Australian National University

PIPELINES

Week 12 Laboratory for Systems, Networks and Concurrency

Uwe R. Zimmer

Pre-Laboratory Checklist

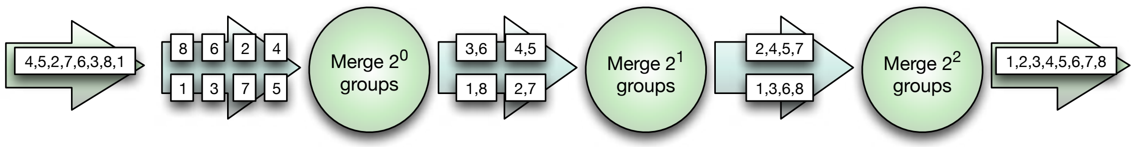
- ❑ **You have read this text before you come to your lab session.**
- ❑ **You understand and can utilize message passing locally and distributed.**
- ❑ **You have a firm understanding of memory based synchronization.**
- ❑ **You understand and can apply implicit concurrency.**
- ❑ **You can create and control tasks.**
- ❑ **You have basic skill to distribute computations.**

Objectives

This lab will confront you with a more real-world challenge than the previous labs: You need to design a concurrent implementation without hints as to what specific techniques would lend themselves to the job at hand.

You can also use this lab to patch all the gaps and questions which you may still have about previous labs. This is your last lab and so you should take this opportunity to get as much out of your tutor and lab time as possible.

Interlude: Pipelined Mergesort



Mergesort as you know from its sequential version sorts lists by merging smaller, sorted lists (starting from single element lists, which are by definition sorted) into larger, sorted lists until all elements have been merged into a single, sorted list. This algorithm lends itself to concurrent implementations in multiple ways. One way would be to set up a pipeline of merging stages, where each stage i merges two sorted lists of length 2^i into one sorted list of length 2^{i+1}.

This description of the algorithm is purposefully kept informal as it will be your job to translate this idea into a working solution.

Assuming that you have $\lceil \log_2 n \rceil$ computing nodes available to sort an n element list, what would be the time complexity for this algorithm (assuming that all computational nodes are running in parallel)? What would be the total computational complexity (by adding up the computational complexities in each node)?

Exercise 1: Implement Pipelined Mergesort

In contrast to previous labs which exercised specific methods, you have now the full freedom of choice to design an implementation of this algorithm which is smooth, elegant and efficient.

To enable you to focus fully on the concurrent aspects, I provide some sequential code fragments for you, which you might need to program the overall framework and the sequential parts of each stage.

This package beginning will provide you with flexible data-structures for a lists of generic elements (without making it an actual generic package at this point) and a random element list which you can feed into your pipeline:

```
with Ada.Command_Line;                   use Ada.Command_Line;
with Ada.Containers.Vectors;             use Ada.Containers;
with Ada.Exceptions;                     use Ada.Exceptions;
with Ada.Numerics.Discrete_Random;       use Ada.Numerics;
with Ada.Numerics.Elementary_Functions;  use Ada.Numerics.Elementary_Functions;
with Ada.Task_Identification;            use Ada.Task_Identification;
with Ada.Text_IO;                        use Ada.Text_IO;

procedure Pipelined_Mergesort is

   No_Of_Elements : constant Positive := Positive'Value (Argument (1));

   subtype Element is Natural;

   package Random_Elements is new Discrete_Random (Result_Subtype => Element);
   use Random_Elements;

   Random_Generator : Generator;

   type Index is new Natural;

   type Element_Array is array (Index range <>) of Element;

begin
   Reset (Random_Generator);

   declare
      Data : constant Element_Array (1 .. Index (No_Of_Elements)) :=
                                        (others => Random (Random_Generator));
```

Then some testing routines which could help you to check whether your algorithm worked:

```
function Is_Sorted (D : Element_Array) return Boolean is
   (for all i in D'First .. D'Last - 1 => D (i) <= D (i + 1));

function Is_Permutation (Field_A, Field_B : Element_Array) return Boolean is

   package Elem_Vectors is new Vectors (Positive, Element); use Elem_Vectors;
   package Sorting      is new Generic_Sorting;             use Sorting;

   Vector_A, Vector_B : Vector := Empty_Vector;

begin
   for A of Field_A loop
      Append (Vector_A, A);
   end loop;
   for B of Field_B loop
      Append (Vector_B, B);
   end loop;
   Sort (Vector_A); Sort (Vector_B);
   return Vector_A = Vector_B;
end Is_Permutation;
```

The number of pipeline stages derives from the number of elements in your list as such:

```
No_Of_Stages : constant Positive :=
   Positive (Float'Ceiling (Log (Float (No_Of_Elements), 2.0)));
```

Inside each stage you will need to merge existing lists. You could chose functional style:

```
function Merge (A, B : Element_Array) return Element_Array is

  (if    A'Length = 0 then B
   elsif B'Length = 0 then A
   elsif A (A'First) < B (B'First)
   then A (A'First) & Merge (A (Index'Succ (A'First) .. A'Last), B)
   else B (B'First) & Merge (A, B (Index'Succ (B'First) .. B'Last)))

with
  Pre  => Is_Sorted (A) and then Is_Sorted (B),
  Post => Is_Sorted (Merge'Result) and then Is_Permutation (Merge'Result, A & B);
```

Or imperative style:

```
function Merge_Imperative (A, B : Element_Array) return Element_Array

with Pre  => Is_Sorted (A) and then Is_Sorted (B),
     Post => Is_Sorted (Merge_Imperative'Result) and then
                                 Is_Permutation (Merge_Imperative'Result, A & B);

function Merge_Imperative (A, B : Element_Array) return Element_Array is

begin
   if A'Length = 0 then
      return B;
   elsif B'Length = 0 then
      return A;
   else
      declare
         Merged : Element_Array (A'First .. A'Last + B'Length);

         A_Index : Index range A'Range := A'First;
         B_Index : Index range B'Range := B'First;

      begin
         for M_Index in Merged'Range loop

            declare
               Merge_Element : Element       renames Merged (M_Index);
               Merge_Tail    : Element_Array renames
                                     Merged (Index'Succ (M_Index) .. Merged'Last);

               A_Element : constant Element := A (A_Index);
               B_Element : constant Element := B (B_Index);

            begin
               if A_Element < B_Element then
                  Merge_Element := A_Element;
                  if A_Index = A'Last then
                     Merge_Tail := B (B_Index .. B'Last); return Merged;
                  else
                     A_Index := Index'Succ (A_Index);
                  end if;
               else
                  Merge_Element := B_Element;
                  if B_Index = B'Last then
                     Merge_Tail := A (A_Index .. A'Last); return Merged;
                  else
                     B_Index := Index'Succ (B_Index);
                  end if;
               end if;
            end;
         end loop;
```

```
            raise Program_Error with “Merge for-loop should never complete”;
         end;
      end if;
   end Merge_Imperative;
```

After you chose one of the merge implementations, take a mental note what the reasons were for your choice. Did computational complexity, performance, maintainability, memory usage or any other reasons play a role and what was the major reason in the end?

From this point on (while you may of course as well chose to ignore the above framework) the stages are all yours and you need to come up with a water-tight, maintainable and elegant implementation. You may already have a clear idea in your head at this time, but it might still be valuable to reflect about multiple options before you starting hacking away. To give you an idea about the solution space: a neat and clean solution does not need to be any longer than around 30-40 (non empty) lines of code inside a single pipeline stage. So if you find yourself beyond 100 lines of code, you may want to step back and reconsider your design. On the other hand, your design could be specifically performant or elegant and therefore justify a larger code section? You have to make your educated choices to achieve the goal.

To start with, you may assume that the number of elements to be sorted is a power of two[1] – which will fill all your pipeline stages and you do not need to consider edge cases of partly filled pipelines. A more robust solution will need to include the general case of arbitrary length lists.

The number of elements to sort is taken from the command line, such that you call your function for instance with:

```
./pipelined_mergesort 1000
```

Submit a zip archive of your completed project to the *SubmissionApp* under “Lab 12 Pipelined Mergesort” for code review by your peers and us.

Exercise 2: Distributed Pipelined Mergesort

Now that your solution works on one node: does your algorithm also smoothly translate into a distributed implementation? Use the BSD socket techniques from the previous lab to distribute your stages over multiple computers. Can you set up a sorting machine which spans the whole of your lab?

This is for highly advanced students only (you probably see the really dark red in the headline) and I expect only very few students to get this far in your labs.

Submit a zip archive of your completed project to the *SubmissionApp* under “Lab 12 Distributed Pipelined Mergesort” for code review by your peers and us.

**MAKE SURE YOU LOGOUT
TO TERMINATE YOUR SESSION!**

Outlook

You are close to the end of an intense course and you accumulated a large set of knowledge and skills at this point. Your final job is to set things into context and to add all the relations between the different concepts in your mind which you may have overlooked during the semester.

1 While it is folklore in Engineering-land that all data is Gaussian-distributed and differential equations exist, it is folklore in Computer-Science-land that all data is reproducible, uniformly-distributed and of length 2^n.