Brock University

Department of Computer Science

Parallel Software Engineering: a Tutorial for the State of Mind

Vladimir Wojcik
Technical Report # CS-02-13
May 2002

Brock University
Department of Computer Science
St. Catharines, Ontario
Canada L2S 3A1
www.cosc.brocku.ca

PARALLEL SOFTWARE ENGINEERING: A TUTORIAL FOR THE STATE OF MIND

Vladimir Wojcik
Department of Computer Science
Brock University
St.Catharines, Ontario L2S 3A1
e-mail: vwojcik@sandcastle.cosc.brocku.ca

KEY WORDS

ABSTRACT

This is the first of two essays, introducing the reader to performance and programming issues in parallel processing. A list of benefits and difficulties associated with parallel processing is presented. Some computer architectures are compared. Only minimal experience in traditional, sequential programming on the part of the reader is required. The programming issues are illustrated with simple examples. Finally, a parallel programming methodology is recommended.

SOME PROMISES, SOME PERILS

In the last decades we have witnessed spectacular advances of computing technology. That technology made entries in almost every discipline of human endeavour. There is a vast body of literature describing these advances. This essay is not an addition to that collection, however.

There is a number of computer scientists, the current author among them, who seek ways of preserving, or even accelerating, this phenomenal growth rate. Curiously, surrounded by all these successes, they are concerned that our appetite for computing power grows even faster than the current technology can deliver.

Specifically, they are preoccupied with the idea that computing is approaching a certain impasse: we are uncomfortably close to the limits of computing power obtainable from single-processor systems. It seems that the switch to multiprocessor systems is inevitable. The parallelizing and vectorizing compilers in use now constitute the first steps in this direction, as they extract some parallelism from our sequential code. However, their use can be viewed as a mere stopgap measure, because full extraction of parallelism requires the understanding of the problem at hand. Current compilers do not understand anything. Programmers, on the other hand, do.

Any parallel program contains a number of cooperating execution threads, called *processes* or *tasks*. Having identified and implemented them, a programmer creates a concurrent application with a number of characteristics:

1. Concurrent application closely reflects the parallel structure of the problem at hand. In nature many events tend to happen at the same time. This fact is not lost on scientists, nor on designers of operating systems, real-time systems, database systems, simulation tools, etc.

2. Concurrent application facilitates proper utilization of the multiprocessor architecture. The utilization may be maximized by leaving to the programmer the allocation of tasks to the processors available. The compiler is unlikely to accomplish this feat alone, as it lacks understanding of the problem reflected in the application program. To complicate matters further, neither the programmer, nor the compiler, using the host machine, have access to the information regarding the architectures of the target computers.

3. Multiprocessing promises significant execution speedups. It is not unrealistic to expect a thousandfold increase in speed; in comparison, parallelizing and vectorizing compilers offer modest speedup factors of 10 to 30.

4. Multiprocessing offers a promise of further miniaturization of computing equipment. So far, the advances in processor technology have been largely offset by the needs for larger and larger memories. Consequently, during the past decades, the price of the turnkey computing installations continued to fluctuate about $200/lb. Instead of using large amounts of silicon to idly store vast amount of data for relatively long periods, perhaps we could use some of this silicon to introduce extra processors to access these data sooner? Tasks running on these processors could differ in priority (eager vs. just-in-time), in order to further reduce storage needs.

All this is easier said than done. We face a number of difficulties, some conceptual, some technological. Some of them are of our own making. In particular:

1. Traditional programming languages were designed with a single processor in mind. They prevent us from expressing parallelism. Worse still: they inhibit our intellectual search for parallel solutions. Granted, there are some nonstandard extensions to the traditional languages (Concurrent Pascal, Concurrent C, etc.) providing old designs with some parallel facilities. New programming tools must be designed with parallelism in mind, because nonstandard solutions affect code portability. That is why the designers of newer programming tools (like ADA, OCCAM2, etc.) included the I/O and parallel processing facilities as part of the languages.

2. Parallel programs execute more instructions than their sequential counterparts. Extra code is needed for task synchronization and communication. Only rarely such a program will execute faster on a uniprocessor. This would happen if the I/O and DMA access operations were allowed to run on a dedicated I/O processor. Given the currently dominant computer architecture, such an incentive is typically not sufficient to embark on parallel programming.

3. The programmer willing to undertake the parallel approach, faces two chicken and egg issues. Firstly, parallel programming tools must be ported to multiprocessors. Anyone who has executed extensive ADA compilations on a uniprocessor can appreciate the improvement in the programming environment that could be obtained by parallel execution of various compiler

passes on dedicated processors. After all, the rules of separate compilation allow for parallel compilation of packages in a controlled way. The same applies to linkers, editors, debuggers.

4. The second issue is application specific. It is the responsibility of the programmer to identify and implement application tasks, and then to allocate them to particular processors. The issue: There are many ways to split an application into a number of tasks. The optimal split depends both on the nature of application and on the number and characteristics of processors available. Optimal allocation is usually not known at the beginning of the project. Processor allocation tools must therefore be provided, to delay the allocation until the tasks are implemented. This would also allow some reallocations, should the number of processors or tasks change.

5. The search for optimal processor allocation is a non-trivial job, as there are P^T allocations possible of T tasks among P processors. To put this into perspective: It is not uncommon to run 50 processes on a multiprogrammed UNIX uniprocessor. With 10 processors, there are 10^{50} ways of allocating 50 tasks among them. If we were to simulate all allocation variants with each variant simulation lasting 1 second, then the simulation program would execute for exceedingly long time. For comparison: the age of the universe is approx. $4.7*10^{17}$ seconds. A parallel simulation is not possible, either: our observable universe contains approx. 10^{80} elementary particles; therefore there is scarcity of building materials, if we were to construct a computer capable of performing such parallel simulation. Consequently, only the evolutionary search methods for optimal processor allocation seem feasible, and in order to use them, the programmer's understanding of the application is essential.

6. Processor allocation may be even more complex, if fault tolerance is desired. Under this scenario, some tasks must be duplicated so that a vote could be taken when comparing the results of their computations. The redundant results, differing from the majority vote, can be helpful in identifying hardware malfunctions.

7. Processor allocation complexity skyrockets if we allow tasks to have preferences for processors of specific characteristics. After all, the processors may differ: they may have different amounts of local memory, may or may not have floating point facilities, graphics facilities, etc. Other differences may include cost, speed, word length, etc. Some processors may be suitable for specific types of computations only, like FFT, matrix manipulation, etc.

8. If the number of tasks exceeds the number of processors available, then some processes must share a single processor. Some of them still must run in quasi-parallel, in interleaving mode. Others could be run sequentially in any order. For efficiency, those to be run sequentially on a single processor should be combined into one process, devoid of the superfluous code for interleaving and bookkeeping of their volatile environments. Luckily this job is not as hard as it may initially seem: it may be automated, because the serialization of independent parallel tasks can be performed in any order. The inverse of it is much harder: exhaustive parallelization of a sequential code requires its analysis and understanding.

9. The multiprocessor architecture must be under programmer's control, so that the communication links between processors could be directly programmable. This constitutes the most fundamental departure from the established ways of computing. After all, some tasks may wish to

communicate with other (but not necessarily all) tasks, depending on the nature of application. Therefore, the computer architecture should reflect the nature of the application. Currently, we do things the other way around: given a problem at hand, we code it in a way compatible with the prevalent computer architecture. In future, a multiprocessor will likely be regarded as a pool of computational resources (processors, communication links, memories, etc.). An application will dynamically draw from these resources, perform configurations and computations, and on task completion return resources to the pool.

The last point implies that the whole concept of time-sharing has to undergo a transformation. It will continue to apply as before to tasks sharing the same processor. Traditionally, with uniprocessors, i.e. computers of a necessarily fixed architecture, it was relatively easy to allocate the whole computer in turn to a number of user processes. Now, we begin to think about allowing each user process to draw the required computational resources from a pool, and to configure them in a particular way. It appears that it might be impractical to require each process to surrender these resources to the common pool when its time slice expires, only to draw and configure them again for the duration of the next time slice. Such policy would introduce tremendous overhead; in particular, it would require that the time slices on all processors were synchronized, resulting in delays and poor equipment utilization. Perhaps it would be better to allow each process to hold the resources it needs. Initially, it may look expensive. In the longer term, this should not matter: the costs of processors continue to decrease. (A bit more worrisome are the issues of deadlock and livelock prevention, detection, and resolution).

The above two (non all-inclusive) lists enumerate the promises and perils of multiprocessing. Having compared their lengths, the reader may understandably get depressed. Fortunately, help is under way.

THIS IS NOT A VICIOUS CIRCLE

The continuation of advances of multiprocessing requires close cooperation of hardware and software communities. Most of the multiprocessing equipment is currently high priced, because it is targeted at the (relatively) well-endowed research and defense establishments. It is essential to start proliferating low-cost equipment, within reach of the owners of microcomputers and workstations. Luckily it starts happening, one of the most promising offerings being the line of the transputing equipment by INMOS Corp.

The programming community must be willing to explore parallel programming possibilities, in order to be able to tap the power of multiprocessing. Sticking with the conventional programming languages is the main obstacle. ADA (sponsored by the US Department of Defense) and OCCAM2 (by INMOS Corp.) are arguably the most popular members of the group of new languages, suitable for multiprocessing.

The compiler technology is likely to play a major role in the future. Consider a situation in which a programmer has already created a parallel application, ready to run on an installed base of parallel, but not identical, computers. As the number of processors available on these machines may vary, the application must be able to distribute itself over the number of the processors available. Similarly, should a computer be upgraded or partially damaged, resulting in the change of the number of processors available, the application must be able to dynamically re-distribute itself. This cannot be

done through a recompilation on customer site, as the application owners are unlikely to publish the source code.

Every new language defines not only new syntax, but more importantly, a new programming methodology, i.e. a culture, a paradigm. The introduction of new and better methodology constitutes the main contribution to computing. At the same time, this methodology is the hardest to learn by a programmer perusing a language reference manual.

Easy and natural presentation of this programming methodology is one of the goals of this essay. Having picked ADA as the most popular language readily suitable for multiprocessing, the author would like to describe a certain point of view at its tasking facilities. The broad lines of this point of view are readily expandable to all other languages supporting multiprocessing. At this instance the author would like to ask readers for indulgence: some examples might at first appear obviously trivial.

MULTIPROCESSING IS REALLY VERY OLD

The point of view presented here predates electronic computing. Indeed, it is millenia old. How did we use to deal with a job too big for a single human? For example, how did we build the pyramids? We used to split a big job into a number of smaller jobs, so as not to overwhelm a single human worker. Each worker was given a single task. This task consisted of a list of things to do, and of instructions to exchange/share the work results with other workers, performing related tasks. As the workers were pretty intelligent, they knew that to accomplish these exchanges they had to share some communications protocol. The overall system was pretty complex, even fault-tolerant: The pyramids were built, although many workers died.

In electronic computing we prefer to use the term 'processor' to 'worker.' Our processors are not as intelligent as our workers, and therefore must be given detailed rules for cooperation, i.e. synchronization and communication, in order to exchange or share the work results. As the work details depend on the nature of their tasks, dictated by the nature of application, they must be included in the task definition. The methods of synchronization and communication, being independent of the problem at hand, can be part of a programming language.

The described methodology is well known and well tested; indeed, we used it to build our civilization. The fact that in the first fifty years of electronic computing we chose to abandon it merely because the architecture of our early computing devices did not facilitate its use, is likely to be considered an episode rather than a trend.

TASKING: EXPRESSION OF AN OLD IDEA

A task is a part of a program defining a single execution thread, together with intertask synchronization and communication mechanisms. A computing job is split into tasks depending on its nature and on the number of processors available, much the same way a contractor would allocate duties to his workers, to promptly complete the contract.

Some tasks will unavoidably be larger than others. At this point it is convenient to introduce the ideas of a logical processor and of a physical processor. The first of them roughly corresponds to the list of skills necessary to execute a given task, the second corresponding to a single worker.

Since in a human enterprise the tasks vary in size and the number of workers usually does not equal the number of tasks, some individual workers must have the skills to perform several small, but different tasks, while some other tasks are still too large and must be assigned to a group of workers. (For example: A secretary typically uses two sets of skills: clerical skills and those of a telephone operator. Another example: Two car mechanics must cooperate to replace a large and unwieldy transmission unit). This arrangement allows for a quick reallocation of duties, should the number of workers change.

The programming analogy in ADA is full. Each task is understood to run on its own logical processor. Depending on the number of tasks and their size, a single physical processor may be used to implement a number of logical processors (using interleaving), or a number of physical processors may be needed to implement a logical processor (should a task be big and the analysis of its sequential code would indicate that further parallelism is feasible).

In this way two feats are accomplished:

1. Should the number of physical processors change, quick reallocation of tasks is possible.

2. A programmer can now be preoccupied with problem logic only (in terms of tasks and their logical processors, as the nature of application dictates) and can postpone the consideration of the target computer architecture. The issue of allocating these tasks to the physical processors available can now be delayed to the software installation stage.

A frequently wielded accusation at ADA software generation tools is that the ADA code is large and rather slow. Typically the accusation is made without understanding that these tools are required to accomplish more than the traditional ones. Regarding the code size: ADA tools are generating an all-inclusive code intended to run on a bare machine, i.e. devoid of any operating system. As to the speed: To pass a fair judgment one should run an ADA code on a multiprocessor.

Indeed, when it comes to multiprocessing, speed is not the factor of foremost importance. To appreciate that, please remember that the best computers we have are our brains. They are made out of exceedingly slow processing elements: neurons. It is the level of parallelism that is of foremost importance!

A CRASH COURSE IN PARALLEL PROGRAMMING

Software engineers may consider as blasphemous my attempt to describe the methodology of multiprocessor programming within a short essay. To placate them, a reference to a more elaborate example is indicated at the end of this paper.

In the meantime, let us define an ADA task, being an execution thread distinct from the main program. It consists of a specification and of a body. The specification lists the services (if any) a task is willing to offer to the main program or to other tasks (if any). The body contains a code implementing these services.

A specification of a task not offering any services, but doing some useful work independently, could look as follows:

```
task SELFISH;
```

Within its body calls may be inserted acquiring the services of other tasks.

Let us introduce some terminology here. The tasks offering services to other tasks are usually called 'server tasks,' those acquiring the services are called 'client tasks.' The relationship between the server tasks and the client tasks is asymmetric: client tasks must know the server tasks and the services they offer to be able to acquire them, while the server tasks need not to know any client tasks: servers just offer services to any requesting task. Incidentally, a given task may be a server and a client: it may offer some services, thus being a 'server,' but the implementation of these services may require it to request some services offered by other tasks, thus making it also a 'client,' with respect to other services.

To use an explanatory analogy: A woodworking shop may be seen as a server task offering services to some customer tasks. There may be more than one task of type 'customer' in a program, as well as more than one task of type 'woodworking shop.' Similarly, the implementation of each 'woodworking shop' may contain a number of tasks of type 'carpenter.' In this context, a woodworking shop is a client of carpenter tasks.

Generalizing the above: A task must belong to some type. There may be a number of tasks of the same type in a program. Also, there may be a number of task types defined, and a number of tasks of each type cooperating and executing concurrently with the main program.

This is just an extension of the familiar relationship between variables and their types. As we all know, in a program there may be a number of variables of a given type, each storing some data. A collection of related variables is called a data structure.

Extending this logic, tasks may be seen as computationally active entities, acting on passive data. There can be a number of tasks of a given type. A collection of related tasks is called a computational structure. Indeed, nothing stops us from declaring, say, a record, of which some elements are traditionally passive, while others are computationally active!

Let us consider some very simple tasks and their implementations.

A specification of a task type capable of storing one message may look as follows:

```
task type MAILBOX is
   entry PUT (MESSAGE :  in LETTER);
   entry GET (MESSAGE : out LETTER);
end MAILBOX;
```

The entries above define services offered by a task of type `MAILBOX`. We may create a number of tasks of this type, like so:

```
JIM, JOE : MAILBOX;
```

Having done it, in another part of the program we can start using services offered by these mailboxes, calling their entries as procedures:

```
JOE.GET (NOTICE);
JIM.PUT (WARNING);
```

Before all this is put to work, we must define the actions a task of the type `MAILBOX` must perform in order to offer the services `GET` and `PUT`. Arguably, the crudest implementation of the body of `MAILBOX` would look like this:

```
task body MAILBOX is
   SAVED : LETTER;  -- a variable to hold a single message
begin
   loop
      accept PUT (MESSAGE : in LETTER) do
         SAVED := MESSAGE;
      end PUT;
      accept GET (MESSAGE : out LETTER) do
         MESSAGE := SAVED;
      end GET;
   end loop;
end MAILBOX;
```

A word on timing: The above implementation of the task type `MAILBOX` consists of an infinite loop made of two separate steps: First the `MAILBOX` task will wait for some client task to insert a message (using the entry `PUT`), and then it will wait for the removal of the message (using the entry `GET`). After all, there is no point in attempting to remove a message before it has been put in a mailbox.

This raises a question: what happens if a client task attempts to remove a message before such a message is deposited? After all, the client tasks do not know whether the mailbox is empty or not! Similarly, other tasks intending to deposit messages may attempt to do so, not knowing that the mailbox is full. What happens then?

A given task can accept only one entry at a time. Transparently to the programmer, the ADA run time system defines a queue associated with every task entry. Tasks requesting attention of a server task which is currently 'busy' will be queued up against their entries of interest. Once the server task becomes 'idle', it will pick one service request from one of the entry queues. The choice of the entry queue depends on the implementation of the server task. If this queue has more than one pending request, the choice of the request to be services is beyond programmers control: it is indeterminate.

The language allows the programmer to set up deadlines for the clients' wait for service: with such deadline not met, the client may leave the queue and perform some alternative set of actions. In particular, with no such deadline, a client task may de facto inspect a mailbox for a message, and, finding none, perform some actions without delay.

For example, having declared:

```
SECONDS : constant DURATION := 1.0;
```

the following sequence of statements will cause a client to wait 2.5 seconds for a chance of obtaining a message from JIM's mailbox. Should such attempt prove unsuccessful within the specified time limit, the client task is going to execute some alternate code.

```
select
   JIM.GET (MEMO);
   ...   -- some optional code here
or
   delay 2.5*SECONDS;
   ...   -- insert alternate code here
end select;
```

A similar code will enable an 'impatient' client task to attempt the inspection of a relevant mailbox, and, should such an attempt be unsuccessful for any reason (mailbox empty, mailbox server otherwise busy), to execute an alternate code without delay:

```
select
   JIM.GET (MEMO);
   ...   -- some optional code here
else
   ...   -- insert alternate code here
end select;
```

From the client task point of view, a call to an entry of a server task (viz. `JIM.GET(MEMO)`) looks just like a procedure call: the execution of the client's code is suspended until the server processes the entry call (or a deadline expires).

A word on entry queueing: Many a reader may ask now: are the entry queues always helpful? Would it not be faster to allow sometimes clients to access the requested entries immediately, without delay? Yes, it would be faster, but incorrect.

To illustrate that, consider a simple job of incrementing a counter. Let a counter variable, accessible to one hundred tasks, be initially set to zero.

```
COUNTER : NATURAL := 0; -- type NATURAL covers non-negative integers
```

Suppose each of these tasks would on occasion increment the `COUNTER` like this:

```
COUNTER := COUNTER + 1;
```

What would be the value of the `COUNTER` f each task was given the chance of incrementing it once? An unsuspecting reader may be surprised to learn that the value, being indeterminate, will range anywhere from 1 to 100 (inclusive).

To explain why, consider the act of incrementing the `COUNTER` once. Due to the prevailing computer architecture, it is done in three steps. Firstly, the contents of the memory location containing the value of the `COUNTER` variable is copied to some CPU register, secondly, it is actually incremented, and thirdly, the new value is copied back to the original memory location.

Now, consider two tasks trying to increment the `COUNTER`. Suppose that, while the first task executes the first (and possibly the second) step, the second task manages to execute the first step. Both tasks have read in the value 0 and will increment it to 1. With both increments eventually done, the `COUNTER` variable will be left with the value 1, while some of us would expect it to hold a 2. Indeed, the final value of the `COUNTER` variable depends on task timing, i.e. is indeterminate. Such bugs are most difficult to detect and reproduce!

The whole problem arose from allowing two tasks access simultaneously one shared variable. Instead, one of them should be queued, until the other finishes its job.

Methods of problem decomposition: Please note that this simple example gives us very good clues on the methods of decomposing a big programming problem into a number of smaller tasks. We should do it in such a way that we avoid any shared variables, but the task should communicate (as rarely as possible, to avoid queueing) by calling each other's entries. In the particular example before, if incrementing a counter is the only thing the tasks are going to do, it would be beneficial to dispense with tasks altogether: the job of `COUNTER` incrementing has to be done serially, anyway!

If, however, incrementing the `COUNTER` is one minor and infrequent of many chores the client tasks have to do, then, to protect against risks of mutual task interference, the job of incrementing a `COUNTER` should be implemented as a task:

```
task type COUNTER is
   entry RESET;   -- used for initialization
   entry INCREMENT;
   entry READ (VALUE : out NATURAL);
end COUNTER;
```

Observe also, that this method of job decomposition is well known and tested. For example, when building a house we tend to allocate the work to a carpenter, a plumber, an electrician and a heating and A/C specialists so that they do not work in the same area of the house at the same time. Otherwise, they would interfere with each other, resulting in delays and/or mutual destruction of work. Luckily, human workers are intelligent: there is no risk that they will trample each other.

Shared variables and multiprocessor architectures: ADA allows the declaration and use of shared variables, to placate conservative programmers who find it difficult to part with the old ways. The run time system creates a transparent queue associated with every shared variable as a part of mechanism preventing task interference when accessing these variables. In short, the variables are treated as task entry calls anyway. The author would like to discourage the use of shared variables: relying on side-effects for program correctness is a violation of good software engineering practices.

Indeed, as we all know, there are two kinds of multiprocessor architectures: shared memory and distributed memory multiprocessors. It appears that the latter architecture has an edge: when using the shared memory multiprocessors we have to be specially careful to avoid problems that do not arise at all when using distributed memory multiprocessors...

Implementation of server tasks: it is the programmer's responsibility to decide whether the clients are in control of the servers, or vice versa. In the former example, the task `MAILBOX` was really in control. It was dictating that it was willing to accept a `PUT` or a `GET` request in turn. Any client task attempting to access the `MAILBOX` out of turn was made to wait.

Sometimes this approach is not convenient. For example, the `COUNTER` task should not dictate when it's value should be initialized, incremented, and read. It all depends on the client tasks, and their role in the application program. Therefore, the `COUNTER` task should be implemented as:

```
task body COUNTER is
   CLICKS : NATURAL := 0;     -- initialized to zero
begin
   loop
      select
         accept RESET;
         CLICKS := 0;
      or
         accept INCREMENT;
         CLICKS := CLICKS + 1;
      or
         accept READ (VALUE : out NATURAL) do
            VALUE := CLICKS;
         end READ;
      end select;
   end loop;
end COUNTER;
```

In this example the `COUNTER` task, not knowing which of its three services (initialization, incrementing, and reading the counter value) is to be needed next, is prepared to offer any of them at any time. Indeed, the counting variable is called `CLICKS` here, and the whole purpose of the `COUNTER` task is to increment the counting variable in a controlled way.

A less efficient implementation of this task could look as follows:

```
task body COUNTER is
   CLICKS : NATURAL := 0;     -- initialized to zero
begin
   loop
      select
         accept RESET do
            CLICKS := 0;
         end RESET;
      or
         accept INCREMENT do
            CLICKS := CLICKS + 1;
         end INCREMENT;
      or
         accept READ (VALUE : out NATURAL) do
            VALUE := CLICKS;
         end READ;
      end select;
   end loop;
end COUNTER;
```

This implementation, running on a multiprocessor, although producing the same results, would be a slower one. Why?

It has been said that from the client task point of view, a call to an entry of a server task (viz. `JIM.GET(MEMO)`) looks just like a procedure call: the execution of the client's code is suspended until the server processes the entry call. This processing terminates when the server finishes execution of the corresponding `accept` statement. It is therefore advantageous to limit the size of the code within the `accept` statement to the bare minimum, and to append the rest of the necessary code after it. In this way, while the server task processes the rest of the code, the calling task is freed to execute its own code in parallel.

Incidentally, it is common for task implementation to contain infinite loops: the servers typically do not know how many times and in what sequence their entries will be called. This, however, introduces the server halting problem.

How can the server know that it is time to stop operating? A simple solution is to have the main program to kill a server task, like this:

```
abort SELFISH, JIM, JOE;
```

Alternatively, a server task may commit suicide, possibly preceded by some cleanup chores, when it cannot be needed again. For example:

```
task body COUNTER is
   CLICKS : NATURAL := 0;     -- initialized to zero
begin
   loop
      select
         accept RESET;
         CLICKS := 0;
      or
         accept INCREMENT;
         CLICKS := CLICKS + 1;
      or
         accept READ (VALUE : out NATURAL) do
            VALUE := CLICKS;
         end READ;
      or
         terminate;
      end select;
   end loop;
end COUNTER;
```

In this example every task of type `COUNTER` is programmed to commit suicide when there are no outstanding calls at its entries and all other tasks (including the main program), capable of launching an entry call, have terminated, or are about to do so.

Sometimes the issue of control (server vs. client) is not so clear cut. For example, consider a mailbox with capacity of 100 messages. When partially full, the `MAILBOX` task should be willing to accept both `GET` and `PUT` entry calls. But, when empty, it should be accepting only `PUT` entries (insertion of messages), and queueing up all the `GET` entry calls. Similarly, when full to capacity, it should accept only `GET` entries (message removals), and continue queueing up all the `PUT` attempts. With the task specification exactly as before the typical implementation of a `MAILBOX` task behaving in this manner would look as follows:

```
task body MAILBOX is
   SAVED : LETTER;  -- a variable to hold a single message
   EMPTY : constant NATURAL := 0;
   FULL  : constant NATURAL := 100;
   CONTENTS : NATURAL := 0;
begin
   loop
      select
         when CONTENTS << FULL =>
            accept PUT (MESSAGE : in LETTER) do
               SAVED := MESSAGE;
            end PUT;
```

```
            -- code to store the SAVED message goes in here
            CONTENTS := CONTENTS + 1;
      or
         when CONTENTS > EMPTY =>
            accept GET (MESSAGE : out LETTER) do
               -- code to fetch the SAVED message goes in
here
               MESSAGE := SAVED;
            end GET;
            CONTENTS := CONTENTS - 1;
      or
         terminate;
      end select;
   end loop;
end MAILBOX;
```

Many readers, versed in the traditional, sequential programming, will recognize above the idea of a buffer. However, the `MAILBOX` buffer here is an active computational structure (a task), protecting a passive, data structure, storing the relevant information. This is so, because a "naked" data structure, accessed simultaneously by two or more client tasks, could easily get corrupted. (Think of pointers, being modified by two tasks at once). Consequently, in parallel programming, all familiar data structures, like stacks, queues, lists, trees, etc. need such a protection, in order to avoid corruption dangers. These dangers did not exist in the traditional, sequential programming.

CONCLUSIONS

In this short essay we have focused on benefits, dangers and issues of concurrency in parallel processing. With caution, in creating parallel software it is possible to borrow from sequential programming many well established concepts. One such extremely useful concept is that of a module, or package, being a collection of logically related computational resources (constants, types, variables, subprograms and tasks).

With the concept of a package in mind, the parallel programming methodology could be outlined as follows:

1. Define the problem;

2. Develop informal approach:
- identify objects and their attributes;
- identify the operations on these objects;
- identify tasks responsible for these operations
- establish server/client relationships between tasks;
- identify entries for each task;
- aggregate related objects, operations and tasks into modules (packages);

3. Formalize the approach:
    - create subprogram and package specifications:
        - select identifiers for objects and their attributes;
        - create specifications for operations on objects (procedures and task entries);

- create subprogram and package bodies:
  - implement subprograms
  - implement tasks.

We have also listed a number of promises and perils, afforded by parallel processing, as well as characterized several parallel computer architectures, in terms of their potential.

The short programming examples chosen here were intended to illustrate the nature of the parallel programming issues. Their brevity has prevented the author from illustrating some other, essential issues in parallel software engineering, like error detection and debugging of the parallel code.

To do this, a bit larger examples are needed. There exists another essay describing these issues, taking as a working example a package of anticipatory random number generators. This package runs on any computer with ADA. How is it possible to make such a package "know" *a priori* the types of the distributions needed? For more information please contact the author, preferably at

vwojcik@sandcastle.cosc.brocku.ca.