programs have been developed to demonstrate this methodology.

Keywords: Parallel Programming, Very High Level Languages, the Chemical Reaction Model, IBM Tuple Space.

1. Higher-Level Parallel Computing - Implicit Parallelism

Higher level parallel programming models express parallelism in an implicit way. Instead of imposing programmers to create multiple tasks that can run concurrently and handle their communications and synchronizations explicitly, these models allow programs to be written without assumptions of artificial sequenciality. The programs are naturally parallel. Examples of such kind of models include the Chemical Reaction Models (CRMs) [1, 2], Linda [3], and Unity [4, 5]. These models are created to address higher level programming issues such as formal program specification, program synthesis, program derivation and verification, and software architecture. Efficient implementation of these models has limited success and therefore obscures its direct applications in software design [6, 7]. Despite this limitation, efforts have been made in both academic and industrial settings to avail these models in real-world programming. For example, Unity has been used in industrial software design and found successful; execution efficiency of Linda has been affirmed by experiments and it is implemented by IBM Tuple Space. Recent discussions of these models in multi-agent system design have also been found in literature [8]. In the following discussion, we focus on the Chemical Reaction Models and its applications.

The Chemical Reaction Models describe computation as “chemical reactions”. Data (the “solution”) are represented as a multiset. A set of “reaction” rules is given to combine elements in

the multiset and produce new elements. Reactions take place until the solution becomes inert, namely there are no more elements can be combined. The results of computation are represented as the inert multiset. Gamma is a kernel language in which programs are described in terms of multiset transformations. In Gamma programming paradigm, programmers can concentrate on the logic of problem solving based on an abstract machine and are free from considering any particular execution environment. It has seeded follow-up elaborations, such as Chemical Abstract Machine (Cham) [9], higher-order Gamma [10, 11], and Structured Gamma [12]. While the original Gamma language is a first-order language, higher order extensions have been proposed to enhance the expressiveness of the language. These include higher-order Gamma, hmm-calculus, and others. The recent formalisms, γ-Calculi, of Gamma languages combine reaction rules and the multisets of data and treat reactions as first-class citizens [13-15]. Among γ-Calculi, γ_0-Calculus is a minimal basis for the chemical paradigm; γ_c-Calculus extends γ_0-Calculus by adding a condition term into γ-abstractions; and γ_n-Calculus extends γ_0-Calculus by allowing abstractions to atomically capture multiple elements. Finally, γ_{cn}-Calculus combines both γ_c-Calculus and γ_n-Calculus. For notational simplicity, we use γ-Calculus to mean γ_{cn}-Calculus from this point on.

The purpose of the presented study is to investigate a method for implementing γ-Calculus using IBM Tuple Space. TSpace supports network computation in client/server style. The target of this effort is to enable higher order programming in a parallel computing platform, such as computer clusters, and allow for refinement of the executable programs using transformation techniques.

The paper will be organized as follows. In Section 2, we give a brief introduction to γ-Calculus. In Section 3, we discuss the method for implementing γ-Calculus in IBM Tuple space. Program examples are presented in Section 4. We conclude in Section 5.

2. γ-Calculus

The basic term of a Gamma program is molecules (or γ-expressions), which can be simple data or programs (γ-abstractions). The execution of the Gamma program can be seen as the evolution of a solution of molecules, which react until the solution becomes inert. Molecules are recursively defined as constants, γ-abstractions, multisets or solution of molecules. The following is their syntax:

```
M ::=   0 | 1 | … | 'a' | 'b' | …     ; constants
    |   γP[C].M                       ; γ-abstraction
    |   M1, M2                        ; multiset
    |   <M>                           ; solution
```

The multiset constructor "," is associative and commutative (AC rule). Solutions encapsulate molecules. Molecules can move within solutions but not across solutions. γ-abstractions are elements of multisets, just like other elements. They can be applied to other elements of the same solution if a match to pattern P is found and condition C evaluates to true and therefore facilitate the chemical reaction. The pattern has the following syntax:

P ::= x | P, P | <P>

where x is a variable. In addition, we allow for the use of tuples (written x_1:… : x_n) and names of types. For example, γ-abstraction

γ(x: Int, y: Int)[x ≥ y].x

can be interpreted as: replace x, y by x if x ≥ y, which is equivalent to finding the maximum of two integers.

The semantics of γ-Calculus is defined as the following:

$(\gamma p[c].m_1), m_2$	$= \phi m_1$ if match(p/ m_2) = ϕ and ϕc	; γ-conversion
m_1, m_2	$= m_2, m_1$	; commutativity
$m_1, (m_2, m_3)$	$= (m_1, m_2), m_3$	; associativity
$E_1 = E_2$	$=> E[E_1] = E[E_2]$	; chemical law

The γ-conversion describes the reaction mechanism. When the pattern p matches m_2, a substitution ϕ is yielded. If the condition ϕc holds, the reactive molecules $\gamma p[c].m_1$ and m_2 are consumed and a new molecule ϕm_1 is produced. match(p/m) returns the substitution corresponding to the unification of variables if the matching succeeds, otherwise it returns fail.

Chemical law formalizes the locality of reactions. $E[E_1]$ denotes the molecule obtained by replacing holes in the context E[ ] (denoted by [ ]) by the molecule E_1. A molecule is said to be inert if no reaction can be made within:

Inert(m) □
$(m \equiv m'[(\gamma p[c].m_1), m_2] => match(p/m_2) = fail)$

A solution is inert if all molecules within are inert and normal forms of chemical reactions are inert γ-expression. Elements inside a solution can be matched only if the solution is inert. Therefore, a pattern cannot match an active solution. This ensures that solutions cannot be decomposed before they reach their normal form and therefore permits the sequentialization of reactions. The following inference rule governs the evolution of γ-expressions:

$$\frac{E_1 \rightarrow E_2 \quad E \equiv C[E_1] \quad E' \equiv C[E_2]}{E \rightarrow E'}$$

This high level language can be implemented in Java using IBM's TSpaces server and Java package. The method is detailed in the following section.

3. IBM Tuple Space

IBM Tuple Space was originally invented as an implementation of Linda computational model. While version 3.0 is only available after obtaining a site license, version 2.12 is freely available. Installing TSpaces is as easy as unpackaging the TSpaces package on the networked file system (NFS), adding it's directory to the users classpath, and starting the server in a GNU Screen session.

3.1 Data Structures and Methods

A TSpaces program uses more of a client/server model, with each node only communicating with a 'host' or any machine with a known name, which on a cluster would usually be the head node. And although TSpaces is flexible enough to assign global names and ranks to each node, micro-managing the communications, this would defeat the purpose of having the abstraction layer TSpaces offers you. Data in a TSpaces program is shared through Tuples, a tuple is a data structure that wraps all other data structures in a tuplespace, this can include data primitives, as well as standard java classes and user defined classes. Every tuple is in a TuplesSpace, and every TupleSpace has a host (actually a fully qualified host name).

The TSpaces methods used to obtain tuples are: read(), waitToRead(), take(), waitToTake(). The read() and take() methods vary in that the read() method leaves the tuple returned to the program in the tuplespace, and the take() method removes it from the tuplespace. The waitTo versions wait until a tuple appears in a tuplespace, and then takes it; these are good for synchronization, and can be set to time out in order to prevent the program from hanging indefinitely. These

methods take Tuples as arguments, and return the first tuple from the tuplespace that matches either the data types you specified, or specific values. There is also the scan() method, which works like read() except it returns all values that match the tuple specified. There are other Tspace methods that allow you more control over your program; countN() returns all the matching tuples and delete() removes all matching tuples. There are many other advanced features that allow you to manipulate TSpaces with a great deal of precision.

Another bonus of using a platform that runs on Java is being able to run these programs on any OS, and even inside a web browser using an applet. Such applets are ideal for being able to monitor or visualize a reaction that is running.

3.2 Synchronization

The most difficult task in writing a CRM/gamma program in Tspaces is synchronization. Synchronization is needed to determine when to terminate the program, or when the molecules stop reacting.

One program acts as one molecule, and different programs can read from the same tuplespace, but in order to test that the reaction of a molecule is exhausted, a program must run through the entire data space. So, reactions that involve more then one molecule type should be run in a cycle, the first program reacting all of the molecules from the original tuplespace, and writing the resulting tuples, as well as all of the unused tuples to the next tuplespace. The next program acts on this tuplespace until it is stable with regard to the molecule it represents, leaving the tuples in a third tuplespace. This continues until all of the programs that represent molecules have had a chance to run, and then it starts over. Termination occurs when the programs cycle through all of the programs without any changes to the tuples.

Molecules that reduce tuplespaces to a single solution are the simplest. They require only one tuplespace and no synchronization. The max number finder and the tuple adder are examples of this, they simply react until there is only one element left, then they are done.

After some further thought (but not further programming) a relation of numbers of tuplespaces to input and output elements of the gamma function can be noticed. In order to use the style of synchronization used in the sorting program, a program with X input molecules and Y output molecules requires X tuplespaces for the input and Y tuplespaces for the output. In order to detect stability, it will have to empty the input tuples 1 time with no changes and empty the output tuplespace 1 time.

The sorting program is an example of this; the method it uses involves alternating between comparing two tuples of the form (even index n, dataValue), (n +1, dataValue) and (odd index m, dataValue), (m+1, dataValue), and incrementing a counter tuple whenever a change occurs. This allows a comparison after the last tuples are removed from the tuplespace and into another tuplespace to determine if any changes have been made. If two consecutive changeless runs occur, then every data element is in order, and the program terminates.

There is a complication with programs where number of inputs is not equal to the number of outputs. The indexing must be handles specially or removed entirely; with numbers removed, there will be n + 1 elements missing, and with numbers added, there will be elements that need to be added somewhere, these can usually be appended to the end?

If there are more inputs then outputs, then the tuplespace will eventually be reduced to the number of outputs for the one molecule. These types of programs can use the termination style of the max element and tuple adder programs; simply running until there are Y elements left, and then stopping. The only synchronization required is when emptying a tuplespace(or set of tuplespaces), and to prevent deadlock when (number of elements) < (number of processors) * (number of inputs), but this can be handled by detecting null reads and random timeouts on the waitToRead() calls.

Although Tuple Space was initially implemented to support Linda computation model, its functions well suite in the operational semantics of the chemical reactions models. We propose implementing γ-Calculus on Tuple Space. In the following, we demonstrate a few sample programs in γ-Calculus and their Tuple Space implementation.

4. Examples

4.1 Max Number Finder

Given a set of values of an ordered type M, this program returns the maximum number of the set. The following γ-abstraction compares two randomly selected values. If the first value is greater than or equal to the second, it removes the second value from the set:

select = γ(a: M, b: M)[a ≥ b]. a: M, select

No global control is imposed on the way multiset elements are selected to ignite the reaction. If select is placed in an initial set M_0 of values, it will compare two values and erase the smaller at a time till the maximum is left. So the maximum number program can then be written as:

Max M_0 = <select, M_0>

If the multiset M_0 is represented as a tuple space, this program can be converted into one that finds and displays the greatest tuple inside a tuple space. It works with each node taking two tuples from the tuple space, comparing them, and placing the greatest one back to the tuple space. This process repeats itself until the termination condition is met, that is, when there is only one tuple left in the tuple space. When a node picks tuples up, if both tuples happen to be the same size, it simply places one of them back in the tuplespace while discarding the other one. If a node happens to take only one tuple because another node already picked the last remaining ones in the tuple space, this puts it back and repeats the process. This ensures that by the next check, a node will be able to take two tuples and perform the remaining operations to find the greatest tuple. If a node sees no tuples in the tuple space, this displays a message and terminates. If a node sees only one tuple in the tuple space, it assumes the greatest tuple was already found, displays a message and terminates.

Check appendix A for an example of the code implemented in TSpaces as well as its flowchart.

4.2 Tuple Adder

Given a set of values of numerical type M, we write a program to summarize all the values. The following γ-abstraction adds two randomly selected values and put the sum back into the multiset:

add = γ(a: M, b: M)[true]. a+b: M, select

The tuple adder program can then be written as:

Sum M_0 = <add, M_0>

If M_0 is represented as a tuple space, the corresponding TSpace program will add all of the tuples in a tuple space and displays their total sum. It works with each node taking two random tuples from the tuple space, adding them up, and placing a new tuple with their total sum back in the tuplespace (the tuples themselves are discarded). This process repeats itself until there is only one tuple left in the tuplespace, which is the total sum. If there are no tuples in the tuplespace before execution, the nodes display a message and terminate.

Check appendix B for the flowchart of the code implemented in TSpaces. Code is omitted because of the page limit.

4.3 Sorter

If a list is represented by multiset M = {(a, i) | a is value and i an index and i's are consecutive}, the following recursive γ-abstraction replaces any ill-ordered pairs by two other pairs:

sigma = γ((a, i): M, (b, j): M)[i < j ∧ a > b]. (b, i): M, (a, j): M, sigma

It specifies that any two selected pairs (a, i) and (b, j) that satisfy the condition, i < j □ a > b are replaced by two other pairs (b, i) and (a, j), and a copy of itself. If sigma is placed in an initial set M_0 of pairs, it will replace ill-ordered pairs until the entire list is sorted. So a sorting program can be defined as:

Sort M_0 = <sigma, M_0>

In a tuple space, a similar process will happen. The program will sort all of the tuples in a tuple space in ascending order. Each tuple has an index and a value in the following format: (index, value). When two tuples, (i, x) and (j, y) from said tuple space S are taken by a node, it first checks whether x > y & i < j. If this happens to be true, then the following swap is performed: (i, y), (j, x) they are put back in the tuple space, and the tuples are in order. This process repeats itself until no more swaps can be performed, that is, when all of the tuples in a tuple space are arranged in ascending order.

As mentioned above, multiple tuplespaces are required to synchronize this 'single pool' abstraction, in this case four tuplespaces were used. There is a primary pool, where the data is initially stored and an alternate pool where the data is written as it is being processed. Each of these pools is broken in to an even and an odd pool

Check appendix C for the flowchart of the code implemented in TSpaces, the primary feature of this programming model, is that it can utilize up to n/2 processing nodes, where n is the number of data items being sorted.

We have tested the above TSpace programs on a PC cluster and observed the computation in multiple nodes, and how the increase of nodes divides the number of comparisons per node, and increases speed; all of this thanks to the abstractions and portability offered by TSpaces.

We want to point out that when converting a γ-Calculus program into a TSpace program, details must be added to make a working program. However, through the above examples, we can see the conversion is straightforward in sense of the computation model. Therefore, it is technically feasible to design an automatic conversion system that can parse γ-Calculus and convert the program into a TSpace program and this is the next goal of our ongoing project.

5. Conclusion

The Chemical Reaction Models are higher level programming models that address parallelism implicitly and allows programmers to focus on the logic of problem solving instead of deal with operational details in parallel computing. IBM Tuple Space supports client/server computation based on Linda model that uses a similar concept for data structures. We discuss a method for implementing a higher order chemical reaction model, γ-Calculus, in IBM Tuple Space. We present the rules for converting γ-Calculus constructs into TSpace codes and discuss the critical techniques such as synchronizations. Our work shows that the conversion is practical. Experiments are also conducted on a computer cluster.

6. Acknowledgements

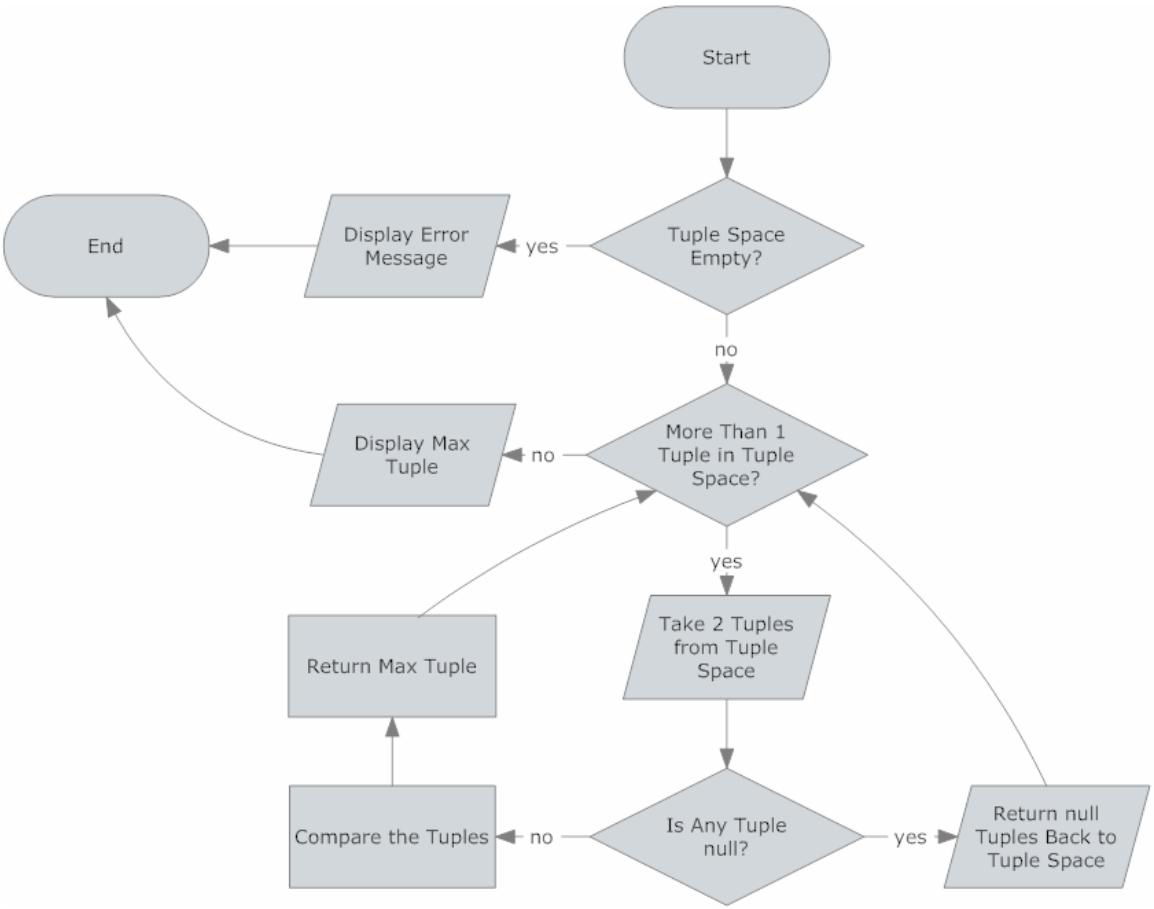
This research is partially supported by NSF grant “Acquisition of a Computational Cluster Grid for Research and Education in Science and Mathematics” (#0619312).

7. References

1. Banatre, J.-P. and Le Metayer, D. “*The Gamma model and its discipline of programming*”. Science of Computer Programming, 15, 55-77, 1990.
2. Banatre, J.-P. and Le Metayer, D. “*Programming by multiset transformation*”. CACM, 36(1), 98-111, 1993.
3. Carriero, N. and Gelernter, D. “*Linda in context*”. CACM, 32(4), 444-458, 1989.
4. K.M. Chandy and J. Misra. “*Parallel Program Design: A Foundation*”, Addison-Wesley (1988)
5. Misra, J. “*A foundation of parallel programming*”. In M. Broy (ed.), Constructive Methods in Computing Science. NATO ASI Series, Vol. F55, 397-443, 1989.
6. C. Creveuil. “*Implementation of Gamma on the Connection Machine*”. In Proc. Workshop on Research Directions in High-Level Parallel Programming Languages, Mont-Saint Michel, 1991, Springer-Verlag, LNCS 574, 219-230, 1991.
7. Gladitz, K. and Kuchen, H. “*Shared memory implementation of the Gamma-operation*”. Journal of Symbolic Computation 21, 577-591, 1996.
8. Cabri, et al. “*Mobile-Agent Coordination Models for Internet Applications*”. Computer, 2000 February, http://dlib.computer.org/co/books/co2000/pdf/r2082.pdf. 2000.
9. Berry, G. and Boudol, G. “*The Chemical Abstract Machine*”. Theoretical Computer Science, 96, 217-248, 1992.
10. Le Metayer, D. “*Higher-order multiset processing*”. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 18, 179-200, 1994.
11. Cohen, D. and Muylaert-Filho, J. “*Introducing a calculus for higher-order multiset programming*”. In Coordination Languages and Models, LNCS, 1061, 124-141, 1996.
12. Fradet, P. and Le Metayer, D. “*Structured Gamma*”. Science of Computer Programming, 31(2-3), 263-289, 1998.
13. J.-P. Banâtre, P. Fradet and Y. Radenac. “*Chemical specification of autonomic systems*”. In Proc. of the 13th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE'04), July 2004.
14. J.-P. Banâtre, P. Fradet and Y. Radenac. “*Principles of chemical programming*”. In S. Abdennadher and C. Ringeissen (eds.): Proc. of the 5th International Workshop on Rule-Based Programming (RULE'04), 124, ENTCS, 133-147, 2005.
15. J.-P. Banâtre, P. Fradet and Y. Radenac. “*Higher-order Chemical Programming Style*”. In Proceedings of Unconventional Programming Paradigms, Springer-Verlag, LNCS, 3566, 84-98, 2005.

Appendix A – Maximum Number Finder

Flowchart

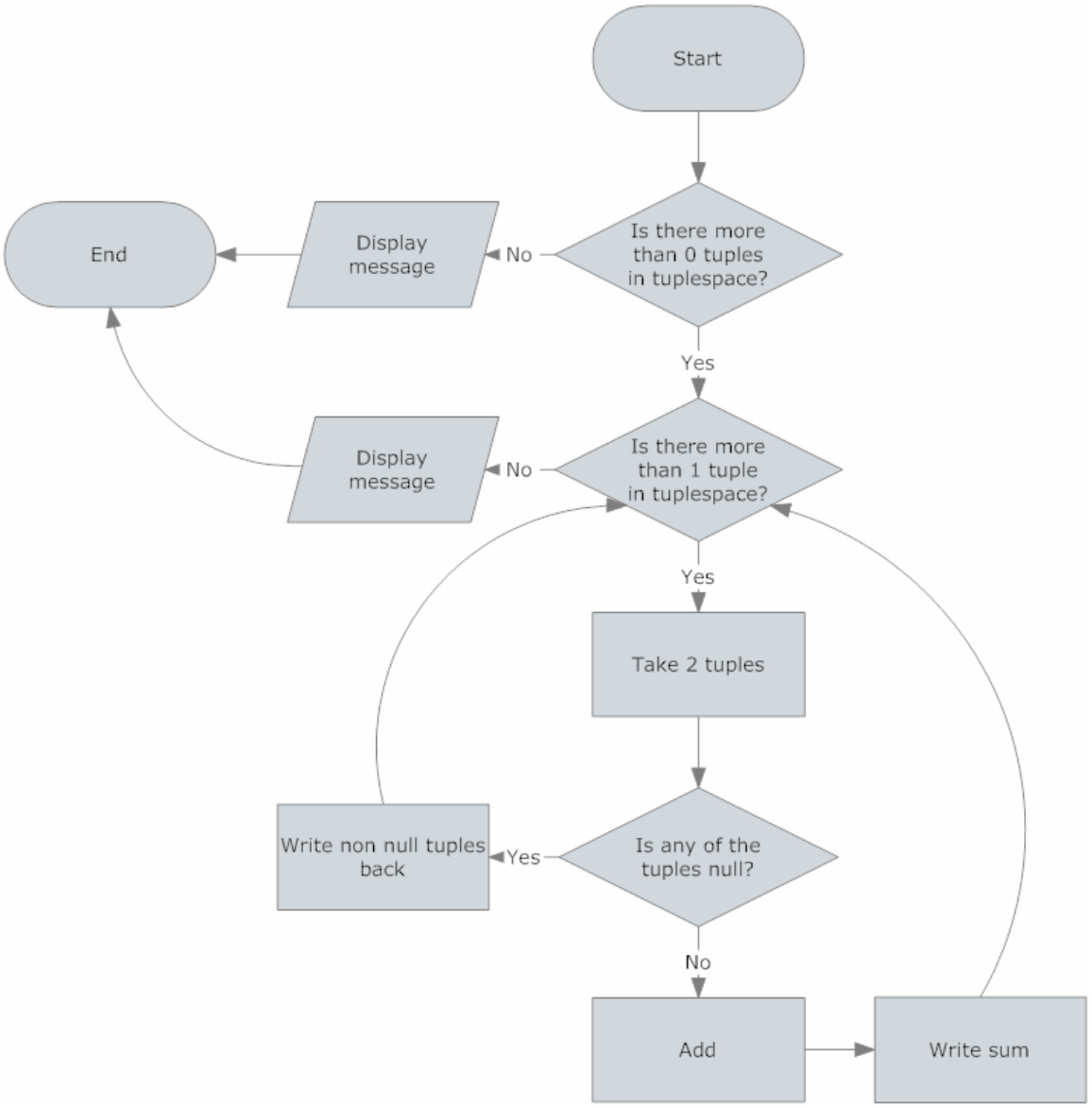


Code

```
//Wilfredo Molina – July 10, 2009

import com.ibm.tspaces.*;

public class gammaMax3
{
          public static void main(String[] args)
          {
                    try
                    {
                              String host = "grid.uhd.edu";
                              TupleSpace ts = new TupleSpace("gammaSort", host);
                              Tuple template = new Tuple(new Field(Integer.class), new Field(Double.class));

                              Tuple t1;
                              Tuple t2;

                              if ((Integer)ts.countN(template) < 1)
                                        System.out.println("TupleSpace Empty Here");
                              else
                              {
                                        while ((Integer)ts.countN(template) > 1)
                                        {
                                                  t1 = (Tuple)ts.take(template);
                                                  t2 = (Tuple)ts.take(template);

                                                  if (t1 == null || t2 == null)
                                                  {
                                                            if (t1 != null)
```

```
                                                                ts.write(t1);
                                                        if (t2 != null)
                                                                ts.write(t2);
                                                }
                                                else
                                                {
                                                        if ((Double)t1.getField(1).getValue() >
(Double)t2.getField(1).getValue())
                                                                ts.write(t1);
                                                        else
                                                                ts.write(t2);
                                                }
                                        }

                                        if ((Integer)ts.countN(template) == 1)
                                        {
                                                t1 = (Tuple)ts.read(template);
                                                System.out.println("Max Found: " +
(Double)t1.getField(1).getValue());
                                        }
                                }
                        }
                        catch (TupleSpaceException tse)
                        {
                                System.out.println("It's Broke, Wil.");
                        }
                }
        }
```

Appendix B – Tuple Adder



Appendix C – Sorter