Proxecto de paralelización de aplicacións. O algoritmo de ordenación CombSort.

Arquitectura e Enxeñaría de Computadores

Pablo Castro Valiño
(pablo.castro1@udc.es)

Marcos Chavarría Teijeiro
(marcos.chavarria@udc.es)

2012-2013

Índice

1. O algoritmo

O algoritmo escollido para a realización desta practica foi o algoritmo de ordenación *Combsort*. Este algoritmo presenta na súa versión secuencial unha complexidade de $O(nlog(n))$ no seu caso medio e de $O(n^2)$ no caso peor. Foi deseñado por Włodzimierz Dobosiewicz no ano 1980 e mais tarde no 1991 redescuberto por Stephen Lacey e Richard Box. Supón unha mellora do algoritmo de ordenación por burbulla.

A idea do algoritmo de ordenación por burbulla e ir recorrendo o array comparando elementos veciños e de estaren mal ordenados intercambiar estos dous elementos. O problema é que cando hai valores moi pequenos o final do array, para que o final este estea ordenado temos que recorrer moitas veces o vector ca conseguinte perda de eficiencia.

Comb sort soluciona isto comparando a diferentes distancias, isto é, mentres *bubble sort* comparaba con elementos a distancia 1, comb sort compara con elementos máis afastados aforrando no proceso iteracións ao algoritmo. No Listing 1 podemos ver unha implementación do algoritmo.

Listing 1: Algoritmo secuencial

```
1  void
2  combsort(int a[], long nElements)
3  {
4    short swapped;
5    long i, j, gap;
6
7    gap = nElements;
8    while (gap > 1 || swapped)
9    {
10     gap = gap * 10 / 13;
11     if (gap < 1) gap = 1;
12
13     swapped = 0;
14     for (i = 0, j = gap; j < nElements; i++, j++)
15     {
16         if (a[i] > a[j])
17       {
18         _swap(a,i,j);
19         swapped = 1;
20       }
21     }
22   }
23 }
```

O algoritmo considera unha variable *gap* que vai reducindo iteración tras iteración e que representa o espazo entre os elementos que se comparan. Este *gap* empeza tendo o valor da lonxitude total do array e en cada paso é dividido por un factor de encollemento, que experimentalmente establecese en 1,3. Nas ultimas iteracións do algoritmo este é equivalente ao algoritmo de ordenación por burbulla.

No momento de paralelizar o algoritmo anterior decatámonos de que estabamos ante un algoritmo intenso en datos, é dicir, un algoritmo que realiza operacións moi sinxelas sobre un conxunto enorme de datos. Por unha parte o bucle da liña 8 non é paralelizable pois as iteracións deste bucle son moi dependentes unhas doutras, en cada iteración modificase o array común. Se nos

centramos no bucle interno en certas circunstancias si que podemos paralelizalo. Cando o gap é suficientemente grande pódese en vez de comparar elemento a elemento cargar estes elementos nun rexistro multimedia e comparalos nunhas sola operación unha vez feito a comparación, os intercambios de datos pódense facer en paralelo (empregando OpenMP).

Non obstante, hai dous factores que nos limitan esta posibilidade, por unha parte as operacións que se fan a este nivel son moi rápidas e a ganancia sería pequena mais por outra parte o feito de que o tamaño dos rexistros é de 4 elementos fai que a ganancia sexa a simple vista inexistente ou inapreciable debido a que a sobrecarga que supón cargar e descargar os elementos dos rexistros, minimizaría a ganancia de facer a comparación en paralelo.

A opción que eliximos foi a creación dun algoritmo semellante ao *mergesort* e que é transparente ao algoritmo de ordenación que se use a baixo nivel para ordenar. No Listing 2 pódese ver parte da implementación deste algoritmo.

Listing 2: Parelización parte MPI

```
// Fase 1: Comunicación
// Scatter del array que se quiere ordenar
if( !myrank )
{
  sendcounts = malloc(sizeof(int) * numprocs);
  assert(sendcounts);

  displs = malloc(sizeof(int) * numprocs);
  assert(displs);

  for(i=0; i < numprocs; i++)
  {
    sendcounts[i] = CALC_IT_PER_PROC(numberofitems, numprocs, i);
    displs[i] = i ? displs[i-1] + sendcounts[i-1] : 0;
  }
}

  recvcount = CALC_IT_PER_PROC(numberofitems, numprocs, myrank);
  subarray = malloc(sizeof(int) * recvcount);
  assert(subarray);

  MPI_Scatterv(   data, sendcounts, displs, MPI_INT, subarray,
                  recvcount, MPI_INT, 0, MPI_COMM_WORLD);

// Fase 2: Computo
// Ordenación secuencial local en cada proceso MPI
omp_sort(subarray,(long) recvcount);


// Fase 3: Comunicación
// Ordenación global a partir de los arrays ordenados localmente
if(myrank != 0) //PROCESOS FILLOS
{
  send_data(subarray,recvcount);
}
else //PROCESO ROOT
{
  h = new_heap(numprocs * CHUNK);
  num_el_heap = malloc(sizeof(long) * numprocs);
  indiceroot = 0;

  //METEMOS NO MONTICULO OS DATOS DO PROC ROOT
```

```
43    num_el_heap[0] = insert_root(subarray,recvcount,&indiceroot, h);
44
45    //METEMOS NO MONTICULO OS DATOS DO RESTO DE PROCs.
46    for(proc=1; proc<numprocs; proc++)
47      num_el_heap[proc] = receive_and_insert(proc, h);
48
49    for(i = 0; i < numberofitems; i++)
50    {
51      int v;
52      pop_heap(h,&v,&proc);
53      data[i] = v;
54
55      if ( ! (--num_el_heap[proc]))
56      {
57        num_el_heap[proc] = proc ?
58                            receive_and_insert(proc,h) :
59                            insert_root(subarray,recvcount,&indiceroot,h);
60      }
61    }
62    dispose_heap(h);
63  }
```

O funcionamento do algoritmo é sinxelo, distribúense os datos entre os procesos, estes ordenan localmente os seus datos e o final faise un ordenamento global. O proceso consta de 3 fases:

Fase 1: Difusión. Nesta primeira fase un dos procesos, elixido como proceso pai ou root, que por comodidade será o proceso 0, difundirá a partir da sentencia *scatherv* de MPI o array entre todos os procesos. Para este proceso calculará internamente os parámetros desta función de forma privada e todos os procesos (incluído o root) calcularán en paralelo o número de elementos a ordenar que recibiran e reservarán memoria para estes datos.

Fase 2: Ordenamento Local. Nesta fase cada proceso ordenará localmente o seu fragmento de datos. Para iso chamará a un procedemento que empregara OpenMP para paralelizar a nivel de fío e que detallaremos máis tarde. A saída desta fase, cada proceso ten un fragmento do array ordenado en memoria.

Fase 3: Ordenamento Global. Nesta fase ordenarase o array global empregando os fragmentos que ten cada proceso. Para isto cada proceso enviará unha certa cantidade de elementos ao proceso pai. A cantidade de elementos que envía cada proceso denominase *chunk* e é un parámetro de compilación. O proceso pai recibe estes elementos e introdúceos nun montículo para a continuación entrar nun bucle no que, en cada elemento, extraerá un elemento do montículo introducirao no array final e no caso de non quedar elementos dun procesador no montículo, pediralle novos elementos. Os procedementos para introducir estes elementos no montículo son os que se poden ver no Listing 3.

Listing 3: Procedementos para insertar datos no montículo.

```
1  int
2  insert_root (int datos[], long nDatos, long *indiceroot, heap_t h)
3  {
4      long ini_ind,tam;;
5
```

```
6     if (*indiceroot >= nDatos)
7       return 0;
8
9     ini_ind = *indiceroot;
10
11    tam = (*indiceroot) + CHUNK;
12    if (tam > nDatos)
13      tam = nDatos;
14
15    for(; *indiceroot < tam; (*indiceroot)++)
16      push_heap(h, datos[*indiceroot], 0);
17
18    return (*indiceroot) - ini_ind;
19 }
20
21 int
22 receive_and_insert (int proc, heap_t h)
23 {
24   int i;
25   int value[CHUNK];
26   MPI_Status status;
27
28   MPI_Recv(value, CHUNK, MPI_INT, proc,
29           MPI_ANY_TAG, MPI_COMM_WORLD, &status);
30
31   for (i = 0; i < status.MPI_TAG; i++)
32     push_heap(h, value[i], proc);
33
34   return status.MPI_TAG;
35 }
```

O procedemento para insertar os datos do pai e do resto de elementos é diferente pois, mentres que os datos do pai están na memoria local, os datos do resto de elementos terán que recibirse a través de comunicación MPI. O proceso pai, o ter os datos en memoria local, simplemente mantén un índice que marca cal foi o último elemento que insertou. Os procesos fillos (que non son o pai) executan unha operación MPI_Send que envía como máximo *chunk* elementos ao proceso pai. Esta operación é un envio bloqueante, polo que mentres o pai non execute unha operación MPI_Recv , o proceso fillo quedará detido nesta operación. O método que envía os procesos o pai pódese ver no Listing 4.

Listing 4: Procedemento para enviar datos ao root.

```
1  void
2  send_data (int data[], long nDatos)
3  {
4    int tam;
5    long i;
6
7    for(i=0; i<nDatos; i += CHUNK)
8    {
9      tam = i + CHUNK < nDatos ? CHUNK : nDatos - i;
10
11     MPI_Ssend (data + i, tam, MPI_INT, 0, tam, MPI_COMM_WORLD);
12   }
13
14   MPI_Ssend(NULL, 0, MPI_INT, 0, 0, MPI_COMM_WORLD);
15 }
```

Os métodos que insertan datos no montículo devolven como resultado o número total de datos insertados, de forma que se poida levar a conta de cantos elementos de cada procesador hai no montículo.

Aínda que este algoritmo de *"merge"* foi deseñado para que tivese complexidade lineal, recibíase e insertabanse un dato por iteración con dúas operacións lixeiras computacionalmente falando, extracción do montículo e inserción no montículo, a carga que supón a comunicación entre procesos fai que sexa tremendamente ineficiente.

Para paliar estes efectos introduciuse o concepto de chunk polo que, no canto de enviar os elementos de un en un, enviamos un conxunto maior de elementos. Isto por unha parte reduce a cantidade de comunicación pero tamén aumenta o número de elementos que ten o montículo e polo tanto aumenta o tempo que se tarda en insertar e extraer elementos do montículo. Teremos que acadar unha relación de compromiso entre estes dous factores.

No que respecta a parelización da parte de OpenMP o que fixemos foi replicar o algoritmo que empregamos na parte anterior pero a nivel de thread. A implementación pódese ver no Listing 5.

Listing 5: Paralelización parte OpenMP.

```
1  void
2  omp_sort (int a[], long nElements)
3  {
4    long  it_p_th, i, *mx_sta, *mx_end;
5    int num_th, thread, * aux, proc, value;
6    heap_t h;
7
8    num_th = omp_thread_count();
9
10   if(num_th > nElements)
11   {
12     combsort(a,nElements);
13     return;
14   }
15
16   mx_sta = malloc(sizeof(long) * num_th);
17   mx_end = malloc(sizeof(long) * num_th);
18
19   it_p_th = nElements / num_th + ((nElements % num_th) > 0);
20
21   #pragma omp parallel for private(thread,i)
22   for(i=0; i < nElements; i += it_p_th)
23   {
24     thread = omp_get_thread_num();
25
26     mx_sta[thread] = i;
27     mx_end[thread] = i + it_p_th;
28     if(mx_end[thread] >= nElements)
29       mx_end[thread] = nElements;
30
31     combsort(a + mx_sta[thread], mx_end[thread] - mx_sta[thread]);
32   }
33
34
35   h = new_heap(num_th);
36   i = 0;
37   aux = malloc(sizeof(int) * nElements);
38
```

```
39    for(proc=0; proc < num_th; proc++)
40    {
41      if(mx_sta[proc] < mx_end[proc])
42      {
43        push_heap (h,
44              a[mx_sta[proc]],
45              proc);
46        (mx_sta[proc])++;
47      }
48    }
49
50    do
51    {
52      pop_heap(h, &value, &proc);
53      aux[i++] = value;
54
55      if(mx_sta[proc] < mx_end[proc])
56      {
57        push_heap (h,
58              a[mx_sta[proc]],
59              proc);
60        (mx_sta[proc])++;
61      }
62    }while(!empty_heap(h));
63
64    #pragma omp parallel for
65    for(i=0; i< nElements; i++)
66      a[i] = aux[i];
67
68    free(aux);
69    dispose_heap(h);
70  }
```

A diferencia do caso da implementación en MPI, neste caso o tratarse de memoria compartida non hai necesidade de empregar mensaxes. O que se fai é empregar dous arrays *mx_ sta* e *mx_ end* que marcan o principio e final do sub-array que ordena cada thread. Na fase de *"merge"* empregaranse estos mesmos arrays para controlar qué elementos se introduciron no montículo.

Necesitamos tamén un array auxiliar *aux* que empregaremos para gardar o array ordenado debido a que a diferencia da implementación en MPI non se pode facer no propio vector de entrada. O feito de ter que usar este array fai que o consumo de memoria aumente de forma relevante. Ademáis, o terminar o proceso de merge pasarase o resultado dese array *aux* o array orixinal.

2. Rendemento do programa

Para comprobar o rendemento do programa fixemos medicións co máximo numero de procesadores e fíos que permiten os ordenadores do CESGA. Este número máximo é 24. Por iso probamos a executar o noso programa para ordenar 150000000 de elementos cas seguintes combinacións de procesadores e fíos.

- 1 Procesador e 24 fíos.
- 2 Procesador e 12 fíos.
- 3 Procesador e 8 fíos.
- 4 Procesador e 6 fíos.
- 6 Procesador e 4 fíos.
- 8 Procesador e 3 fíos.
- 12 Procesador e 2 fíos.
- 24 Procesador e 1 fíos.

Fixemos probas tamén con diferentes valores para o parámetro *chunk*. Probamos para 10, 50, 100, 250, 500, 1000, 1500 e 2000. Os resultados pódense ver na Táboa 1. Estas son medicións de tempo. Para ter unha referencia, o tempo que lle leva ó programa secuencial ordenar a mesma cantidade de elementos é de 148,56 segundos.

	10	50	100	250	500	1000	1500	2000
1×24	—	—	653	676	707	766	797	827
2×12	835	368	328	324	341	354	368	383
3×8	692	271	233	219	224	231	237	247
4×6	621	228	180	166	168	172	178	180
6×4	558	174	135	123	115	121	116	120
8×3	534	156	115	93	111	92	91	91
12×2	429	126	91	71	67	70	64	68
24×1	22	22	22	24	18	22	23	24

Táboa 1: Táboa de tempos con diferentes configuracións.

Como podemos ver os tempos con un alto número de procesadores son mais altos que os que usan un alto número de fíos. Esto tamén se pode ver na Figura 1 onde vemos unha comparativa dos tempos con diferentes chunks e configuracións.

Nesta gráfica podemos confirmar o que xa adiantabamos, a comunicación MPI é tan pesada que fai que aumente o tempo do algoritmo en gran cantidade a pesar de empregar cantidades altas de chunk. Podemos ver como a liña negra marca o tempo da execución secuencial e que so as configuracións que empregan menos de seis procesadores melloran este tempo.

Por outra parte tamén podemos ver nesta gráfica a relación de compromiso que temos que acadar en canto o valor do chunk xa que, por unha parte un maior numero de chunk implica menos comunicacións pero tamén implica un

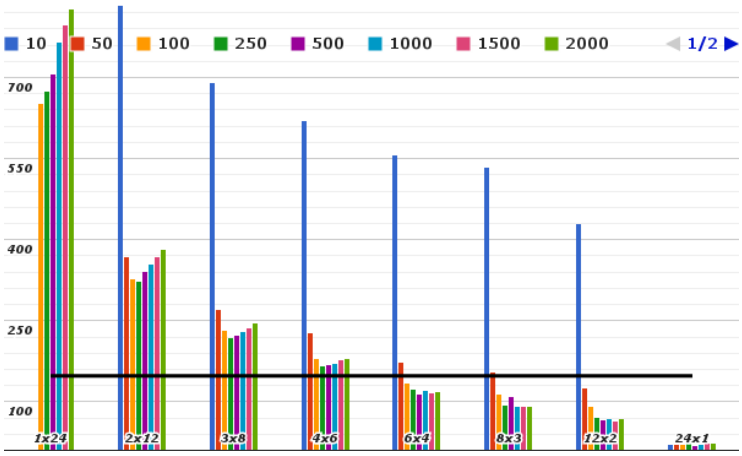

Figura 1: Gráfica de tempos con diferentes configuracións de fíos, procesadores e valores de chunk.

montículo máis grande e polo tanto as operacións que se fan con este montículo son máis pesadas.

Se empregamos *chunks* baixos como 10 ou 50 o rendemento do programa empeora considerablemente. Non obstante, se empregamos un *chunk* alto de 1000 ou 2000 o rendemento tamén baixa como se pode ver na figura.

Xa que o tempo con un so procesador é o mellor, estudamos a evolución do programa o usar máis ou menos fíos. Na Figura 2 podemos ver os resultados.

Figura 2: Gráfica de tempos con diferentes configuracións de fíos, procesadores e valores de chunk.

Como podemos ver na gráfica, o *speedup* non medra xunto co numero de fíos. Cando usamos 5 fíos obtemos con respecto a versión secuencial, un *speedup* de 7,4. A pesar de ser este un moi bo resultado para ese numero de fíos, se aumentamos mais o numero de fíos, o *speedup* mantense constante. As razóns disto

encóntranse no importante cuello de botella que temos na fase de ordenamento global do algoritmo cuxos efectos son mais importantes na parte de MPI pero que tamén se notan na parte de OpenMP. Por outro lado o feito de ter que traspasar os resultados do array auxiliar ao array orixinal, aínda que se fai en paralelo, tamén supón unha penalización de memoria.

3. Conclusións

Unha das principais vantaxes do algoritmo implementado é a súa transparencia a hora de escoller o algoritmo de ordenación final. Esta característica fai que sexa interesante estudar o rendemento do algoritmo con outros algoritmos de ordenación.

Por outro lado non podemos esquecer o feito de que a implementación actual do algoritmo non aproveita en ningunha media o gran numero de procesadores dos que se dispoñía debido a sobrecarga producida pola comunicación, a pesar de introducir o concepto de chunk e de cambiar a implementación inicial con arbores a unha implementación con montículos. Sería interesante estudar algoritmos de *"merge"* diferentes e estudar o seu comportamento conxuntamente ca parte programada en OpenMP.

Outro aspecto mellorable é, na parte de OpenMP, a necesidade de ter que ordenar no mesmo array que se recibe. Se a función que ordena arrays empregando OpenMP recibise un punteiro a un array que se puidese modificar, aforrariamos ter que traspasar os datos do array auxiliar ao array final.