Algoritmos de Ordenação

Carlos Torrão 55381 João Martins 55385 Maria Couceiro 53964

Resumo: Neste artigo são apresentados vários algoritmos de ordenação: *bubble sort*, *mergesort*, *quicksort*, *hyperquicksort*, *rank sort*, *counting sort* e *radix sort*. É feita uma descrição do seu funcionamento em série e em paralelo, fazendo-se referência a vantagens e desvantagens e problemas resultantes do seu uso. Concluímos que, na grande maioria dos casos, a implementação paralela dos algoritmos produz melhores resultados a nível de complexidade temporal que em série.

1 Introdução

A ordenação é um dos aspectos fundamentais das ciências computacionais. Torna-se, então, importante reduzir ao máximo a complexidade temporal dos algoritmos que lidam com este problema. As melhores ordenações em série normalmente demoram O(n log n), tempo que tende a agravar com o aumento do número de elementos. Deste modo, foram desenvolvidas versões para funcionamento em paralelo destes algoritmos, cujo objectivo é diminuir consideravelmente o tempo de execução dos mesmos.

Neste texto vamos abordar vários algoritmos de ordenação. Relativamente aos que realizam operações de comparação e troca, descrevemos o *bubble sort*, *quicksort* e *hyperquicksort*, *mergesort*, ***odd-even mergesort*** **e** ***bitonic mergesort***. Também falamos sobre o *rank sort* e o *counting sort,* que não recorrem a este tipo de operações. No que diz respeito a algoritmos que obtêm uma performance muito superior quando paralelizados, vamos descrever o *radix sort*.

2 Bubble Sort

2.1 Sequencial

O algoritmo de ordenação *bubble sort* usa uma estratégia de "comparação e troca", que é aplicada em várias iterações sobre os dados a ser ordenados. Passo a explicar os passos do algoritmo sequencial.

Partindo do principio que se deseja ordenar (de forma crescente) um vector de números nas seguintes posições: x_0, x_1, …, x_{n-1}, x_n. O algoritmo começa por comparar o número que se encontra na posição x_0 com o da posição x_1, e troca (quando necessário) os números de maneira a ficar o maior dos dois na posição x_1. De seguida, repete o mesmo procedimento para as posições x_1 e x_2, onde na posição x_2 ficará o maior dos números que se encontra nas posições x_1 e x_2. Este processo é repetido de forma análoga até chegar à comparação entre as posições x_{n-1} e x_n. Este processo anterior de comparações e trocas será considerado uma fase do algoritmo, sendo o algoritmo *bubble sort* constituído (na pior das hipóteses) por *n-1* fases semelhantes à anterior.

2.2 Paralelo

Após esta breve descrição do algoritmo sequencial, a questão que se impõe é a seguinte: será que é possível uma versão paralela eficiente deste algoritmo?

A resposta a esta pergunta é afirmativa. Através da técnica de *pipelining* é possível melhorar o desempenho do algoritmo, em comparação com a versão sequencial. Isto deve-se ao facto de ser possível serem executadas várias fases em simultâneo na versão com *pipelining*.

É possível tal método, pois como cada fase só faz uma comparação por cada par de posições, quando a primeira fase se encontra na comparação entre as posições x_2 e x_3, a segunda fase já pode (sem problema algum) fazer a comparação entre as posições x_0 e x_1. A terceira fase, terá início quando a segunda fase estiver a comparar as posições x_2 e x_3, e a primeira fase, por sua vez, se encontrar na comparação das posições x_4 e x_5. (Ver Figura 1)

Existe ainda uma variação do algoritmo *bubble sort*, cujo nome é *odd-even (transposition) sort*, que consiste em duas fases, a fase *even* (par) e a fase *odd* (ímpar). Na fase *even* (par), comparam-se (e se necessário trocam-se) as posições pares com a posição seguinte a cada uma delas. Na fase *odd* (ímpar), a ideia é análoga à anterior, mas para as posições ímpares. (Ver Figura 2)

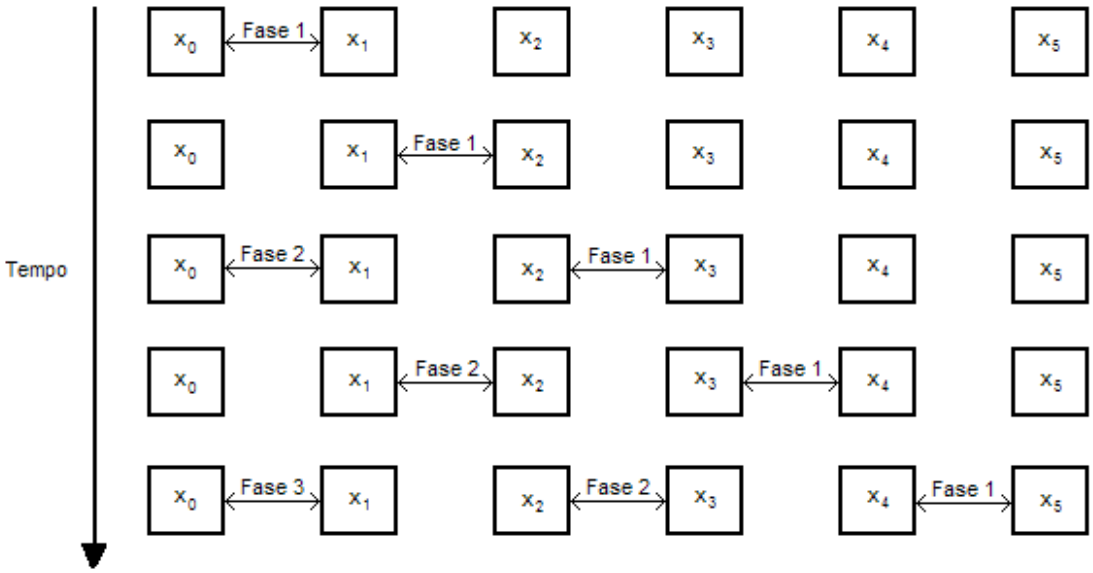

Figura 1 – *Bubble sort* com *pipelining*

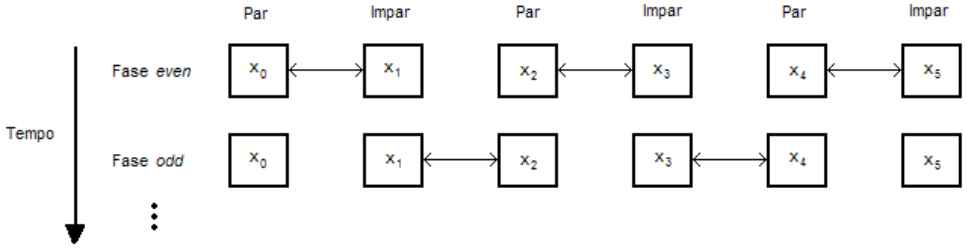

Figura 2 – *Odd-Even transposition sort* em paralelo

Como é de prever, esta solução em sequencial não traz benefícios nenhuns em relação ao *bubble sort* (sequencial), mas o mesmo já não se pode afirmar da versão paralela desta, que é bastante mais eficiente. Esta solução paralela tem complexidade O(n), enquanto que o algoritmo *bubble sort* na sua versão sequencial, apresenta uma complexidade de $O(n^2)$.

3 Merge Sort

3.1 Sequencial

O algoritmo *merge sort* consiste em ir separando em metades uma lista de elementos, até ficar com todos os elementos separados. Esta técnica tem o objectivo de "dividir-para-conquistar". Após estes elementos estarem separados, o algoritmo procede à sua junção (*merge*), retornando no final uma única lista com todos os elementos ordenados.
Vamos considerar a título de exemplo, uma lista de quatro elementos sem ordem alguma, e que se querer ordenar de forma crescente todos os seus elementos.

Inicialmente, durante todo o processo da divisão da lista, não é feita qualquer computação sobre os elementos e sua ordem na lista, e a primeira lista de quatro elementos é apenas separada em metade, ficando duas listas, sendo de seguida o método repetido, ficando desta vez todos os elementos separados.
É na fase da junção (*merge*) que o algoritmo ordena todos os elementos da forma pretendida. Neste caso, em que queremos ordenar de forma crescente, o algoritmo pega nos primeiros dois elementos e junta-os numa lista, ficando o menor valor na primeira posição e o maior na segunda posição. Para os dois últimos elementos que sobram, utiliza o mesmo método. Este processo é agora repetido para as duas listas criadas anteriormente, juntando os elementos de cada uma dessas duas listas numa única lista, ficando essa lista ordenada de forma crescente.

3.2 Paralelo

A versão simples paralela, baseia-se em atribuir a cada processador uma lista de

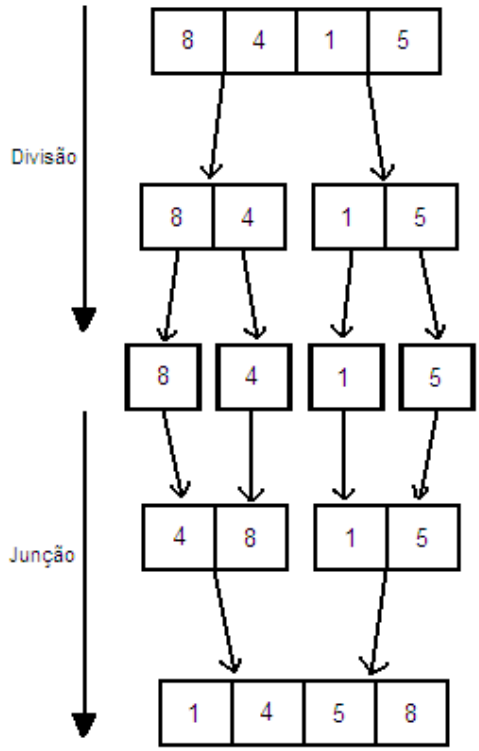

Figura 3 – *Merge sort* sequencial

elementos, na parte inicial da divisão da lista a ordenar.
Esta versão tem o problema de em algumas fases do algoritmo ser necessária bastante comunicação entre processadores. Mas por outro lado, esta versão torna-se bastante leve a nível computacional para cada um dos processadores envolvidos, pois cada processador é responsável por uma lista pequena.

3.3 Odd-Even merge sort

Este algoritmo tem como objectivo ordenar duas listas ordenadas, numa só lista ordenada. Isto apresenta potencial, para ser invocado de maneira recursiva. Na parte de junção das listas no algoritmo *merge sort*.
Considerando a lista A = $a_1, a_2, ..., a_n$ e a lista B = $b_1, b_2, ..., b_n$. O algoritmo começa por criar uma lista C, constituída pelas posições (índices) ímpares de ambas as listas, ou seja, C = $a_1, a_3, ..., a_{n-1}, b_1, b_3, ..., b_{n-1}$. Existe o processo análogo para as posições (índices) pares, sendo criado a lista D = $a_2, a_4, ..., a_n, b_2, b_4, ..., b_n$. Para maior facilidade de percepção futura, os elementos da lista C estarão referenciados da seguinte forma: $c_1, c_2, ..., c_n$, e o mesmo para os elementos da lista D.
A lista final ordenada, E, constituídas pelos elementos: $e_1, e_2, ..., e_{2n}$, é obtida através das seguintes regras ($1 \leq i \leq n-1$):

- $e_{2i} = \min\{ c_{i+1}, d_i \}$
- $e_{2i+1} = \max\{ c_{i+1}, d_i \}$

Basicamente, esta lista E é obtida pela comparação das posições pares e ímpares das listas C e D. O primeiro elemento da lista E, e_1, é dado por c_1, e o mesmo acontece com o ultimo elemento, e_{2n}, que é dado por d_n. Isto deve-se ao facto, de em c_1 estar o elemento menor das listas C e D, e de em d_n estar o maior elemento das listas C e D.

3.4 Análise das versões

O *merge sort* sequencial tem uma complexidade temporal O(*n.logn*). O que é bastante bom para um algoritmo de ordenação sequencial.
O *odd-even merge sort* em comparação com a versão sequencial é bastante superior, pois apresenta uma complexidade temporal O($log^2 n$) com *n* processadores.

4 Quicksort

O algoritmo *Quicksort*, inventado por C.A.R. Hoare, é um algoritmo recursivo que tem como base a comparação de valores para ordenar uma lista desordenada. Mais especificamente, quando é passada uma lista de números, o algoritmo selecciona um número dessa lista, tornando-o *pivot*. Após isso, a lista é partida em duas sub-listas: uma contendo números inferiores ao *pivot*, e outra com os números superiores. Chama-se depois a si mesma recursivamente para ordenar as duas sub-listas. A função retorna a concatenação da primeira lista, *pivot* e segunda lista.

4.1 Quicksort Paralelo

Existem algumas vantagens na utilização deste algoritmo, no que toca a paralelização de algoritmos. Primeiro, porque é considerado dos algoritmos mais rápidos no que trata a ordenação através de comparação de valores. Em segundo lugar, porque como é um algoritmo que se chama a si próprio recursivamente, tem concorrência natural, sendo que essas chamadas podem executar-se independentemente.

Em relação ao desenvolvimento deste algoritmo, inicialmente os valores encontram-se distribuídos ao longo dos processos. Escolhemos então um *pivot* de cada um dos processos e divulgamo-lo. Após isso, cada processo divide os seus números em duas listas: aqueles que são menores ou iguais ao *pivot*, e aqueles que são maiores do que o *pivot* (que chamaremos aqui de "lista menor" e "lista maior" respectivamente). Cada processo na parte de cima da lista de processos envia a sua "lista menor" para um processo concorrente na parte inferior da lista de processos e recebe uma "lista maior" em retorno. Neste momento, os processos na parte superior da lista de processos têm valores maiores do que o *pivo*t, e os processos na parte inferior da lista de processos têm valores menores ou iguais ao *pivot*. A partir daqui os processos dividem-se em dois grupos e o algoritmo é chamado de novo. Em cada grupo de processos é distribuído um *pivot*. Os processos dividem a sua lista e trocam valores com processos concorrentes. Após *log p* recursões, cada processo tem uma lista desordenada de valores, que são diferentes dos valores possuídos pelos outros processos. Cada processo pode agora ordenar a lista que controla usando *quicksort* sequencial.

Em relação à complexidade, se tivermos *p* processadores, podemos dividir a lista de *n* elementos em *p* sub-listas em O(*n*). Após isso, a sua ordenação será em O(*(n/p)log(n/p)*).

Uma das vantagens deste algoritmo em relação a outros algoritmos em paralelo é o facto de não ser necessária a sua sincronização. Um processo é criado por cada sub-lista gerada, sendo que esse processo só trabalha com essa lista, não comunicando com os outros processos.
O tempo de execução do algoritmo começa quando o primeiro processo começa a sua execução, e termina quando o ultimo processo termina a sua execução. É por isto que é importante assegurar que todos os processos tenham a mesma carga de trabalho, para que terminem todos por volta do mesmo tempo. Neste algoritmo, a carga de trabalho é relacionada com o número de elementos controlados pelos processos. Este algoritmo tem como ponto negativo que neste caso faz um mau trabalho a balancear o tamanho das listas. Tem que se conseguir um *pivot* perto da mediana dos valores, para garantir maior divisão de trabalho pelos processos.
No entanto, poderia ser feito um melhor balanceamento do tamanho da lista pelos processos, se em vez de escolher um número arbitrário da lista para ser *pivot*, escolhêssemos um valor mais perto do valor mediano da lista ordenada. Este ponto é a motivação por trás do próximo algoritmo em paralelo: *hyperquicksort*.

4.2 Hyperquicksort

Este algoritmo, inventado por Wagar, começa onde o *quicksort* acaba. Considerando que os valores continuam a mover-se de processos para outros processos, teremos então um *pivot* para dividir os números em dois grupos: a parte superior e a parte inferior. Devido à lista de elementos de cada processo estar ordenada, o processo responsável por fornecer o *pivot* pode usar o mediano da sua lista como *pivot*. Este *pivot* é então, mais próximo do verdadeiro mediano da inteira lista desordenada do que escolhendo um valor arbitrário.
Os próximos três passos do *hyperquicksort* são os mesmos do que o *quicksort* paralelo.
Após a execução desses passos, cada processo tem uma sub-lista ordenada própria e uma sub-lista ordenada que recebe de um outro processo. Esse processo vai então juntar as duas listas, para que todos os elementos que controla estejam ordenados. É importante que os processos terminem esta fase com listas ordenadas, porque quando o algoritmo se chamar de novo, dois processos precisarão de escolher o elemento mediano das suas listas como *pivot*.
O *hyperquicksort* assume que o número de processos é um expoente de 2. Se imaginarmos a lista de processos como um *hypercube*, poderemos modelar as comunicações, de modo a que estas sejam sempre feitas entre pares de processos adjacentes.

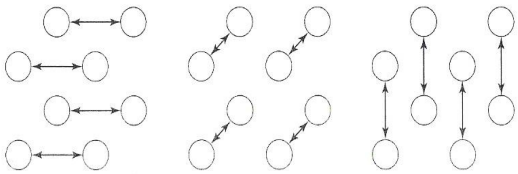

Figura 4: Neste exemplo há 8 processos, fazendo *log p* = 3 vezes separar-e-juntar

Há a referir também a eficiência deste algoritmo, onde para *p* processadores, serão ordenados *n* elementos, onde *n >> p*. No início do algoritmo, cada processo não tem mais do que *n/p* valores. A esperada complexidade temporal do passo inicial do *quicksort* é Θ*[(n/p)log(n/p)]*. Assumindo que cada processo guarda *n/2p* valores e transmite *n/2p* valores em cada passo de separar-e-juntar, o numero esperado de comparações necessárias para juntar as duas listas numa única lista ordenada é *n/p*. O numero de comparações feitas durante o algoritmo todo é Θ[(*n/p)(logn+logp*)], para *log p* iterações.
Assumindo que cada processo passa metade dos seus valores em cada iteração, o tempo necessário para enviar e receber *n/2p* valores ordenados para e de outro processo, é Θ(*n/p*).
A complexidade do tempo sequencial do *quicksort* é *n.log n*. O *overhead* da comunicação do *hyperquicksort* é *p* vezes a complexidade da comunicação, ou Θ(*n.logp*).

Para finalizar, o *hyperquicksort* tem duas desvantagens que limitam as suas vantagens. Primeiro, o número esperado de vezes que um valor é passada de um processo para outro é *(logp)/2*. Este *overhead* da comunicação limita a escalabilidade do algoritmo em paralelo. Poderíamos reduzir este *overhead* se conseguíssemos encontrar uma maneira de enviar as chaves directamente para os seus destinos finais.
Em segundo lugar, a maneira como os *pivots* são escolhidos podem levar a certos problemas no balanceamento da carga de trabalho entre processos. Se conseguíssemos ter exemplares de todos os processos, poderíamos ter uma melhor divisão das listas de elementos pelos processos.

5 Rank Sort

O *rank sort*, também conhecido como *enumeration sort*, é um exemplo de um algoritmo de ordenação estável, isto é, os registos aparecem na sequência ordenada pela mesma ordem que estão na sequência original, mantendo assim a sua posição relativa.
O seu modo de funcionamento consiste em determinar a quantidade de números menores que um determinado número, sendo assim calculada a sua posição (*rank*) na lista ordenada. Tomando como exemplo um vector com *n* números, a[0] … a[n-1], a[0] começa por ser comparado a todos os outros elementos, sendo a soma total dos menores que ele guardados em *rank*. Este valor é o seu índice no vector ordenado *b*, onde é colocado. A mesma operação é repetida para todos os elementos do vector *a*.
Segue-se uma possível implementação para este algoritmo, tendo em conta a existência de valores duplicados:

```
for(i=0; i < n; i++){
    rank = 0;
    for(j=0; j < n; j++){
        if(a[i] < a[j] ||
           (a[i] == a[j] && j < i))
               rank++;
    b[rank] = a[i];
}
```

A nível de complexidade, este algoritmo é pouco eficiente quando executado em modo sequencial, uma vez que faz *n*-1 comparações *n* vezes, ficando o algoritmo com *n*(*n*-1) iterações, o que resulta numa complexidade temporal de O(n^2).
Como é necessário existir um acesso partilhado ao vector, este algoritmo é mais indicado para sistemas de memória partilhada. O cálculo da posição final de cada número no vector ordenado é dividido por *n* processadores, que a determinam em paralelo, ficando assim o algoritmo com uma complexidade temporal de O(*n*). No entanto, é possível implementá-lo em sistemas de memória distribuída. Neste caso, um processo central (*master*) distribui pelos outros processadores (*slaves*) *n*/*p* elementos do vector, sendo *p* o número total de processadores. Cada processador fica responsável por calcular o *rank* de cada um dos seus *n*/*p* elementos. No final, os cálculos são reportados ao processo central, que constrói o vector ordenado. Esta distribuição só compensa quando *n* é suficientemente grande, devido ao custo das operações de comunicação. Uma outra limitação do uso deste algoritmo neste tipo de sistemas prende-se com a utilização de uma grande quantidade de memória, uma vez que cada processador precisa de uma cópia do vector desordenado e de um vector do tamanho deste último para poder posicionar os elementos repetidos ordenados na posição correspondente.
Utilizando n^2 processadores e uma estrutura em árvore, a complexidade temporal diminui para O(*log n*). Cada processador executa a comparação de um elemento com outro, somando-se os

resultados de cada comparação cada vez que se muda de nível na árvore, como se pode ver na figura 5.

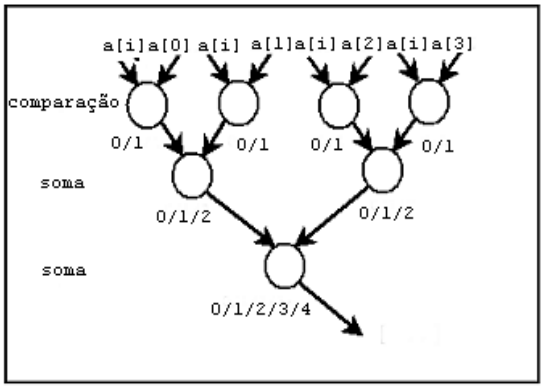

Figura 5: Computação do cálculo do *rank* em paralelo.

Como a utilização de n^2 processadores (e até mesmo de n processadores, quando n é muito grande) se torna proibitiva, é comum agrupar números e distribuí-los pelos processadores disponíveis. Partindo do princípio que temos m grupos, seriam apenas necessários n/m processadores para calcular a posição no vector ordenado em cada grupo de números (sem que este cálculo seja paralelizado). No entanto, o número de operações em cada processador aumentaria m vezes.

6 Counting Sort

Outro exemplo de um algoritmo estável é o *counting sort*. Sendo uma optimização do *rank sort* para o caso em que todos os números a ordenar são inteiros, reduz a complexidade temporal da execução em série de $O(n^2)$ para $O(n)$.
Tal como no *rank sort*, a ideia fundamental é determinar, para cada número i, a quantidade de elementos menores que ele. Os vectores a e b, ambos com dimensão n, contêm, respectivamente, os números desordenados e ordenados. Existe também um vector c, de dimensão m, que contém um elemento para cada valor possível de a, por exemplo, de 1 até m.
Para calcular a quantidade de elementos inferior a um dado número, utiliza-se o método denominado *prefix sum*, que consiste na soma parcial de todas as posições de um dado vector.
Segue-se uma possível implementação para este algoritmo, tendo em conta a existência de valores duplicados:

```
for(i = 1; i <= m; i++)
   c[i] = 0;
for(i = 1; i <= m; i++)
   c[a[i]]++;
for(i = 2; i <= m; i++)
   c[i] = c[i] + c[i-1];
for(i = 1; i <= n; i++){
   b[c[a[i]]] = a[i];
   c[a[i]]--;
}
```

Este código tem complexidade temporal de $O(n + m)$, mas caso $m = O(n)$, O algoritmo corre em $O(n)$.
Na paralelização do *counting sort*, usa-se a versão paralela do *prefix sum*, que requer um tempo de $O(\log n)$ para n-1 processadores. Para atingir este tempo, partimos recursivamente o vector c ao meio e somamos as duas metades também recursivamente, estando associada a esta computação uma árvore binária completa, em que cada nó interno contém a soma das suas folhas descendentes. A ordenação final pode ser feita em $O(n/p)$, caso tenhamos $n < p$ processadores, dividindo-se o vector em p sub-vectores, cada um com n/p elementos, ou então $O(1)$, caso tenhamos n processadores, em que cada um executa o corpo do ciclo.

7 Radix Sort

O *radix sort* é um dos algoritmos de ordenação mais utilizados, devido à sua rapidez e simplicidade de implementação. Parte do pressuposto que os dígitos representam valores e que a sua posição, que está ordenada do menos significativo para o mais significativo, indica o seu peso relativo. Imaginando que um número tem b bits (ou dígitos, caso se esteja a falar de notação decimal), na sua versão mais simples, o algoritmo ordena os números em b passagens, sendo que em cada n-ésima passagem se ordena pelo n-ésimo bit menos significativo. Este número de passagens pode ser reduzido de b para b/r se se considerar blocos de r dígitos em vez de um único bit. A complexidade temporal vai depender do algoritmo de ordenação usado, que tem de ser estável, de modo a preservar a ordem relativa dos elementos.
O *counting sort* é um algoritmo de ordenação muito usado pelo *radix sort*, cuja complexidade temporal em modo sequencial é $O(n + m)$, sendo que n corresponde ao número de elementos inteiros, estando cada um deles entre, por exemplo, 1 e m. Com números de dimensão r, ficamos com um intervalo de 1 a 2^r-1 e b/r fases do algoritmo. Assim,

obtém-se uma complexidade temporal em modo sequencial de O(b/r ($n + 2^r$)), mas se b e r forem constantes, a complexidade baixa para O(n).
Podemos paralelizar o *radix sort* usando um algoritmo de ordenação paralelo, como o *counting sort* descrito na secção anterior, atingindo-se assim um tempo de O(log n) com n processadores e b e r constantes. No entanto, em sistemas de memória distribuída, para grandes quantidades de dados, o tempo gasto em comunicação é demasiado elevado e o balanceamento da carga é muito irregular, pois não se sabe à partida o formato de todos os elementos a ordenar. Consequentemente, foram propostas várias formas de dividir a carga entre os vários processadores, com o objectivo de diminuir estes problemas. Uma dessas propostas é denominada de "*Load Balanced Parallel Radix Sort*" e consiste em atribuir a cada processador exactamente o mesmo número de elementos a ordenar, eliminando assim o problema da distribuição de carga, mas gastando ainda muito tempo a redistribuir os elementos pelos processadores. O "*Partitioned Parallel Radix Sort*" veio solucionar o problema de comunicação desta proposta, uma vez que elimina a redistribuição global de elementos, ao ordená-los, inicialmente, pelos seus g bits mais significativos e, depois, distribuir grupos destes elementos pelos processadores disponíveis, onde são ordenados pelos seus $b - g$ bits menos significativos, reduzindo assim a troca de mensagens entre processadores.

8 Conclusão

Durante a elaboração deste artigo, pudemos concluir que a execução em paralelo dos algoritmos estudados aumenta consideravelmente a eficiência temporal da ordenação de elementos. Há que referir que existem outros que não foram aqui abordados, pois não os considerámos tão relevantes quanto os aqui expostos.
Foi ainda possível observar que, relativamente à investigação nesta área, há uma grande preocupação em optimizar certos aspectos dos algoritmos existentes de modo a reduzir os *overheads* introduzidos pela comunicação e distribuição de carga pouco equilibrada, que continuam a limitar o seu funcionamento.

9 Referências

1. Michael Quinn, *Parallel Programming*, McGrawHill, 2003

2. Barry Wilkinson and Michael Allen, *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*, Prentice Hall, 2005

3. Philippas Tsigas and Yi Zhang, *A Simple, Fast Parallel Implementation of Quicksort and its Performance Evaluation on SUN Enterprise 10000*

4. N. M. Amato, R. Iyer, S. Sundaresan, Y. Wu*, A Comparison of Parallel Sorting Algorithms on Different Architectures*, 1998

5. S. Lee, M. Jeon, D. Kim, A. Sohn, *Partitioned Parallel Radix Sort*, 2002

6. A. Sohn, Y. Kodama, *Load Balanced Parallel Radix Sort*, 1998
7. http://beowulf.csail.mit.edu/18.337/book/Lecture_03-Parallel_Prefix.pdf, 2006