Parallel enumeration of word overlap correlation classes using MPI

Paul C. Leopardi *

Abstract

I now have an MPI version of the word-overlap-correlation program running on Orac. At the moment, each job runs on 8 cores, as 6 "worker" processes and 2 "scribe" processes. The workers enumerate Restricted Growth Strings and transform these into correlation classes for pairs of words. The scribe processes record and count the correlation classes. This note describes some aspects of the organization of the code and how it uses MPI.

Splitting the sequence of beta-restricted growth strings into subsequences

The program `word-over-corr-mpi-worker-scribe` enumerates a sequence of beta-RGS (restricted growth strings for set partitions into beta parts) in parallel, by splitting it into subsequences. Each subsequence is defined by two beta-RGS, which are instantiated by `ulong` arrays, called `rgs_begin` and `rgs_end`, in the functions `check_setpart_p_rgs_range()` and `main()`. Each subsequence begins with `rgs_begin`, which is used to instantiate an object of class `setpart_p_rgs_lex` via the following call:

```
setpart_p_rgs_lex p = setpart_p_rgs_lex(2*T,beta,rgs_begin);
```

The function `check_setpart_p_rgs_range()` contains a loop with the structure:

```
  for (bool more=true; more; more=p.next())
  { ...
```

*Centre for Mathematics and its Applications, Australian National University. `mailto:paul.leopardi@anu.edu.au`

```
    const ulong* rgs_data = p.data();
    if (arrays_are_equal(2*T, rgs_data, rgs_end))
      break;
    ...
  }
```

Thus the subsequence begins with `rgs_begin`, and continues via `p.next()`, up to but not including `rgs_end`. A special sentinel value is used for `rgs_end` in the case of the final subsequence.

The program currently uses two methods to create the beta-RGS `ulong` arrays.

The first method is via the function `new_setpart_p_rgs_array()` as follows:

```
ulong* new_setpart_p_rgs_array(const ulong beta,
                               const ulong gamma)
{
  ulong* s = new ulong[2*T]; ...
  for (ulong k = 0; k != 2*T; k++)
    s[k] = 0UL;
  if (gamma > beta)
    s[0] = ~0UL;
  else
  {
    for (ulong k=0; k != gamma; k++)
      s[k] = k;
    for (ulong k=0; gamma + k < beta; k++)
      s[2*T-beta+gamma+k] = gamma+k;
    ...
  }
  return s;
}
```

This routine can create `beta+1` different beta-RGS arrays, depending on the value of `gamma`.

The second method is to call

```
setpart_p_rgs_lex q = setpart_p_rgs_lex(delta,beta);
```

with `delta` $\geqslant$ `beta`, continue via `q.next()`, and extend the array by appending zeros. In the function `main()`, this looks like:

```
ulong delta = (beta+WOC_DELTA_OFFSET > 2*T)
            ? beta : beta+WOC_DELTA_OFFSET;
setpart_p_rgs_lex r = setpart_p_rgs_lex(2*T,beta);
const ulong* r_data = r.data();
ulong* rgs_begin = new ulong[2*T]; ...
for (int i = 0; i != 2*T; i++)
  rgs_begin[i] = r_data[i];
ulong* rgs_end = new ulong[2*T]; ...
for (int i = delta; i < 2*T; i++)
  rgs_end[i] = 0;
setpart_p_rgs_lex q = setpart_p_rgs_lex(delta,beta); ...
for (bool more = true; more; more = q.next(), ...)
{
  const ulong* q_data = q.data();
  for (int i = 0; i != delta; i++)
    rgs_end[i] = q_data[i];
  // The subsequence [rgs_begin,rgs_end)
  // has now been defined.
  for (int i = 0; i != 2*T; i++)
    rgs_begin[i] = rgs_end[i];
}
rgs_end[0] = ~0UL;
// The final subsequence [rgs_begin,rgs_end)
//  has now been defined.
```

I call this second method a double-loop method. The outer loop enumerates beta-RGS of length `delta`, where `beta` $\leqslant$ `delta` $\leqslant$ `2*T`. The inner loop enumerates beta-RGS of length `2*T`.

Acknowledgements. Thanks to Joerg Arndt for the FXT library and many useful discussions.