```
> restart;
```

Demonstration of a parallel computation in Maple using the Grid package.
W. Auzinger (SS 2019)

Parallelization principle:

- **Single-code multi-thread ('multi-node') computation**
- **Shared memory (uniform access)**
- **Communication via message passing**
- **Master node 0 controls overall execution**

Operates on a multi-core architecture with shared memory.

```
> with(Grid);
```

[*Barrier*, *Get*, *GetLastResult*, *Interrupt*, *Launch*, *Map*, *MyNode*, *NumNodes*, *Receive*, *Run*, *Send*, *Seq*, *Server*, *Set*, *Setup*, *Status*, *Wait*, *WaitForFirst*]

```
> Status();
```

12, 12

NumNodes() returns the number of available nodes (= threads):

```
> NumNodes();
```

12

Here we are still in non-parallel mode. Only one of the nodes, the 'master node' is active; it is has been assigned the identifier 0:

```
> MyNode();
```

0

The other available nodes have been assigned identifiers from 1 to NumNodes()-1; they are not yet active.

Now we define a procedure demonstrating that the nodes can operate in parallel.

```
> TestNodes := proc()
  local n;
  n := MyNode():
  printf("   Hello, I am node %a\n",n)
  end proc:
```

By **Launch()** we execute the procedure on our multi-core machine:

```
> Launch(TestNodes);
   Hello, I am node 0
   Hello, I am node 4
   Hello, I am node 1
   Hello, I am node 6
```

```
Hello, I am node 2
Hello, I am node 11
Hello, I am node 5
Hello, I am node 3
Hello, I am node 10
Hello, I am node 8
Hello, I am node 7
```

Each node executes the same code in parallel. They share common memory,
but each of them has its own workspace.
In particular, local variables take different values on different nodes.

In this very simple example, no communication between the nodes has taken place.

NOTE: When a procedure is executed using **Launch()**, argument passing and return values must be realized via global variables (Launch() parameters **imports, exports**).
See the following example.

Example 1:

Assume you have a very long list of numbers *x[i]* and some function *f*,
and you want to compute the sum of the *f(x[i])*.

In the following code we distribute the sum over the nodes,
using a 'modulo number of nodes' principle'.
This is efficient since all nodes have the same work load.

The speedup obtained is NumNodes()-1.

Communication (message passing) between the nodes operating in parallel
is done via **Send()** and **Receive()**.

```
> distributed_summation:=proc()
  global x,f,total_sum: # for communication with calling worksheet
  local i,iNode,imod,count,active_nodes,this_node,partial_sum:
  this_node := MyNode():
  active_nodes := NumNodes()-1:
  if this_node=0 then # message passing principle:
                      # -------------------------
                      # this is the master node 0 waiting for
                      # all other nodes to complete. It collects
                      # all partial sums computed by the other
  nodes
                      # using Receive(...) (see Send(...) below!)
                      # and accumulates the total sum.
     total_sum := 0:
     for iNode from 1 to active_nodes do
         total_sum := total_sum + Receive(iNode) # wait until Send
  (...) provides data!
     end do
  else # the other nodes compute partial sums and send them to
  master node:
```

```
    partial_sum := 0:
    for i from 1 to nops(x) do
        imod := i mod active_nodes:
        if imod=0 then imod := active_nodes end if:
        if this_node=imod then # each node does its job;
                               # the others skip this term
           partial_sum := partial_sum + f(x[i])
        end if
     end do:
     Send(0,partial_sum) # message passing principle:
                         # -------------------------
                         # send partial sum to master node 0
                         # using Send(...) (see Receive(...)
above!)
end if
end proc:
```

We test the procedure distributed_summation using the simple historical "Gauss sum":

```
> x:=[$1..100];
```

$$x := [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100]$$

```
> f:=t->t;
```

$$f := t \mapsto t$$

```
> Launch(distributed_summation,imports=["x","f"],exports=
  ["total_sum"]):
```

This gives the correct sum:

```
> total_sum;
```

$$5050$$

Example 2:

Assume you have a list of objects; for simplicity here we assume that the number of objects is smaller than the number of available nodes. Each objects undergoes some nontrivial transformation which takes a high effort (whatever this is). Again we speed up this computation by assigning each transformation to a different node.

In the following, the function T represents the desired transformation.
Master node 0 collects the results.

```
> Transform_in_parallel := proc()
  global dataset,transformed_dataset,T:
  local i,active_nodes,iNode,this_node:
  this_node := MyNode():
  active_nodes := NumNodes()-1:
  if this_node=0 then
     transformed_dataset:=[0$nops(dataset)]:
```

```
    for iNode from 1 to nops(dataset) do
        transformed_dataset[iNode] := Receive(iNode)
    end do
  else
    for i from 1 to nops(dataset) do # each nodes does its job
        if this_node=i then
           Send(0,T(dataset[i]))
        end if:
    end do:
  end if
  end proc:
```

A simple test:

```
> dataset:=[$1..3];
```

$$dataset := [1, 2, 3]$$

```
> T:=proc(t) return t^2 end proc;
```

$$T := \mathbf{proc}(t)\ \mathbf{return}\ t\text{^}2\ \mathbf{end\ proc}$$

```
> Launch(Transform_in_parallel,imports=["dataset","T"],exports=
  ["transformed_dataset"]):
> transformed_dataset;
```

$$[1, 4, 9]$$
