Kristoffer Osowski | Jan Westerholm | Mats Aspnäs

Two Cases of Data Overflow in the Protein Sequencing Program BLASTPGP

Turku Centre *for* Computer Science

TUCS Technical Report
No 813, January 2007

Two Cases of Data Overflow in the Protein Sequencing Program BLASTPGP

Kristoffer Osowski
Jan Westerholm
Mats Aspnäs

Åbo Akademi University
Department of Information Technologies
Joukohainengatan 3–5
20520 Åbo, Finland
[kristoffer.osowski | jan.westerholm | mats.aspnas]@abo.fi

TUCS Technical Report

No 813, January 2007

Abstract

For some input sequences, the NCBI BLASTPGP program for gapped basic local alignment search fails, because of errors in the program implementation. One error causes the blastpgp program to crash for certain input sequences, failing to produce any valid results. An other error causes a large number of warning messages "*ObjMgrNextAvailEntityID failed with idx 2048*" to be output, and the program produces incorrect and incomplete search results. This report describes our investigation of the causes of these errors and a correction to both errors.

Keywords: BLAST, BLASTPGP

TUCS Laboratory
High Performance Computing Laboratory

1 Introduction

The High Performance Computing laboratory at Åbo Akademi University is since the year 2005 involved in a three-year project together with CSC, the Finnish IT center for science[1] with the goal of improving the performance of some of the codes that are used on CSC:s supercomputer systems. Within this project, a number of different computer programs in various disciplines of science, like fluid dynamics, plasma fusion, modeling of nuclear cascades, density functional theory and protein sequence matching, have been investigated and optimized for performance.

The BLAST and BLASTPGP [1, 2] programs are extensively used in bioinformatics research all over the world to carry out gapped basic local alignment search of nucleotide or protein sequences. In the autumn 2006, BLASTPGP was selected as one of the programs to be investigated and possibly performance optimized, since it is one of the heaviest used codes on CSC:s supercomputers, and its use is also steadily increasing.

The users of the BLAST software reported that the program fails to produce valid output for some input sequences, crashing the program in some cases and producing large amounts of warning messages in other cases. Before any code optimization efforts were started, it was decided that the bugs first should be corrected. This report describes the causes of these two bugs, and presents solutions to remove them.

This work applies to version 2.2.15 of the BLAST software from NCBI and the `swissprot` database, downloaded on the 17:*th* of November 2006. The BLAST software is available from `http://www.ncbi.nlm.nih.gov/blast`.

[1] `http://www.csc.fi`

2 The segmentation fault bug

For some query sequences, the blastpgp program crashes and reports a segmentation fault. An example of such a query sequence is:

```
>779261
MKVAVNTFLLFLCSTSSIYAAFALNSDGAALLSLTRHWTSIPSDITQSWNASDSTPCSWL
GVECDRRQFVDTLNLSSYGISGEFGPEISHLKHLKKVVLSGNGFFGSIPSQLGNCSLLEH
IDLSSNSFTGNIPDTLGALQNLRNLSLFFNSLIGPFPESLLSIPHLETVYFTGNGLNGSI
PSNIGNMSELTTLWLDDNQFSGPVPSSLGNITTLQELYLNDNNLVGTLPVTLNNLENLVY
LDVRNNSLVGAIPLDFVSCKQIDTISLSNNQFTGGLPPGLGNCTSLREFGAFSCALSGPI
PSCFGQLTKLDTLYLAGNHFSGRIPPELGKCKSMIDLQLQQNQLEGEIPGELGMLSQLQY
LHLYTNNLSGEVPLSIWKIQSLQSLQLYQNNLSGELPVDMTELKQLVSLALYENHFTGVI
PQDLGANSSLEVLDLTRNMFTGHIPPNLCSQKKLKRLLLGYNYLEGSVPSDLGGCSTLER
LILEENNLRGGLPDFVEKQNLLFFDLSGNNFTGPIPPSLGNLKNVTAIYLSSNQLSGSIP
PELGSLVKLEHLNLSHNILKGILPSELSNCHKLSELDASHNLLNGSIPSTLGSLTELTKL
SLGENSFSGGIPTSLFQSNKLLNLQLGGNLLAGDIPPVGALQALRSLNLSSNKLNGQLPI
DLGKLKMLEELDVSHNNLSGTLRVLSTIQSLTFINISHNLFSGPVPPSLTKFLNSSPTSF
SGNSDLCINCPADGLACPESSILRPCNMQSNTGKGGLSTLGIAMIVLGALLFIICLFLFS
AFLFLHCKKSVQEIAISAQEGDGSLLNKVLEATENLNDKYVIGKGAHGTIYKATLSPDKV
YAVKKLVFTGIKNGSVSMVREIETIGKVRHRNLIKLEEFWLRKEYGLILYTYMENGSLHD
ILHETNPPKPLDWSTRHNIAVGTAHGLAYLHFDCDPAIVHRDIKPMNILLDSDLEPHISD
FGIAKLLDQSATSIPSNTVQGTIGYMAPENAFTTVKSRESDVYSYGVVLLELITRKKALD
PSFNGETDIVGWVRSVWTQTGEIQKIVDPSLLDELIDSSVMEQVTEALSLALRCAEKEVD
KRPTMRDVVKQLTRWSIRSYSSSVRNKSK
```

If this sequence is stored for instance in a file named segfault.chunk and the blastpgp program is executed for instance with the command

```
blastpgp -j 10 -e 1 -h 0.001 -a 1 -b 100000 -v 100000 -F F
    -i segfault.chunk -d swissprot -o SegfaultResult.txt
```

a segmentation fault will occur and the program execution is aborted. The reason the program crashes is an index that exceeds the boundary of an array. The crash occurs in `posit.c:posDemographics()` (`posit.c` version 6.80) in the following lines:

```
//SOME CODE
 if (!posSearch->posDescMatrix[seqIndex+1][qplace].used)
  {
  posSearch->posDescMatrix[seqIndex + 1][qplace].used = TRUE;
  posSearch->posDescMatrix[seqIndex + 1][qplace].letter = GAP_CHAR;
  posSearch->posDescMatrix[seqIndex + 1][qplace].e_value = 1.0;
  }
 }
}
else { /*no gap*/
  for(c = 0, qplace=queryOffset, splace=subjectOffset;
      c < matchLength;c++, qplace++, splace++)
  {
  if (!posSearch->posDescMatrix[seqIndex+1][qplace].used) {
      posSearch->posDescMatrix[seqIndex+1][qplace].letter = (Int1)s[splace];
      posSearch->posDescMatrix[seqIndex+1][qplace].used = TRUE;
      posSearch->posDescMatrix[seqIndex+1][qplace].e_value= thisEvalue;
      }
  }
}
//SOME CODE
```

The program crashes because the value of the index variable `seqIndex+1` exceeds the memory area allocated for `PosDesc **posSearch->posDescMatrix`. We first investigate how and where memory is allocated for `posSearch->posDescMatrix`. It all starts in `posit.c:CposComputation()`:

```
Int4 numalign, numseq; /*nr of alignments and matches in previous round*/
  numalign = countSeqAligns(listOfSeqAligns, &numseq, FALSE, 0.0);
  posAllocateMemory(posSearch, compactSearch->alphabetSize,
                    compactSearch->qlength, numseq);
  if (!patternSearchStart)
      findThreshSequences(posSearch, search, listOfSeqAligns, numalign,
                          numseq);
  posDemographics(posSearch, compactSearch, listOfSeqAligns);
//SOME CODE
```

The memory for `posSearch->posDescMatrix` is allocated by the function `posit.c:posAllocateMemory()`, where the value of the variable `numseq` defines the size of `posSearch->posDescMatrix`. The variable also defines the size of other structures in `posAllocateMemory()`: `posSearch->posDescMatrixLength`, `posSearch->posA`, and `posSearch->posRowSigma`.

The value of `numseq` is computed by the function `posit.c:countSeqAligns()`, where `numseq` gets the value of `numSequences`, in the following way:

```
static Int4 countSeqAligns(SeqAlignPtr listOfSeqAligns, Int4 * numSequences,
                           Boolean useThreshold,Nlm_FloatHi threshold)
{
  SeqAlignPtr curSeqAlign, prevSeqAlign;
  Int4 seqAlignCounter;
  DenseSegPtr curSegs;
  SeqIdPtr curId, prevId;  /* Ids of target sequences in current and */
  seqAlignCounter = 0;     /* previous SeqAlign */
  *numSequences = 0;
  curSeqAlign = listOfSeqAligns;
  prevSeqAlign = NULL;
  while (NULL != curSeqAlign) {
    curSegs = (DenseSegPtr) curSeqAlign->segs;
    if(curSegs->ids == NULL)
       break;
    curId = curSegs->ids->next;
    seqAlignCounter++;
    if ((NULL == prevSeqAlign) || (!(SeqIdMatch(curId, prevId))))
       if (!useThreshold || (threshold >
                          minEvalueForSequence(curSeqAlign, listOfSeqAligns)))
            (*numSequences)++;
    prevSeqAlign = curSeqAlign;
    prevId = curId;
    curSeqAlign = curSeqAlign->next;
  }
  return(seqAlignCounter);
}
```

The basic question is: Why is the value of `numseq` less than the value of `seqIndex`? To answer this question we must look in `posDemographics()` and see how the variable `seqIndex` is computed:

```
//SOME CODE
numSeqAligns = countSeqAligns(listOfSeqAligns, &numseq,
                              !compactSearch->use_best_align,
                              compactSearch->ethresh);
posSearch->posNumSequences = numseq;
/*use only those sequences below e-value threshold*/
seqIndex = 0;
curSeqAlign = listOfSeqAligns;
prevSeqAlign = NULL;
for(curSeqAlign = listOfSeqAligns; curSeqAlign != NULL;
    curSeqAlign = curSeqAlign->next) {
  is_new_id = FALSE;
  thisEvalue = getEvalueFromSeqAlign(curSeqAlign);
  curSegs = (DenseSegPtr) curSeqAlign->segs;
  if (NULL != prevSeqAlign) {
      prevSegs = (DenseSegPtr) prevSeqAlign->segs;
      if(curSegs->ids == NULL)
         break;
      curId = curSegs->ids->next;
      prevId = prevSegs->ids->next;
      if (!(SeqIdMatch(curId, prevId))) is_new_id = TRUE;
  }
  if(!(compactSearch->use_best_align && is_new_id)) {
    if (thisEvalue >= compactSearch->ethresh)
      continue;
  }
  if(is_new_id == TRUE) seqIndex++;
  s = GetSequenceWithDenseSeg(curSegs, FALSE, &retrievalOffset,
                              &subjectLength);
  if (s == NULL) {
     //SOME CODE
     continue;
  }
  // SOME CODE
  prevSeqAlign = curSeqAlign;
  s = MemFree(s);
} /*closes the for loop over seqAligns*/
//SOME CODE
```

We note that `seqIndex` is computed in almost the same way as `numSequences` in `countSeqAligns()`. The significant differences are the two if-statements in `posDemographics()`:

```
    if(!(compactSearch->use_best_align && is_new_id)) {
       if (thisEvalue >= compactSearch->ethresh)
           continue;
    }

    if (s == NULL) {
        //SOME CODE
        continue;
    }
```

If the conditions are met for any of the two if-statements, a `continue` statement is executed and the assignment `prevSeqAlign = curSeqAlign` will not be executed for that iteration. This means that `prevSeqAlign` will not be updated, while in `countSeqAligns()` the pointer `prevSeqAlign` will be updated for every iteration. The result is that the `SeqIdMatch(curId, prevId)` function will operate on different Id inputs, and thus the values of `seqIndex` and `numSequences` can differ.

Here is a short sequence of Ids, given by their addresses, matched by `SeqIdMatch(curId, prevId)`:

	countSeqAligns():			posDemographics():	
Num.	CurId	PrevId		CurId	PrevId
50	11631680	11632464		11631680	11632464
51	11630864	11631680		11630864	11631680
52	11630048	11630864		11630048	11630864
53	11629248	11630048		11629248	11630048
54	11628448	11629248		11628448	11629248
55	11574080	11628448	<------>	11574080	11629248
56	11587696	11574080	<------>	11587696	11629248
57	11605600	11587696	<------>	11605600	11629248
58	11609296	11605600	<------>	11609296	11629248
59	11610640	11609296	<------>	11610640	11629248
60	11609456	11610640	<------>	11609456	11629248
61	11623568	11609456		11623568	11609456
62	11622496	11623568		11622496	11623568
63	11621728	11622496		11621728	11622496
64	11620912	11621728		11620912	11621728

Notice the differences starting at number 55.

In the current implementation it is assumed that the value of `numseq` will always be greater than the value of `seqIndex`, and that therefore enough memory will be allocated for `posSearch-> posDescMatrix`. This is not necessarily true because the values are calculated in two different ways and it's not always the case, like for the given query sequence, that `numseq` will have a greater value.

The solution to this problem is to make sure that enough memory will be allocated for `posSearch->posDescMatrix` for all query sequences. This can be implemented in several ways. It's important that the implementation doesn't reduce the readability of the code, complicate the program, or in any way affect the result, apart from eliminating the segmentation fault.

First we must have a look at how the function `posit.c:CposComputation()` is implemented:

```
Int4Ptr * LIBCALL CposComputation(posSearchItems *posSearch,
BlastSearchBlkPtr search, compactSearchItems * compactSearch,
SeqAlignPtr listOfSeqAligns, Char *ckptFileName,
Boolean patternSearchStart, Int4 scorematOutput, Bioseq *query_bsp,
Int4 gap_open, Int4 gap_extend, ValNodePtr * error_return,
Nlm_FloatHi weightExponent)
{
    /*number of alignments and matches in previous round*/
    Int4 numalign, numseq;
    search->posConverged = FALSE;
    numalign = countSeqAligns(listOfSeqAligns,&numseq,FALSE,0.0);
    posAllocateMemory(posSearch, compactSearch->alphabetSize,
                      compactSearch->qlength, numseq);
    if (!patternSearchStart)
        findThreshSequences(posSearch, search, listOfSeqAligns,
                            numalign, numseq);
    posDemographics(posSearch, compactSearch, listOfSeqAligns);
    posPurgeMatches(posSearch, compactSearch);
    posComputeExtents(posSearch, compactSearch);
    posComputeSequenceWeights(posSearch, compactSearch,
                              weightExponent);
    posCheckWeights(posSearch, compactSearch);
    posSearch->posFreqs = posComputePseudoFreqs(posSearch,
                                                compactSearch, TRUE);
    if (NULL == search->sbp->posFreqs)
      search->sbp->posFreqs=allocatePosFreqs(compactSearch->qlength,
                                             compactSearch->alphabetSize);
    copyPosFreqs(posSearch->posFreqs,search->sbp->posFreqs,
                 compactSearch->qlength, compactSearch->alphabetSize);
    if (NULL != ckptFileName) {
      if (scorematOutput == NO_SCOREMAT_IO)
        posTakeCheckpoint(posSearch, compactSearch,
                          ckptFileName, error_return);
      else
        posTakeScoremat(posSearch, compactSearch, ckptFileName,
                        scorematOutput, query_bsp, gap_open,
                        gap_extend, error_return);
    }
    posFreqsToMatrix(posSearch,compactSearch);
    posScaling(posSearch, compactSearch);
    return posSearch->posMatrix;
}
```

The variable numseq is used in `posAllocateMemory()` and `findThreshSequences()`, both functions located in `posit.c`. From earlier we know that `numseq` defines how much memory will be allocated for several data structures in `posAllocateMemory()`. It also defines the size of another data structure in `findThreshSequences()`, named `posSearch->posResultSequences[]`.

A solution is to make a new function that computes the value of `numseq` in the same way as `seqIndex` and then update the value of `numseq` if the new value is higher that the value computed in `countSeqAligns()`

The implementation of the new function looks like this:

```
static void countNumSeq(posSearchItems *posSearch,
compactSearchItems * compactSearch,SeqAlignPtr listOfSeqAligns,
 Int4 *prevNumSeq)
{
   Uint1Ptr s; /*pointer into a matching string */
   Int4  subjectLength;  /*length of subject*/
   Int4  retrievalOffset;   /*retrieval offset */
   /*pointers into listOfSeqAligns*/
   SeqAlignPtr curSeqAlign, prevSeqAlign;
   /*used to extract alignments from curSeqAlign*/
   DenseSegPtr curSegs, prevSegs;
   /*Used to compare sequences that come from different SeqAligns*/
   SeqIdPtr curId, prevId;
   Nlm_FloatHi thisEvalue;  /*evalue of current partial alignment*/
   Int4 newNumSeq; /* numseq computed in another way */
   Boolean is_new_id = FALSE;
   newNumSeq=0;
   /*use only those sequences below e-value threshold*/
   curSeqAlign = listOfSeqAligns;
   prevSeqAlign = NULL;
   for(curSeqAlign = listOfSeqAligns; curSeqAlign != NULL;
       curSeqAlign = curSeqAlign->next) {
       is_new_id = FALSE;
       thisEvalue = getEvalueFromSeqAlign(curSeqAlign);
       curSegs = (DenseSegPtr) curSeqAlign->segs;
       if (NULL != prevSeqAlign) {
           prevSegs = (DenseSegPtr) prevSeqAlign->segs;
           if(curSegs->ids == NULL)
               break;
           curId = curSegs->ids->next;
           prevId = prevSegs->ids->next;

           if (!(SeqIdMatch(curId, prevId)))
               is_new_id = TRUE;
       }
       if(!(compactSearch->use_best_align && is_new_id)) {
           if (thisEvalue >= compactSearch->ethresh)
               continue;
       }
       if(is_new_id == TRUE)
           newNumSeq++;

       s = GetSequenceWithDenseSeg(curSegs, FALSE,
                                   &retrievalOffset, &subjectLength);
```

```
        SeqMgrFreeCache();
        if ( s == NULL)
        {
            continue;
        }
        s = MemFree(s);
        prevSeqAlign = curSeqAlign;
    }
    newNumSeq++;
    /* numseq gets the highest number computed by both methods */
    if (newNumSeq > *prevNumSeq)
      *prevNumSeq = newNumSeq;
}
```

The new function, `countNumSeq()` in `posit.c`, takes the same arguments as `posDemographics()` plus the pointer to the previously computed `numseq`. If the new value is higher than the previous value of `numseq`, then `numseq` is updated with the new value.

The new function is called from `CposComputation()` between the calls of `countSeqAligns()` and `posAllocateMemory()`:

```
    numalign = countSeqAligns(listOfSeqAligns, &numseq, FALSE, 0.0);
    countNumSeq(posSearch, compactSearch, listOfSeqAligns, &numseq);
    posAllocateMemory(posSearch, compactSearch->alphabetSize,
        compactSearch->qlength, numseq);
```

A call should also be inserted in WposComputation() between the calls of countSeqAligns() and posAllocateMemory():

```
        numSeqAligns = countSeqAligns(listOfSeqAligns, &numseq, FALSE, 0.0);
        countNumSeq(posSearch, compactSearch, listOfSeqAligns, &numseq);
        posAllocateMemory(posSearch, alphabetSize, qlength, numseq);
```

These are the changes needed for the implementation of the bug fix. Here is a short summary:

posit.c:

- created a new function `countNumSeq()`
- added a call to `countNumSeq()` in `CposComputation()`
- added a call to `countNumSeq()` in `WposComputation()`

The following is the result of a test run for the query sequence that previously resulted in a segmentation fault:

```
Iter.    numseq      numseq     numseq
         method1     method2    final
2         2436        2075       2436
3         3150        3278       3278
4         3179        3342       3342
5         3157        3301       3301
6         3153        3262       3262
7         3157        3263       3263
8         3148        3253       3253
9         3157        3255       3255
10        3164        3257       3257
```

The first column indicates which iteration is being run and the following ones indicate the value of `numseq`: computed in `countSeqAligns()`, computed in `countNumSeq()`, used in `posAllocateMemory()`.

We can see that the value of `numseq` computed in `countSeqAligns()` was updated by the value computed in the new function `countNumSeq()` in 8 out of 9 iterations. As a result of the update of `numseq` enough memory was allocated, preventing the crash of the program.

The implemented solution removed the problem of an index exceeding the boundary of an array, making the program work without crashing. The results for any query sequence are exactly the same as before and the execution speed and memory usage aren't noticeably affected. The solution does not make the program more complicated and the readability of the code remains clean.

3 The warning messages

Sometimes during the execution of blastpgp the following warning message is printed on the screen: "ObjMgrNextAvailEntityID failed with idx 2048" . The problem occurs mostly when the input file contains several query sequences, but can also occur for a single query sequence. As a result of the problem the blastpgp program can write incorrect and incomplete output, which is not desirable . The warning is generated in the function `ObjMgrNextAvailEntityID()` in the file `objmgr.c`:

```
static Uint2 ObjMgrNextAvailEntityID (ObjMgrPtr omp)
{
  Uint2  entityID;
  Int2   idx, jdx;
  Uint4  val;

  if (! assignedIDsInited) {
    ObjMgrInitAssignedIDArray ();
  }
  /* find first 32 bit word with an available entityID */
  idx = assignedIDStackPt;
  while (idx < 2048 && assignedIDsArray [idx] == 0xFFFFFFFF) {
    idx++;
  }
  if (idx >= 2048) {
    ErrPostEx (SEV_ERROR, 0, 0,
               "ObjMgrNextAvailEntityID failed with idx %d", (int) idx);
    return 0;
  }
  /* reset starting point, everything below should be in use */
  assignedIDStackPt = idx;
  /* find first empty bit in array element */

  val = assignedIDsArray [idx];
  jdx = 0;
  while (jdx < 32 && (val & assignedIDsBitIdx [jdx]) != 0) {
    jdx++;
  }
  if (jdx >= 32) {
    ErrPostEx (SEV_ERROR, 0, 0,
               "ObjMgrNextAvailEntityID failed with jdx %d", (int) jdx);
    return 0;
  }
  /* set bit to mark new entityID as in use */
  assignedIDsArray [idx] |= assignedIDsBitIdx [jdx];
  /* calculate entityID */
  entityID = (Uint2) (((Int4) idx) * 32L + (Int4) jdx);

  if (omp != NULL && omp->HighestEntityID < entityID) {
    omp->HighestEntityID = entityID;
  }

  return entityID;
}
```

When the value of the variable idx is equal to or higher than 2048 then the function prints the warning and returns 0, otherwise the program returns entityID which is a Uint2 variable. We notice that idx is used as an index in the array assignedIDsArray, so lets look closer at the array . The definition of the array can be found further up in the code, just before the function ObjMgrInitAssignedIDArray():

```
static Uint4    assignedIDsArray [2050];
static Int2     assignedIDStackPt = 0;
static Boolean  assignedIDsInited = FALSE;
static Uint4    assignedIDsBitIdx [32];
```

The assignedIDsArray is an Uint4 array with 2050 elements, which is not enough for all cases. Before we try to take any measures, we should understand what the assignedIDsArray represents and how it is used.

To make things more simple we will assume that the assignedIDsArray has 2048 elements instead of 2050, the two elements are only a safety marginal. The IDs are represented by the array in the following way. We have 2048 elements , each of them with the size of 32 bits (Uint4) that makes the total number of bits to 65536, which is the maximum number of IDs. Every single bit in the array indicates if the current ID is used or not, where a bit set to 1 indicates an used ID and a bit set to 0 indicates an unused ID.

The entityID, a 16 bits unsigned integer, can have a maximum value of 65535 which is enough to indicate all the bits in the array (the array starts from 0) and is computed in the following way:

```
entityID = (Uint2) (((Int4) idx) * 32L + (Int4) jdx);
```

idx is the index of the current element in the array assignedIDsArray, while jdx is the index of the bit in the current element. We can see that the five first bits represent jdx and the following eleven bits represent idx.

Now that we know what the array assignedIDsArray represents then we should look up how the IDs are used. The IDs are used by a couple of structures in the program which are declared in objmgr.h and only used by the functions in objmgr.c. The following function explains the use of the IDs:

```
NLM_EXTERN void LIBCALL
ObjMgrAddIndexOnEntityID(ObjMgrPtr omp,Uint2 entityID,ObjMgrDataPtr omdp)
{
  Uint1   h,l;
  h=entityID >> 8;
  l=entityID & 0xff;
  if(omp==NULL) omp=ObjMgrGet();
  if(omp){
    if(!omp->entityID_index){
         omp->entityID_index=MemNew(256*sizeof(*omp->entityID_index));
    }
    if(!omp->entityID_index[h]){
         omp->entityID_index[h]=MemNew(256*sizeof(**omp->entityID_index));
    }
    omp->entityID_index[h][l]=omdp;
```

```
  }
}
```

The IDs are used to indicate the elements in `entityID_index` which is a two dimensional array of size 256x256 elements, totally 655536 elements.

The problem with this implementation is that the array `assignedIDsArray` is never reset after an iteration of PSI-BLAST or after each query sequence, which results in lack of available IDs needed by the structures as the number of IDs grows for every iteration. When there are no more available IDs, then the requests are ignored and important data is lost.

The solution to this problem is to reset the array `assignedIDsArray` every iteration to prevent all IDs from being used. This can be easily accomplished by setting the boolean variable `assignedIDsInited` to `false`, which will make the program reset the array. Every time a next available IDs is assigned a check is made if the array is initiated, if not then the function `ObjMgrInitAssignedIDArray()` is called where the array is reset if `assignedIDsInited` is `false`.

The definition of `assignedIDsInited` is found in the same place as the definition of `assignedIDsArray`, which was previously described. The following is the function `ObjMgrInitAssignedIDArray()`, that resets the array `assignedIDsArray`:

```
static void ObjMgrInitAssignedIDArray (void)
{
  Uint4  bit;
  Int2   jdx;
  if (! assignedIDsInited) {
    MemSet ((Pointer) &assignedIDsArray, 0, sizeof assignedIDsArray);
    MemSet ((Pointer) &assignedIDsBitIdx, 0, sizeof (assignedIDsBitIdx));
    /* initialize bit index array */
    bit = 1;
    for (jdx = 0; jdx < 32; jdx++) {
      assignedIDsBitIdx [jdx] = bit;
      bit = bit << 1;
    }
    /* entityID 0 is not available for use */
    assignedIDsArray [0] = assignedIDsBitIdx [0];
    assignedIDStackPt = 0;
    assignedIDsInited = TRUE;
  }
}
```

If the boolean `assignedIDsInited` is `false` then the array `assignedIDsArray` is reset by zeroing all the bits, `assignedIDstackPt` is set to 0 and `assignedIDsInited` set back to `true`.

We will create a new function that will set `assignedIDsInited` to `false`. Here is the implementation of the function:

```
NLM_EXTERN void LIBCALL ObjMgrResetAssignedIDArray (void)
{
assignedIDsInited = FALSE;
}
```

A declaration should also be added in `objmgr.h`:

```
NLM_EXTERN void LIBCALL ObjMgrResetAssignedIDArray PROTO((void));
```

Finally a call to `ObjMgrResetAssignedIDArray()` should be made at the end of every iteration. The call is inserted in the main function of `blastpgp.c` at the end of the iteration loop:

```
 /* Reset the AssignedIDArray , this is a bugfix */
 ObjMgrResetAssignedIDArray ();

 } while (( 0 == search->pbp->maxNumPasses ||
            thisPassNum < (search->pbp->maxNumPasses))
            && (!(search->posConverged)));
```

The implemented solution removed the "ObjMgrNextAvailEntityID failed with idx 2048" problem , resulting in correct output for all the input queries. The execution speed and memory usage aren't noticeably affected. The solution works fine at this moment, however, in the future the max number of IDs (65536) may not be enough. If the amount of IDs is expanded, along with the size of the `entityID_index array`, then the variable `entityID` should be changed from an 16 bits unsigned integer to an 32 bits unsigned integer to be able to indicate all IDs, but this is beyond the scope of this bug fix.

Here is a short summary of changes made in the code by the implementation of the bug fix :

`objmgr.c`: created a new function `ObjMgrResetAssignedIDArray()`

`objmgr.h`: added the declaration of `ObjMgrResetAssignedIDArray()`

`blastpgp.c`: added a call to `ObjMgrResetAssignedIDArray()` in `Main()`

4 Conclusions and further work

We have presented corrections for two bugs in the BLASTPGP program for gapped basic local alignment search. The corrected version of the code has been in test use since mid January 2007 at the Finnish IT center for science, CSC, where it has been tested with 120 sequences which previously caused the program to fail. No more reports of crashes due to segmentation faults of the kind presented here nor any warnings of insufficient space in the object manager have been reported during this time. Therefore, we feel confident in the corrected version and want to make this available to all users of the BLASTPGP program. The modified code and instructions for installing this can be found at `http://www.it.abo.fi/finhpc/blastpgp`.

The next step in our work with the BLAST and BLASTPGP programs will be to optimize the code to perform better on modern processors. A preliminary study has showed that a speedup of 1.2 (measured as the runtime of the original program divided by the runtime of the optimized program) can immediately be achieved by changing the representation of one of the central data structures to a more cache-friendly structure. We are also currently investigating other possibilities to speed up the code.

References

[1] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lippman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[2] Stephen F. Altschul, Thomas L. Madden, Alejandro A. Schäffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lippman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, 25(17):3389–3402, 1997.

Turku
Centre *for*
Computer
Science

Lemminkäisenkatu 14 A, 20520 Turku, Finland | www.tucs.fi

University of Turku
- Department of Information Technology
- Department of Mathematics

Åbo Akademi University
- Department of Computer Science
- Institute for Advanced Management Systems Research

Turku School of Economics and Business Administration
- Institute of Information Systems Sciences

ISBN 978-952-12-1891-0
ISSN 1239-1891