A Programming and Performance Comparison of OpenMP and MPI for Concordance Benchmark

M. S. Aljabri, P.W. Trinder

April 24, 2013

Abstract

Of the parallel programming models available OpenMP is the de facto standard for shared memory and MPI is the standard for distributed memory systems. This report presents a concordance benchmark and uses it to make a programming and performance comparison for MPI and OpenMP and their toolsets.

We present the development of a sequential concordance benchmark which is the basis of the first phase of the SICSA multicore challenge. Initial MPI and OpenMP versions are developed, discussing challenges such as finding sequences across worker boundaries and load balancing. Initial versions are profiled and benchmarked on a multicore architecture using appropriate toolsets. Several means for reducing overheads are investigated and the tuned parallel versions of the concordance benchmark are developed.

A performance comparison of the OpenMP and MPI concordance benchmarks on an 8 core architecture show that OpenMP delivers better performance than MPI, e.g. a speedup of 5.3 compared with 2.6 for MPI, reflecting lower communication/synchronisation costs.

A programmability investigation shows that the effort required to develop and tune the OpenMP version is less than that required for MPI, as a direct benefit of the shared memory model.

1 Introduction

1.1 Context

The emergence of multi-core technology has created a genuine need for parallel software development technologies [35]. At present OpenMP [11] and MPI [38] are de facto standards for shared memory and distributed memory systems respectively. OpenMP was developed as a response to cumbersome native thread programming. Its higher level of abstraction makes threads easier to handle, but, although it is easy to code, it lacks flexibility. MPI is a message passing library and is widely used in High Performance Computing. Although designed for distributed memory systems, in which processors do not share memory, it can also be deployed across shared memory systems like multi-cores [18].

This report compares programmability and performance as delivered by the dominant MPI and OpenMP programming paradigms and their toolsets. The basis of the comparison is a

Technical Report HW-MACS-TR-0106.

concordance benchmark involving a textual analysis to enumerate sequences of words, the pages they occur on, and their frequency. Constructing a concordance is a sophisticated string searching algorithm[14]. Our algorithm is unusual in that it involves constructing a concordance for every sequence of words up to the length N. We are not aware of a published algorithm for this problem. There is, however, a body of work on the far simpler problems of single word sequences. For example Heinz et al [26] explore and compare the sequential performance of hash table variants and trees for determining word frequency. Dhillon et al [16] constructed a single-word concordance using multiple threads with multiple local hash tables for each thread, and then combined the results into a global hash table.

1.2 Contributions

The report makes the following contributions

- A sequential concordance has been developed in C and this program is the basis of the first phase of the SICSA multicore challenge [2]. The program reads a text file with page breaks and the length of word sequences (N), and constructs a concordance listing the pages on which each sequence up to N words occurs. The key data structure used in the C program is the hash table, where the key is the word or the sequence of words, the value is the frequency, and a linked list represents the page numbers on which they occur (Section 5.1).

- Initial parallel concordance programs are developed in MPI (Section 5.2) and OpenMP (Section 5.3), and solutions are identified to resolve some interesting parallelisation problems, such as finding sequences across worker boundaries and load balancing (Section 5.4).

- The initial parallel versions are developed, profiled and benchmarked on a multi-core architecture using appropriate toolsets for each version e.g. ompP for OpenMP and mpiP for MPI. The profiling identifies synchronisation as the performance bottleneck in OpenMP and communication as the bottleneck for MPI. The means of reducing the overheads identified by profiling are investigated to assist in the development of tuned parallel versions (Sections 6.1, 6.2).

- The tuned concordance programs are used to generate a performance comparison of OpenMP and MPI on an 8 core architecture focusing on runtime, absolute speedup, and efficiency. The key results are as follows:

  - The OpenMP version achieves an absolute speedup of 5.3 while the MPI version achieves just 2.6 as a result of higher communication costs (Section 6.3).

  - To reduce the synchronisation overheads by 92%, the tuned OpenMP version has over 650 hash tables (Section 6.1.3).

  - Several alternatives are investigated to improve the MPI performance including compressing communication data, replacing blocking with non-blocking communication, and grouping messages (Section 6.2.3).

- We compare the programming effort required for OpenMP and MPI for the concordance benchmark. In MPI significant effort is expended to specify the message passing coordination and hence, although the MPI program is shorter, the OpenMP program is more readable and easier to develop (Section 7).

2 Languages and Libraries

This section presents and compares the MPI and OpenMP programming models.

2.1 MPI

Message passing is a common parallel programming paradigm where processes with private memory communicate by exchanging messages. Using a standardised message passing library with a standard programming language, e.g. ANSI C with MPI provides portable parallel code. PVM [43] was an early standard library, and MPI [38] has emerged as the dominant standard.

When an MPI program starts, several processes are initiated on a specified number of processors. Since there is no single shared memory, these processes communicate by sending and receiving messages through network connection [44]. At the simplest level, an MPI program uses a master process that sends off work to worker processes, which receive the data, perform tasks on it, and then return the results to the master process which combines all the results [39]. Different modes of communication are supported by MPI, e.g. one-to-one communication and collective communication [38, 45].

MPI lends itself to virtually all forms of distributed-memory. It provides the programmer with portable, flexible and efficient parallel applications. However, parallelising an application using MPI is not an easy process. The programmer must be involved in each aspect of the coordination such as specifying how the work is to be divided among the processing elements, balancing the work to be done, managing the communications between the processors, etc. Moreover, with MPI there is no way to separate the communications and the computation parts of the code, as there is with other high level parallel programming languages.

2.2 OpenMP

Explicit parallel programming like message passing requires the programmer to design the program with considerable care and to add significant amounts of code to coordinate parallel execution. There are a number of parallel programming models designed to reduce the burden on the programmer. One such approach is to provide pragmas, or special comments, that direct a sophisticated language implementation that introduces and manages the parallelism. This is the approach taken by OpenMP [11] which is a de facto standard application programming interface (API) used mainly with shared memory architecture to provide parallel applications [11, 44].

OpenMP consists of a set of compiler directives, supporting library routines and environment variables to specify parallelism, and program runtime characteristics [44]. When the program is first executed, it is referred to as a process. This process will be allocated a memory space. Since OpenMP uses a thread paradigm, the process is initiated with a single thread which works as

the master. When the master reaches the parallelised region, a team of workers is created and these continue working together. All these threads share the same memory and therefore there is no need to exchange messages. An implicit barrier synchronisation mechanism is used, so no thread is able to progress beyond the end of the parallelised region until all the other threads in the team reach the same point. After that is achieved, all threads in the team will terminate, and the pre-existing master will continue executing the program until it encounters another parallel directive. This model of parallel programming is known as the fork/join model.

Several factors have contributed to the success of OpenMP. It is simple to learn and to use requiring little programming effort. Moreover it provides high-performance applications that are able to run on different shared memory platforms with different numbers of threads. Furthermore, as a result of being a directive-based approach, the same code can be developed on a single processor as well as on multiprocessor platforms, where in the former, directives are simply considered as comments and therefore are ignored by the compiler and successfully implemented sequentially [11, 12].

An important advantage of OpenMP is that it allows parallelisation to be carried out incrementally. By starting from a sequential program, the programmer simply needs to add the directives which express the parallelism [42, 41]. However, there are several issues related to parallel programming using OpenMP which programmer needs to consider [44] such as load balancing, scheduling, and race conditions.

Figure 1 contrasts the primary communication models supported by OpenMP and MPI.

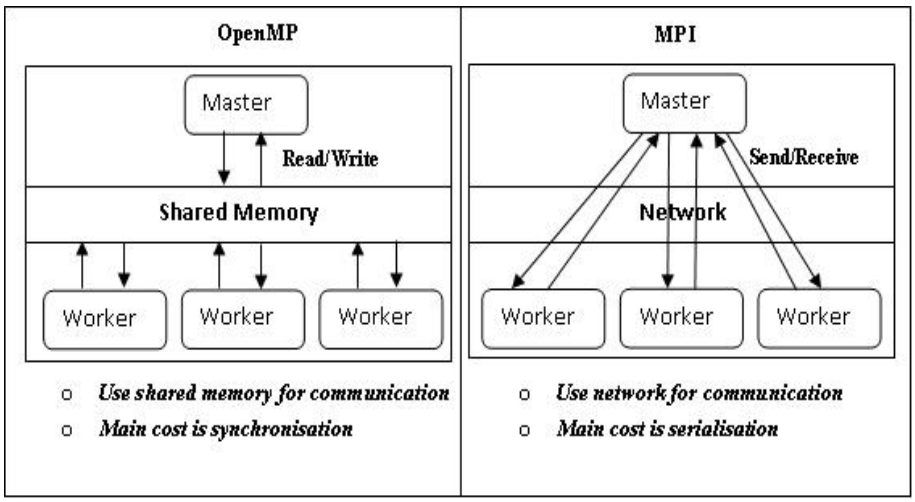


Figure 1: OpenMP and MPI Communication Models

3 The Concordance Benchmark

This section specifies the concordance problem including the expected input and required output. The problem is identical to the first SICSA challenge problem [2], except that our programs record pages rather than start indices.

The problem: Given a text file of N words with page breaks, construct a concordance listing the pages on which every word occurs, and every sequence of words from length 1 to N-1 occurs.
Input: The length of the sequence N, and the file name.
Output: Concordance listing the pages as shown in Figure 2. **The properties of the input:**

Sequence	Frequency	Pages
A	2383	2 , 3 , 5 , 6 , 7 ,...
A AN	1	244
A AN ANGEL	1	244
ALLAH	2988	10 , 11 , 12 ,....
.....		

Figure 2: Example of the Program Output

A text file with no size limitations. The file may include some special symbols, commas, and numbers which should be excluded from the sequence. The length of the sequence should be a positive integer number.
The required properties of the output: The output sequences should only include words. The output should print all possible sequences with lengths from 1 to N. The program should print the sequence, the number of occurrences, and list the pages where the sequence occurs.

4 Experimental Infrastructure

The concordance benchmarks are developed in ANSI C, with gcc 4.1.2 OpenMP, and with MPICH 1.2.7p1, and MPICH2 1.2.1p1. The programs are measured on a common multi-core architecture, namely an eight-core machine comprising two Intel Xeon 5410 quad-core processors, running at 2.33 GHz, with a 1998 MHz front-side bus 6144 KB and 8GB RAM running under CentOS release 5.5.

To minimize the effects of variability the data reported is the median of 3 executions for all experiments and the measurements are made on up to 8 cores. The reported speedup is an absolute speedup that is calculated as Sp = T (1) / T (p), where T1 is the run time of the sequential program and T(p) is the runtime on P processors. The efficiency is calculated as Ep = T(1) / (P * T(P)). In addition, for profiling the benchmarks, the ompP 0.7.1 profiling tool is used for profiling the OpenMP version and the mpiP 3.2.1 profiling tool is used for profiling the MPI version. For all the experiments reported, two samples of files are used with different sequence lengths, as follows:

(i) The first text file (E1) is about 18 MB and the sequences are of up to 10 words.

(ii) The second text file (E2) is about 1 MB and the sequences are of up to 50 words. Long sequences increase the computation granularity.

The work presented preceded the SICSA concordance challenge, and hence the sequential baseline is the C program developed by the authors as specified in the previous section, rather than either the C or Haskell SICSA reference implementations [2]. The E2 dataset is a similar size to the datasets used in the SICSA multicore challenge (3Kb-5Mb) [2], but E1 is significantly larger. Moreover E1 and E2 contain page breaks to meet the specifications in Section 3.

5 Design and Implementation

5.1 Sequential Design and Implementation

This section outlines the key algorithm and data structure selections for the sequential concordance program.

(i) *Data Structure*
There are a number of different ways to organise data, e.g. linked lists, hash tables, trees, stacks and queues. Since the purpose of the program is to read the text file and to construct a concordance for each sequence, as well as to search for the pages where each sequence occurs, an efficient data structure is needed that will be able to maintain a large amount of data as well as facilitate searching and the updating of information. To meet these specifications, a hash table is used to provide fast insertion into, and searching of, the concordance [48]. A single linked list is also used to store the page numbers since how many times each sequence occurs and on which pages will not be known in advance. Figure 3 represents the hash table, which consists of the key representing the sequence, the value representing the frequency of use, and the list of pages on which the sequence occurs.

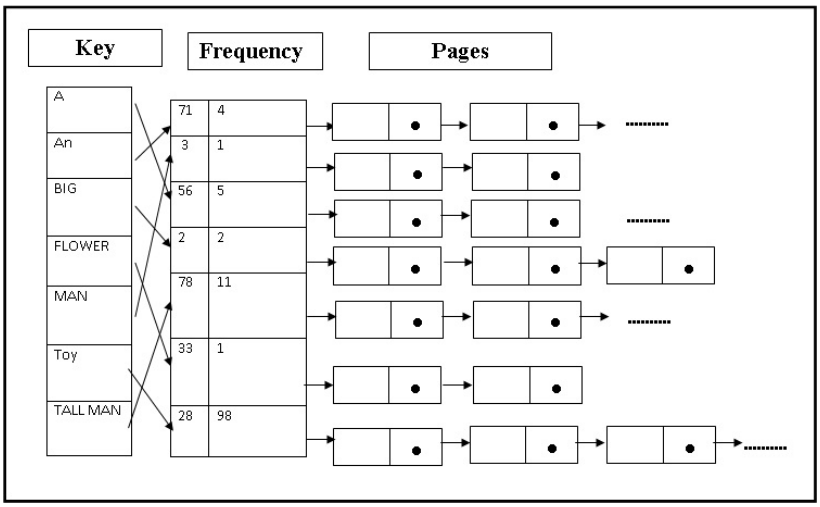


Figure 3: Hash Table Representation

(ii) *Sequential Algorithm* The sequential program is designed to be computationally cheap. An interesting problem encountered when designing the sequential algorithm was how to find sequences across page boundaries. This was solved simply by treating the page

marker as a space and increasing the page numbers. Figure 4 depicts the sequential program which calls the function construct_concordance as presented in Algorithm 1.

a- Read the maximum sequence length (N) and the file name.
b- open the text file .
c- while !EndOfText do
- Read one word to the buffer by reading one character each time and parsing.
- Set the current page number.
- call ***construct_concordance*** function (Algorithm 1).

d- sort the hash table.
e- print the hash table.

Figure 4: Sequential Program

```
for i ← 0 to N do
    concatenate the read word to form the sequence of length i;
    lookup the hash table for the current sequence;
    if Exist(sequence) then
        increment frequency;
        while !EndOfPageList do
            search for the current page number;
            if Exist(page number) then
                break the loop ;
            else
                Create new node representing the new page;
                Add this node at the beginning of the pages list;
            end
        end
    else
        insert the new sequence in the hash table with frequency 1;
        create new linked list representing pages;
        add the current page node at the beginning of the pages linked list ;
    end
end
```

Algorithm 1: Construct_Concordance Function

The hash table is implemented in C using the Glib library [13]: an open source general-purpose library, that provides a large number of useful data types, macros, type conversions, string utilities, file utilities, a main loop abstraction, etc. It works on many UNIX-like platforms, Windows, OS/2 and BeOS.

For the pages, a single linked list is implemented which create a new node each time a sequence is found on a new page. Insertions are made at the head of the list, so the page numbers can be sorted by default in descending order.

5.2 Parallel Design

The parallel algorithm design focuses on the computationally intensive component, specifically the concordance construction. In particular our parallelisation assumes that constructing the concordance, i.e. parsing and checking for the existence of a sequence of words is a relatively expensive compared with operations like merging hash tables. Other computations could also be parallelised, e.g. sorting the hash table as well as the linked lists of pages, but as these use standard techniques like [45] we have focused on parallelising the concordance construction.

The parallel concordance benchmarks use a data parallel Single Program Multiple Data (SPMD) model as it is a suitable match for the problem, provides flexibility, and is widely used, e.g. in large-scale scientific applications [36, 6]. In SPMD, computations are assigned to workers using their ID, and parallelism is exploited with data decomposition. Even though SPMD is highly scalable, it requires significant programmer effort, e.g. coordinating the work and data distribution.

SPMD is strongly favoured for MPI as it is closely related to the message passing model [36]. Although OpenMP is typically used for fine grain loop-level parallelism, it also supports SPMD. One of the most important drivers for using SPMD with a shared memory paradigm is the difficulty of achieving high performance with OpenMP when using loop-level parallelism. There are many reports delivering excellent performance using SPMD with OpenMP, e.g. [33, 32].

5.3 Parallelisation Challenges

The parallel programs provide solutions for three important problems which are finding sequences across worker boundaries, load balancing, and calculating page numbers.

1. **Load Balancing.** Load balancing is achieved by dividing text into chunks and assigning these chunks to the workers. The size of each chunk is computed by dividing the file size by the number of workers and, since the remainder of the division is not large, the last worker is assigned the remainder of the text, if any. To minimise file reading costs each worker receives the offset and the size of the file to read, because the file is stored inside the same shared memory.

2. **Finding sequences across worker boundaries.** Sequences reflect a common problem with tiled data: sequences of 2 or more words may span the boundary of the data assigned to two workers. We adopt the standard solution of replicating the data in a ghost boundary for each chunk [36]. That is, each worker has an extra N-1 words from the next worker's data, with the exception of the last worker. Each worker generates sequences using N-1 words from the next worker's data, and the next worker will read the same N-1 words but will use these words to generate sequences with the following words N, N+1. Figure 5 illustrates.

   It is also sometimes possible to have the worker boundary in the middle of the word. In this case, the first worker will proceed by reading the word as if it is in its own chunk. When the next worker starts reading from the beginning of the boundary, which is in the middle of the word, the worker will read 1 more character before the boundary to

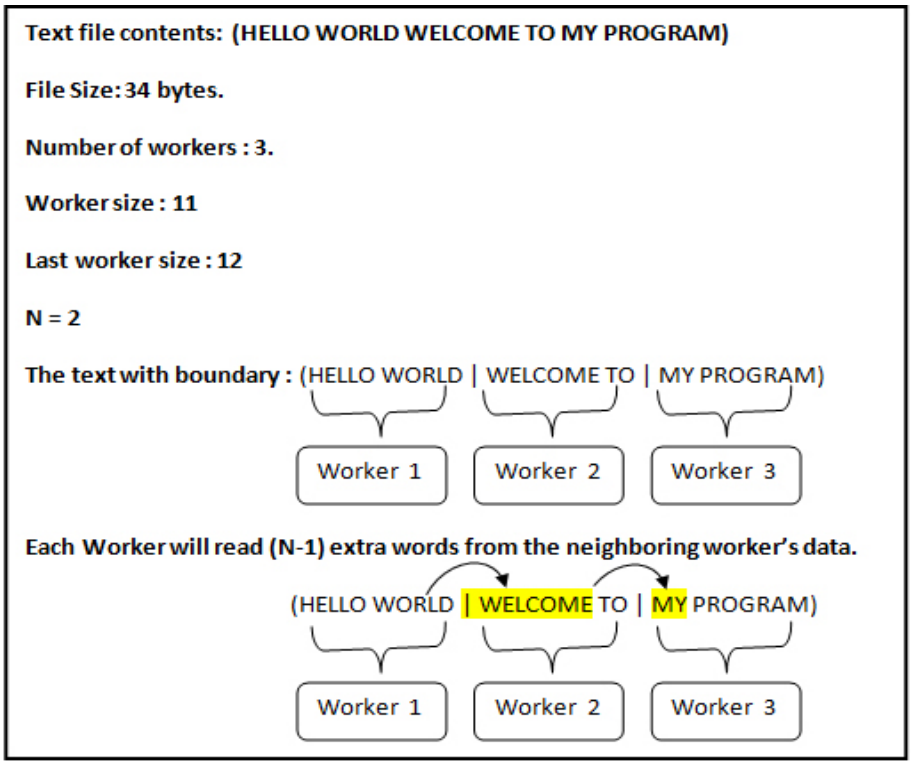


Figure 5: Finding sequences across workers boundaries

be sure that the boundary is in the middle, it will then skip the rest of this word since it has already been considered by the previous worker.

3. **Calculating the page numbers** since each worker will work on its allocated chunk of data, there is no way for each worker to know what page the previous worker reached to continue counting from this number. There are two possible solutions to this problem:

   (i) The master reads the whole file and counts how many pages the text file contains. The master can, therefore, send to each worker, in addition to the offset and the size of the allocated data, the page number that the worker has to start counting from .

   (ii) Assume a large integer number as a page boundary so that each worker will start counting the pages using this boundary and the worker ID. For example, let the page boundary be 1000 000 , so , the first worker will start from 1, 2, 3, and so on, and the second worker will start from page boundary * ID which is 2000000, 2000001, 2000002, and so on. Finally, the master sorts the hash table and calculates the correct page numbers.

Since reading the complete file using the master in advance is expensive, and we need to focus on constructing the concordance, the second solution is used.

5.4 OpenMP Design and Implementation

OpenMP Parallel Algorithm OpenMP supports the fork/join pattern, and in the concordance the master thread reads the file size and the number of available threads, and computes the size of the work for each thread. In the parallel region, the master thread will fork to the number of specified threads, and each will have the offset and size of file to work on. Since the memory is shared, all threads have access to the same hash table, as it includes the concordance results which need a synchronisation mechanism to control reading from or writing to it. At the end of the parallel region, the group of threads will be joined to one thread, which will proceed to the sorting of the hash table and printing the final results. Figure 6 shows the parallel OpenMP program.

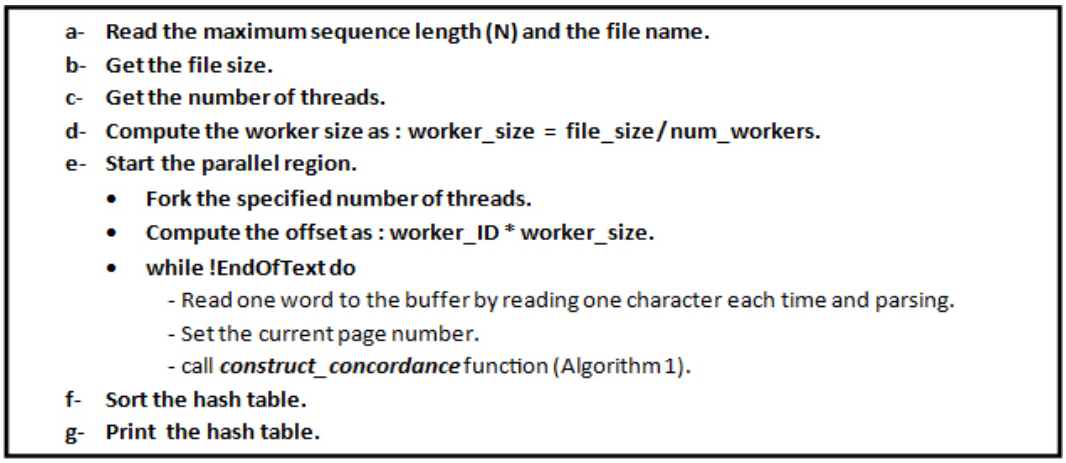


Figure 6: OpenMP Program

OpenMP Implementation OpenMP is implemented in a Single Program Multiple Data (SPMD) model by spawning the specified number of threads in the parallel region. Each thread uses its id value, which has private scope for the purpose of specifying the area of a text on which the thread has to work. This is entirely based on the OpenMP *parallel* directive, which encloses the parallel region where a set of spawn threads execute concurrently, leading to coarse-grain parallelism [11, 27].

Since OpenMP is a shared memory model, the hash table is shared among all threads so as to facilitate communication by reading from and writing to it. Consequently, access to the hash table needs to be explicitly organised among all threads to avoid conflicting access and incorrect data. It is highly recommended to use the atomic directive for synchronisation, as it uses hardware support to and ensure that the shared memory is atomically updated, resulting in a higher performance than when using the critical construct [4, 51, 12]. However, the atomic directive is not used since it has a set of restrictions over the critical directive. For example, it can only be applied to a critical section consisting of a single assignment statement with a special format, whilst the critical directive can be applied to a critical region with an arbitrary number of statements [12]. For the purpose of parallel concordance, we have mutual exclusion through a critical construct, which is used to protect the updating of the hash table by looking up, inserting and updating operations.

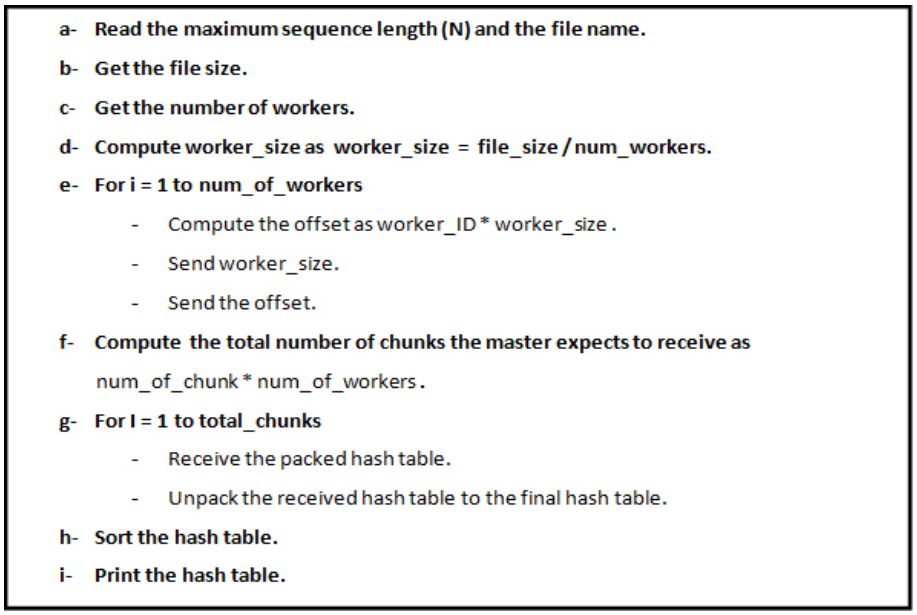


Figure 7: MPI Program (Master)

5.5 MPI Design and Implementation

MPI Parallel Algorithm MPI supports master/worker parallelism, and in the concordance the master reads the file size, the number of available workers, and computes the size of work for each worker. At the start of the program, the master process is responsible for computing the worker size and offset to indicate the part of text assigned to each worker, and it then sends information to the group of specified workers. As the program runs on a distributed file system (NFS), it is unnecessary to distribute the file among workers. Instead, each worker receives the offset and the size of file to work on, and reads the file. Since it is a message passing programming model, each worker has its own hash table, which includes the concordance results for corresponding chunks of data. At the end of the parallel work, each worker will convert its hash table into a suitable format for communication, and send it to the master, which is responsible for de-converting the received hash tables and combining them into one final hash table. Finally, the master will sort and print the final results.

Communication is critical for the performance of MPI programs. Being able to tune the message size is important to achieve the good performance, and this is controlled by a parameter.

Figures 7 and 8 show the parallel MPI programs for the master and the workers respectively.

MPI implementation MPI is implemented in the Single Program Multiple Data (SPMD) model using the master/worker parallel programming pattern. Blocking communications are used for sending and receiving the worker-size, the offset and the packed hash tables.

Since the hash table consists of more than one element which is not a suitable format for ordinary communication in MPI as it is stored in non-contiguous memory locations, it has to be converted to a suitable format for communication. To achieve this, MPI provides three different

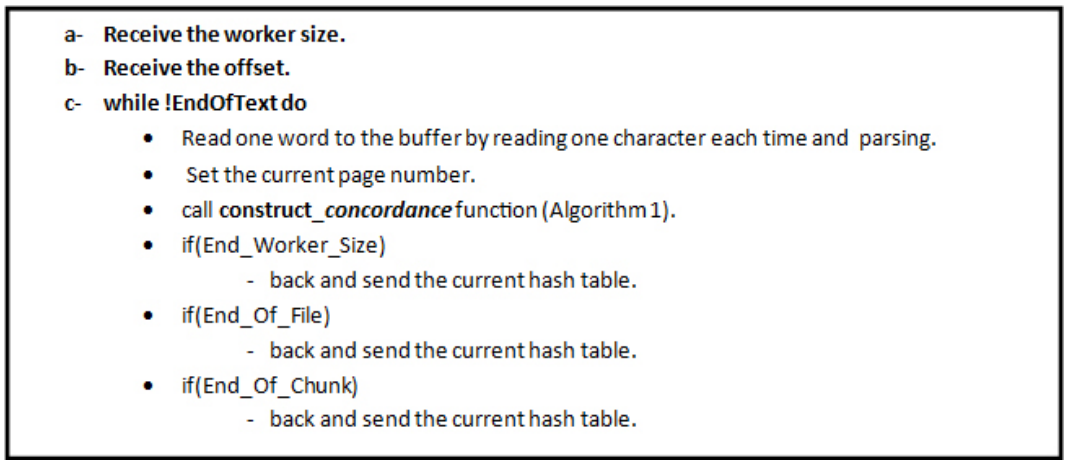


Figure 8: MPI Program (Worker)

techniques, each with its own positive and negative features. These are count parameter, derived data type and *MPI_ pack() /MPI_ unpack()* functions. The count parameter requires the data to be stored in contiguous memory locations [39, 40] and the derived data type is too complicated to implement; in addition, many of its implementations perform poorly [10, 50]. For this reason, the third mechanism, which is the pack and unpack functions is used. To send the hash table, the *MPI_ pack()* function is responsible for packing the entire hash table to a buffer, and then sending the buffer. When the buffer is received, the master will unpack the buffer using the function *MPI_ unpack()* to the final hash table.

6 Evaluation

6.1 OpenMP Performance

This section describes the performance of the initial OpenMP version of the concordance benchmark. Timings, scalability, the runtime profiles of the parallel region, and the overhead analysis of this benchmark are reported. In addition, performance tuning as well as reduction in different kinds of overhead are discussed. Finally, the final results of the tuned benchmark are presented.To accurately time the concordance construction part of the benchmark the *omp_ get_ wtime()* function is used.

6.1.1 OpenMP Preliminary Results

Table 1 and Figure 9 show that the runtimes of the initial OpenMP implementation are greater than for the sequential implementation, and these get worse as more cores are added. We are not surprised by these results, suspecting that the issues are high synchronisation overheads and excessive sequentialisation. That is, while one thread executes within the critical section protecting the hash table, many other threads are blocked. Moreover, the `critical` construct is the most expensive synchronisation construct in OpenMP [31, 11, 51]. This time is not only the time that each thread has to wait before the critical region, but also

Table 1: Runtime for Initial OpenMP for E1

No. Cores	Runtime(s)
1	48.4
2	42.7
3	47.0
4	52.1
5	56.7
6	61.8
7	74.2
8	82.6
Sequential Runtime	39.0

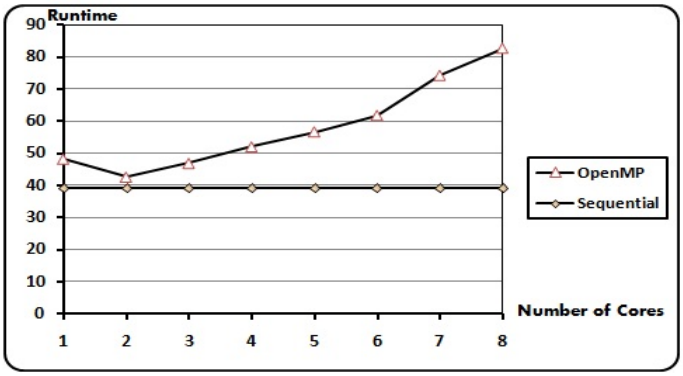


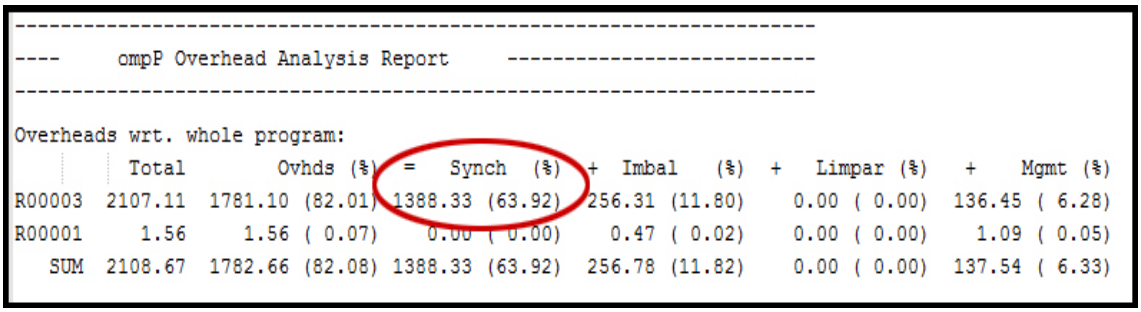
Figure 9: Parallel Runtime for the Initial OpenMP for E1

the time for executing the construct. Furthermore, the abstraction provided by the OpenMP directives hides the majority of the implementation details from the programmer such as creation and destruction of the team of threads. As a result, the performance of many OpenMP applications is significantly effected by the compiler and runtime system. Experiments studying the performance characteristics of OpenMP constructs show that another source of overhead in OpenMP applications results from using the *PARALLEL* directive, which has the largest overhead of the OpenMP directives [21].

6.1.2 OpenMP Profiling Results

OmpP [22] profiling is used to analyse, and hence improve the OpenMP performance, and specifically to analyse the overheads for each parallel region as well as for the whole program. The ompP overhead analysis report presented in Figure 10 shows four overhead categories. These are Synchronisation, Imbalance, Limited Parallelism and Thread Management overhead [22, 3, 23]. As expected, the synchronisation overhead is the dominating overhead affecting the increase in the runtime of the initial OpenMP program, with 63.92% of the execution time.

The report shows that there is no limited parallelism overhead. It also shows that the imbalance overhead is about 11.8%, which is difficult to reduce at this stage, as it relates to the amount of work allocated to each thread. Since the text in the file is irregular, some threads



```
------------------------------------------------------------------
----     ompP Overhead Analysis Report     ----------------------
------------------------------------------------------------------

Overheads wrt. whole program:
        Total        Ovhds (%)  =   Synch  (%)  +  Imbal   (%)  +   Limpar (%)   +    Mgmt (%)
R00003  2107.11  1781.10 (82.01) 1388.33 (63.92)  256.31 (11.80)    0.00 ( 0.00)  136.45 ( 6.28)
R00001     1.56     1.56 ( 0.07)    0.00 ( 0.00)    0.47 ( 0.02)    0.00 ( 0.00)    1.09 ( 0.05)
   SUM  2108.67  1782.66 (82.08) 1388.33 (63.92)  256.78 (11.82)    0.00 ( 0.00)  137.54 ( 6.33)
```

Figure 10: ompP Overhead Analysis Report

may have a portion of text that contains a higher proportion of frequently used English words than others or lots of spaces, comma, or symbols. As a result, they may spend more time gaining access to the shared hash table related to these common words or editing the text. This is clearly related to the text file and not to the load balancing mechanism.

Using the *nowait* clause could help to reduce the time spent on implicit barrier synchronisation at the end of the parallel construct [22]. Unfortunately, the *nowait* clause can only be applied for loop-level parallelism. The last kind of reported overhead is thread management overhead, about 6.28%, which is related to the implementation of the underlying runtime system [11].

6.1.3 OpenMP Performance Tuning

The key improvement here is the reduction of the overheads associated with synchronising access to the hash table.

1. **Dividing the Hash Table** One way of reducing the synchronisation overhead is to divide the hash table into multiple hash tables, and to use a different lock for each one. This approach ensures that no two threads are updating the same part of the hash table simultaneously, while allowing them to update different hash tables at the same time and hence significantly reduce the waiting time [11, 46]. Consequently, as experimental results show, dividing the hash table results in up to a 92% synchronisation overhead reduction, as well as more than 7 times performance improvement of the OpenMP concordance benchmark.

   Table 2 shows the number of hash tables used, the percentage of synchronisation overhead involved and the speedup on 8 cores. However, the price to be paid for the improved performance is the programming effort required to divide the hash table and explicit access management of the large number of hash tables. Moreover, it yields a code which is long, difficult to debug, and error prone. These results prove that the SPMD programming style can successfully achieves a high performance, but that it requires a considerable programming effort to explicitly manage the memory, coordinate the parallel work and reduce the synchronisation overhead involved.

2. **Increasing the Computation Size**
   Applying the tuned OpenMP version to the second experiment data set, with a file size of

Table 2: Reducing Synchronisation Overhead by Dividing the Hash Table

Number of Hash tables	Synchronisation Overhead	Speedup on 8 cores
1	64.0%	0.6
8	27.7%	1.7
54	21.6%	3.3
364	12.5%	4.2
657	4.9%	4.4

1MB and N = 50 shows a significant increase in the performance, as reported in Table 3. The computation/communication ratio has improved as a result of having more parsing and sequence forming work.

6.1.4 OpenMP Final Results

Table 3: Final OpenMP Runtime, Absolute Speedups, and Efficiency for E1 and E2

OpenMP						
	E1 (N=10, 18MB)			E2 (N=50, 1MB)		
No. Cores	R(s)	S	E	R(s)	S	E
1	51.3	0.7	0.76	54.2	0.8	0.85
2	30.3	1.2	0.64	28.8	1.6	0.80
3	22.2	1.7	0.58	20.1	2.3	0.74
4	18.3	2.1	0.53	15.7	2.9	0.86
5	15.7	2.4	0.49	12.9	3.6	0.72
6	13.9	2.8	0.46	11.1	4.1	0.69
7	12.7	3.0	0.43	9.8	4.7	0.67
8	11.7	3.3	0.41	8.7	5.3	0.66
Sequential Runtime	39.0			46.5		

Figure 11 depicts the measured runtime with the number of cores compared to the sequential version for the final OpenMP implementation for Experiment 1 and Experiment 2.

The results show a great improvement in the performance of the OpenMP code. This has been achieved by dividing the hash table into over 650 hash tables, thus reducing the synchronisation overhead to 4.9Figure 12 depicts the speedup for the final OpenMP implementation of Experiment 1 and Experiment 2.

6.2 MPI Performance

In this section, the performance of the initial MPI version of the concordance benchmark on an eight-core machine is presented. Timings, scalability, runtime profiles of the parallel work, and statistical analyses of all MPI functions of the benchmark are reported. Performance tuning as well as different ways of reducing overheads are also discussed. Finally, the final results of

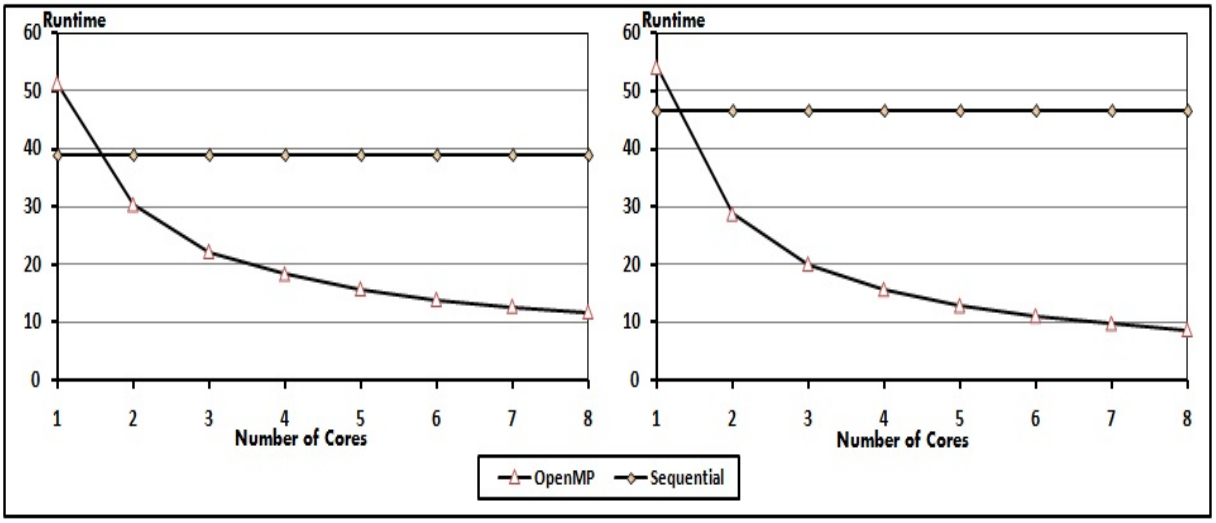


Figure 11: Parallel Runtimes for the Tuned OpenMP Version for E1(left) and E2(right)

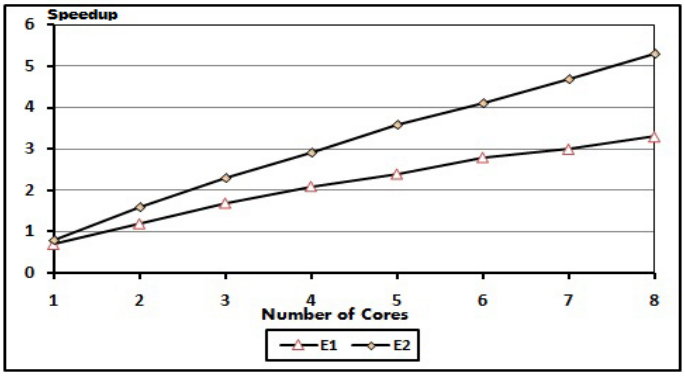


Figure 12: Parallel Speedup for the Tuned OpenMP Version for E1 and E2

the tuned MPI benchmark are presented. To obtain accurate time measurements, the *MPI_Wtime()* function is used.

6.2.1 MPI Preliminary Results

Due to the master/worker model used in MPI, it is not feasible to run the MPI program with a single core, since the design of the MPI program implies having one master responsible for sending work and combining results, but not in the computational work.

Unfortunately, the performance of the initial MPI version is disappointing, as can be seen in Table 4. We obtained a significant reduction in the run time as the number of cores increased, but the performance of this MPI implementation was much worse than the sequential version, which had a runtime of 39.0 seconds.

Table 4: Runtimes for Initial MPI for E1

No. Cores	Runtime(s)
2	101.7
3	52.0
4	48.3
5	45.6
6	47.4
7	46.7
8	46.8
Sequential Runtime	39.0

Figure 13 shows the large gap between the performance of the MPI version and its sequential counterpart. Several factors have a negative effect on the performance of MPI on shared

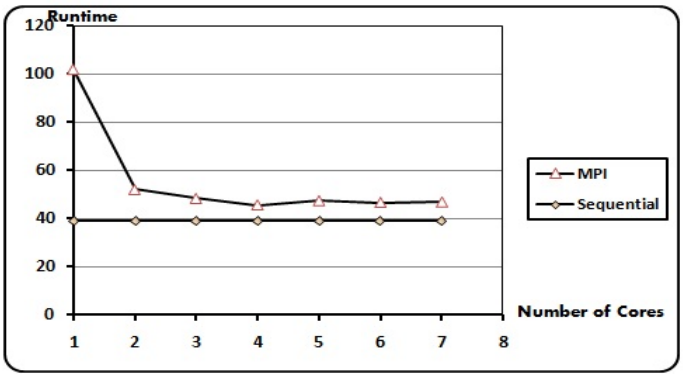


Figure 13: Parallel Runtime for the Initial MPI for E1

memory architectures. Balaji et al. [5] studied different kinds of non-data communication overheads within MPI implementation. Some of these might directly affect the performance of the MPI version of the concordance benchmark. For example,

- *MPI stack overhead:* increases proportionally with the message size as a direct result of the protocol switching [25]. There are three different protocols for message transferring, short, for less than 1 kb, eager, for up to 128 kb,and rendezvous for larger messages.

- *Tag and Source matching overhead* Messages in MPI are classified according to their attached tags. When the message is received, a search is carried out to identify it from the queue of posted receive requests. Problems of scalability may occur if the queue grows too long.

- *Unexpected messages Overhead* When attempting to receive required messages, the receiver treats all messages that have previously been sent as unexpected and they are put in a queue in the MPI stack until they are dealt with. This process can involve copying of data that can result in added overhead.

- *Communication Protocol of the MPI Implementation* With the MPICH1 implementation, Miao et al. [37] returned a poor performance for MPI on shared memory architecture to the double memory copying involved as it increases overheads.

6.2.2 MPI Profiling Results

The MPI version is profiled using the lightweight mpiP profiling library [1]. which is designed specifically for MPI applications. Using mpiP results in far less overhead and less data than using tracing tools, as it limits itself to collecting task-local statistical data regarding MPI functions. The only communication it uses occurs while a report is being generated, which is generally at the end of an experiment, and when merging all the results of the different tasks into a sole output file. The most important section of the mpiP report is the overview of the top twenty MPI call-sites that consume the most aggregate time within the application. Figure 14 shows part of the mpiP profile for the first experiment, and indicates that communicating the hash table is consuming about 67.35% of the application time. This indicates that the performance of the MPI version is affected by communication overheads.

```
---------------------------------------------------------------------------
@--- Aggregate Time (top twenty, descending, milliseconds) ----------------
---------------------------------------------------------------------------
Call                 Site       Time    App%    MPI%     COV
Send                   17   4.13e+06   67.35   72.29    0.06
Barrier                10   4.74e+05    7.73    8.30    0.72
Unpack                  8   2.39e+05    3.90    4.19    0.00
```

Figure 14: mpiP Time Analysis Report

The MPI version is implemented on a shared memory machine using MPICH1 implementation and, the MPICH1 implementation of MPI does not take full advantage of shared memory for communication [37]. This is because, to communicate data, a double copying system is used to copy data from the system buffer, and therefore, obviously the message passing from the sender buffer to the receiver buffer takes longer [37].

6.2.3 MPI Performance Tuning

We have explored several alternatives to improve the performance of the MPI version, as outlined below.

1. **Converting to Non blocking Communication** In theory non-blocking communication allows some message latency to be hidden by overlapping communication with computation [20]. *MPI_Send()* is changed to *MPI_Isend()*, and managing *MPI_Request()* is used. In addition, calling for *MPI_Wait()* is used to wait for the completion of the message receiving. Non-blocking communication allows part of the message latency to be hidden by overlapping the communication with the necessary computation [20]. However, the non-blocking program took longer due to the time needed to wait for message completion. The mpiP profiling report shows that *MPI_Wait()* function is consuming as much as 72.46% of the application time. Balaji et al. [5] explained that this is because of the overhead added by the managing *MPI_Request()*, which is required to allow the non-blocking operation to be completed. Allocation, initialisation and queuing / de-queuing are required for these requests, within the implementation of MPI for each non-blocking operation, leading to the increase in the overhead [29].

2. **Compressing the Messages**
   To reduce the communication volume, messages are compressed before they are sent to the master. For this purpose, the *Zlib* library is used [24]. The *Zlib* compression library offers in-memory compression and decompression functions, which include integrity checks for the uncompressed data. This means that each worker will pack the hash table, compress it and send it to the master. When receiving the message, the master will uncompress the hash table, unpack it and build the final hash table Unfortunately, no improvement is achieved by using compression, because the time used to compress the data outweighs the benefit achieved by reducing the message size, leading to an increase in the communication time, which is about 70% of the application time as reported by mpiP profiler.

3. **Communication through Intermediate Masters**
   This attempt to improve the MPI version aims to reduce the communication to the master. Since we have several workers competing to send their packed hash table to the master, this could account for the increase in sending time. As a result, in this attempt, we changed the way of sending the message through an intermediary master.Figure 15 shows how the messages are being sent. Each worker sends the message to another worker, who acts as an intermediary master, finally forwarding the message to the master. According to the profiling results, a very small improvement in the runtime is achieved, but the communication time is still high, reaching about 56% of the application time.

4. **Using Alternative MPI Library**
   The previous unsuccessful attempts to optimise the performance of the MPI, demonstrated the problem of MPICH implementation for the communication on shared memory architecture. The problem is reported by Miao et al. [37] as an effect of the

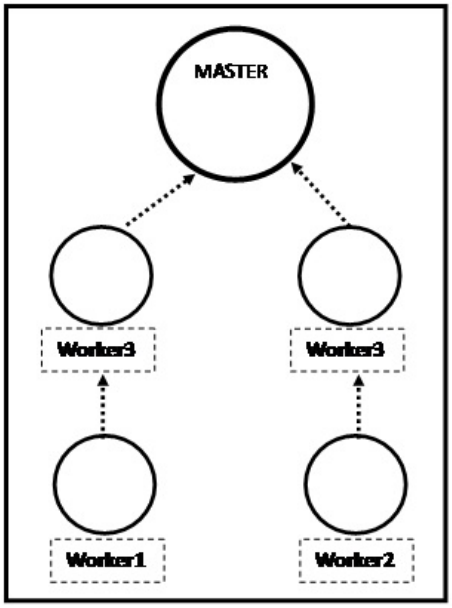


Figure 15: Communication through Intermediate Masters

double copying of data for communication. The MPICH implementation of MPI is an impediment to good performance, due to the fact that this implementation takes little account of the underlying architecture. The level of achievable performance for MPI depends on the implementation [7].

Extensive research has been carried out to improve MPI communication on shared memory architecture. Optimisation of MPI communication on shared memory architecture can be achieved by replacing double data copying protocol with single data copying protocol. This new communication protocol was designed and developed by Miao et al. [37] and has been demonstrated to outperform the MPICH communication protocol with lower latency.

A newer version of MPI implementation, namely MPICH2, is developed to solve several problems such as improving the communication on multicore architecture. MPICH2 is a high-performance and widely portable implementation of the Message Passing Interface (MPI) standard (both MPI-1 and MPI-2). MPICH2 aims to:

(a) Provide an MPI implementation to support different computation and communication platforms, including multicore architecture.

(b) Offer the possibility of cutting-edge research in MPI by using an easy-to-extend modular framework for other derived implementations [9].

Butinas et al. [7, 8] also developed and implemented a communication system called Nemesis as a communication subsystem for MPICH2, providing scalability, high performance shared memory and multi-network communication. It uses lock-free queues in shared memory, so that only one receiving queue is needed, avoiding increased overheads. Large Message Interface (LMI) is defined and added to the middle layer

of MPICH2 to support the different mechanisms for transferring large messages. This improves communication in shared memory architecture. Details of the design and implementation of Nemesis can be found in [7, 8]. Different mechanisms for data transfer on SMP systems have been presented and classified by Buntinas et al. [7]. They also evaluate the performance of these different mechanisms based on latency, bandwidth, and its suitability for different kinds of MPI messages.

The MPICH2 version 1.2.1p1 is installed and the MPI version of the concordance benchmark is tested by considering the MPI process as core, which is the common way of implementing MPI applications on multi-core architectures [9]. The results in Table 5 show a great improvement in the runtime(R) of our MPI version for experiment 1, and experiment 2 on 8 cores.

5. **Increasing the Computation Size**
   The results of Experiment 2 in Table 5 show how performance increases as a result of increasing the length of the sequence of words (N) and hence the computation to communication ratio.

Table 5: Runtimes for MPI version for E1 and E2 on MPICH1 and MPICH2

	E1 (N=10 , File size = 18MB)		E2 (N=50 , File size = 1MB)	
No. Cores	R(MPICH1)	R(MPICH2)	R(MPICH1)	R(MPICH2)
2	101.7	92.3	71.3	67.3
3	52.0	45.7	36.2	33.5
4	48.3	33.8	27.2	23.0
5	45.6	29.8	26.6	18.8
6	47.4	30.0	26.3	18.6
7	46.7	30.6	25.8	18.3
8	46.8	30.4	26.2	18.0
Sequential Runtime	39.0	39.0	46.5	46.5

6.2.4 MPI Final Results

For the final MPI concordance program with the two experimental data sets E1 and E2, Figure 16 shows the runtimes, Figure 17 the speedups, and Table 6 summarises the runtimes (R), speedups (S) and efficiencies (E). The results show an improvement in the performance of the MPI concordance using MPICH2. Moreover as the computation to communication ratio is increased from E1 to E2, the MPI performance improves. The best absolute speedup is modest: only 2.6 on 8 cores for E2.

Figure 17 depicts the speedup for the final MPI implementation of Experiment 1 and Experiment 2.

Table 6: Final MPI Runtime on MPICH2, Speedup, and Efficiency for E1 and E2

MPI						
	E1 (N=10, 18MB)			E2 (N=50, 1MB)		
No. Cores	R(s)	S	E	R(s)	S	E
2	92.3	0.4	0.21	67.3	0.6	0.34
3	45.7	0.8	0.23	33.5	1.3	0.46
4	33.8	1.1	0.28	23.0	2.0	0.50
5	29.8	1.3	0.26	18.8	2.4	0.49
6	30.0	1.3	0.21	18.6	2.5	0.41
7	30.6	1.2	0.18	18.3	2.5	0.36
8	30.4	1.2	0.16	18.0	2.6	0.32
Sequential Runtime	39.0			46.5		

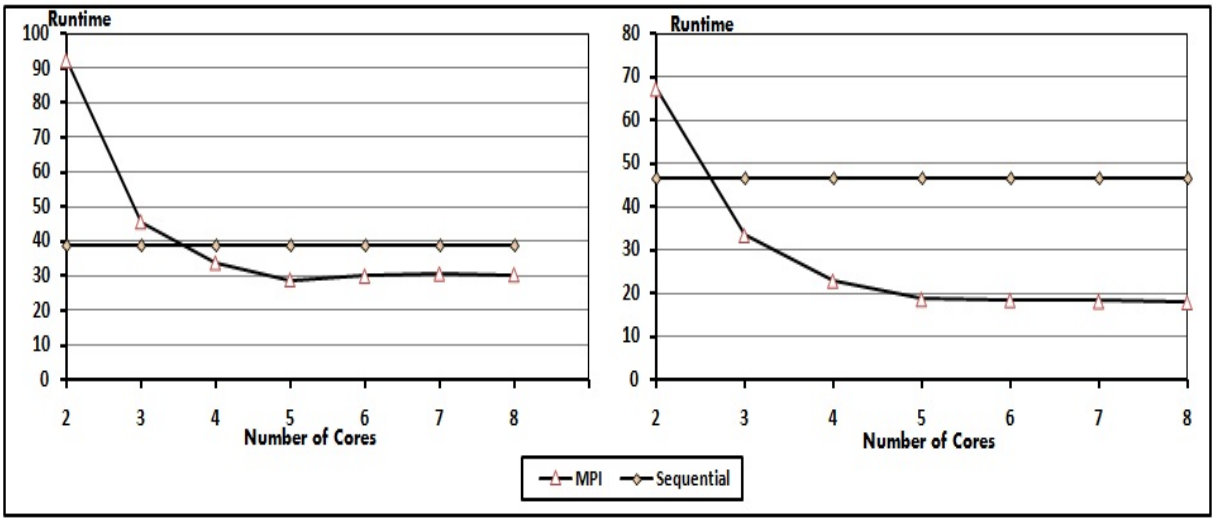


Figure 16: Parallel Runtime for the Tuned MPI Version for E1(left) and E2(right)

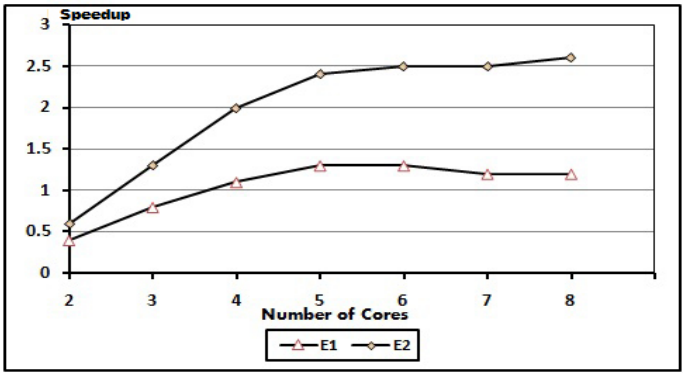


Figure 17: Parallel Speedup for the Tuned MPI Version for E1 and E2

6.3 Performance Comparison

Table 7 and summarizes the runtime (R), speedup (S), and efficiency measured for C+MPI and OpenMP on a different number of cores for experiment 2. Due to the master/worker model in MPI the MPI concordance must be executed on at least two cores. The table reflects the common case where the sequential C program is slightly faster than both the single core OpenMP or single worker (2 core) MPI program due to parallel overheads. Figures 18

Table 7: Comparative Runtimes, Speedups, and Efficiency for MPI and OpenMP for E2

E2 (N=50 , File size = 1MB)						
	OpenMP			MPI		
No. Cores	Runtimes	Speedup	Efficiency	Runtimes	Speedup	Efficiency
1	54.2	0.8	1.000			
2	28.8	1.6	0.940	67.3	0.6	0.34
3	20.1	2.3	0.898	33.5	1.3	0.46
4	15.7	2.9	0.862	23.0	2.0	0.50
5	12.9	3.6	0.839	18.8	2.4	0.49
6	11.1	4.1	0.813	18.6	2.5	0.41
7	9.8	4.7	0.789	18.3	2.5	0.36
8	**8.7**	**5.3**	**0.778**	**18.0**	**2.6**	**0.32**
Sequential Runtime	46.5			46.5		

and 19 show the runtimes and speedups for the final MPI and OpenMP implementations for experiment 2. OpenMP provides good performance with a maximum speedup of 5.3 on 8 cores for E2. Moreover performance continues to scale up to 8 cores. However, the price of this performance is the considerable programming effort of maintaining 650 hash tables to reduce the synchronisation overhead.

Even with an MPI implementation adapted for shared-memory architectures(MPICH2) MPI provides barely acceptable performance with a maximum speedup of 2.6 on 8 cores for E2. In fact the MPI version effectively ceases to scale beyond 5 cores, where it achieves a speedup of 2.4.

As anticipated, OpenMP showed excellent results, outperforming the MPI version for the parallel concordance benchmark. MPI pays a high performance penalty to communicate, and there is a clear advantage in using shared memory and avoiding the communication overheads when possible. Moreover, the scalability of the OpenMP version is superior to that of MPI, as indicated by the efficiency values for both. For all versions, the performance increases significantly as the computation size rises from E1 to E2.

Figure 19 depicts the speedup for the final OpenMP and MPI implementations of experiment 2 .

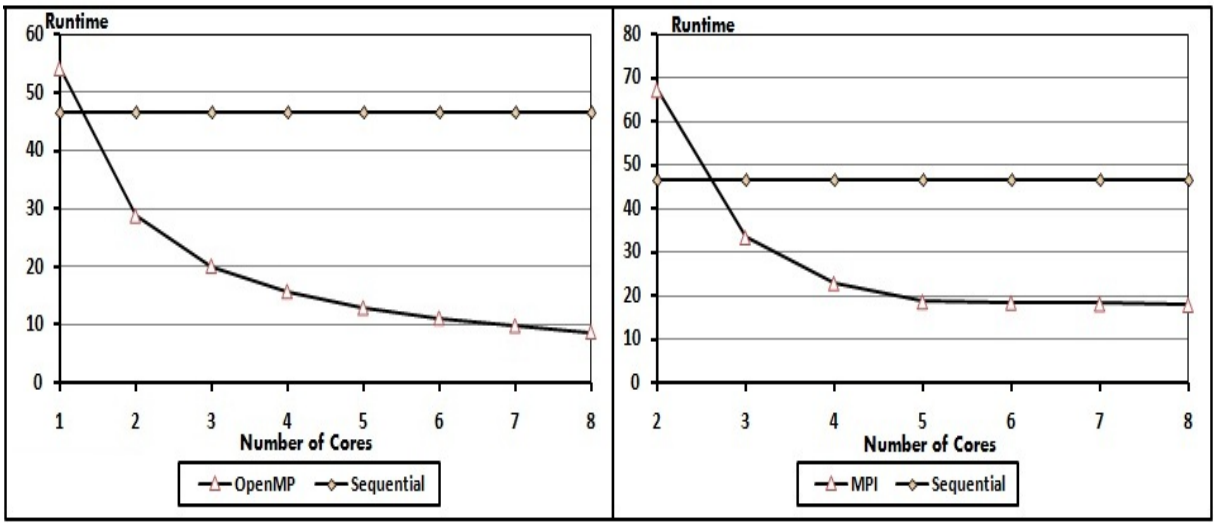


Figure 18: Parallel Runtime for the Tuned OpenMP (left) and MPI (right) for E2

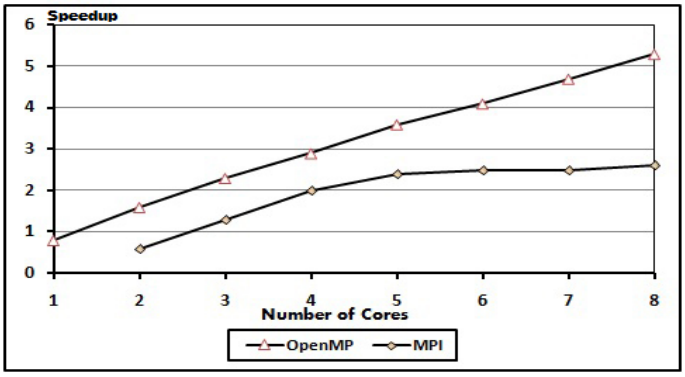


Figure 19: Parallel Speedup for the Tuned MPI and OpenMP Version for E2

7 Programming Model Comparison

This section briefly outlines OpenMP and MPI and discusses the implications of programming models for the Concordance benchmark. Substantial descriptions and critiques of both systems are available, e.g. for OpenMP [11, 15] and for MPI [39, 47], as are comparisons between the models [49, 34].

MPI is designed for distributed memory systems and message passing is conceptually straightforward, but requires the programmer to specify many low level coordination aspects like data serialisation, messaging, and coordination. Moreover these aspects are entwined with the algorithmic parts of the program. Hence when complex coordination is required the program can become hard to understand and debug and hence is error prone.

Adding the explicit serialisation, communication and synchronisation generally requires that a programmer write significantly more code than for a sequential program.

The shared-memory model for OpenMP is generally far less intrusive. OpenMP programs can be developed incrementally, by adding OpenMP pragmas to the sequential program.

OpenMP uses a fork/join model where the main program forks a gang of worker threads. Often relatively small numbers of pragmas and code changes are required to obtain the required performance. The key disadvantage is that OpenMP does not scale beyond a single shared-memory architecture.

When determining whether to use OpenMP or MPI, a programmer must ask whether an existing codebase will scale, or whether it is necessary to rewrite the code so that it will be more scalable to the multi-core architecture. It might seem logical that OpenMP would be the best choice to use with multi-core, because it is designed for shared memory. However, the scaling benefits of OpenMP or threads should be seriously evaluated before embarking on a lengthy rewriting of the code. It is entirely possible that using an existing MPI code base that performs adequately on multi-core will be perfectly satisfactory for a given application [17].

The one big difference between OpenMP and MPI is the OS process space. OpenMP programs run as a single process and the parallelism is expressed as threads whereas MPI starts one process per core using the mpirun np 8 ... command [17].

In OpenMP, communication occurs by means of shared memory. This means that threads share access to a memory location. MPI uses a software library to transmit data, or pass messages, from one process to another, each with its own memory space, from which the messages are copied. Message passing is actually memory copying, and MPI is designed so that messages can be passed between different processes on the same server, or on different servers [17, 19].

An important point to consider is the scalability of the application. Increasing the number of processors improves performance up to a certain point, after which, creating more threads or adding new processes may in fact damage performance [17]. MPI can be used with both shared memory and distributed memory architecture, offering great scalability. OpenMP, which only works with shared memory, also offers good scalability, but has some draw-backs compared to MPI. It requires a shared address space, and therefore, offers scalability to the number of CPUs within one SMP node [28]. The limited parallel scalability is generally considered to be the price that must be paid for ease of programming with a shared memory model. The distributed memory programming model is generally considered to be more difficult, but it is the only way to achieve higher levels of parallel scalability [11].

Distributed memory programming APIs like MPI are also available for the majority of shared memory multiprocessors. The portability of applications between different systems is a prime concern of software developers. This, combined with the lack of standard shared memory parallel API, means that, for the most part, application developers use the message passing model, even when target architectures are shared memory systems. A basic aim of OpenMP is to offer a portable standard parallel API that is designed specifically for programming shared memory multiprocessors [11]. MPI is highly portable and is specifically optimised for use with most hardware. In addition, it is more suitable for use with a wider range of problems than OpenMP. MPI can also handle larger amounts of data, as it can be distributed between multiple processors. This would be impossible with the memory space available on SMP [52, 17].

MPI may well be the best choice if the application is to be portable on clusters and SMP machines, but if more than eight or sixteen cores are not going to be used, it is advisable to choose OpenMP.

For the concordance implementation an SPMD programming model is used in both OpenMP and MPI. The OpenMP program forks the specified numbers of workers. The MPI uses a master/worker pattern, wherein the master sends work and combines results. During the parallelisation several challenges are encountered, including decomposition of data, choosing the best granularity, minimising the different kinds of overhead, debugging, and profiling for optimised performance. The aforementioned challenges are addressed differently, using either mid-level (MPI) or high-level (OpenMP) approach. Solutions to common problems were implemented in a similar way, such as finding sequences across worker boundaries, dividing files, and specifying the offset for each worker.

The OpenMP concordance program made some use of OpenMP's high-level parallelism. Critical regions provided safe synchronisation, and in contrast with MPI the hash tables did not need to be serialised or communicated. However to reduce synchronisation it was necessary to manage hundreds of hash tables, each with a separate critical region.

In the MPI concordance program the packing and unpacking of the hash table was implemented, since the hash table consists of different elements with different data types. Moreover, dividing the hash table into smaller chunks to improve communication granularity was tedious.

In summary the programming effort required to engineer the OpenMP concordance is less than for the MPI concordance as a direct benefit of the shared memory programming model. Although the tuned OpenMP version is about 19 times longer than the MPI version, approximately 90% of the tuned OpenMP code is additional hash tables to reduce synchronisation, and this could be shortened by placing the tables in a data structure.

8 Conclusion

This report presents the development of a new parallel benchmark, a concordance application, followed by a programming and performance comparison of MPI and OpenMP and their toolsets using the benchmark on a multicore platform.

The report begins with a brief introduction of OpenMP and MPI (Section 2). A description of the concordance problem is presented (Section 3). The experimental infrastructure, primarily an 8-core machine, is specified (Section 4). The design and implementation of the sequential concordance benchmark in C is presented, as well as the design and implementation of MPI and OpenMP versions (Section 5).

We presents a programming and performance comparison using the concordance analysing two data sets, describing how the initial parallel versions are profiled and benchmarked on the 8-core machine using appropriate profiling tools, ompP for OpenMP,and mpiP for MPI. We outline how synchronisation and other overheads are reduced to develop tuned versions of the programs (Sections 6.1, 6.2). Unsurprisingly communication overheads are the main performance issue for MPI on a shared-memory architecture, and the newer MPICH2 improves performance significantly (Section 6.2.3).

Several alternatives have been investigated to improve the MPI performance, including compressing data for communication, replacing blocking with non-blocking communication, and grouping data for communication. Unsurprisingly communication overheads are the main

performance issue for MPI on a shared-memory architecture. In particular the MPICH1 library copies the data twice, whereas the newer MPICH2 improves performance significantly by providing high performance shared memory communication (Section 6.2.3).

The performance of the OpenMP and MPI approaches are compared in terms of runtime, absolute speedup, and efficiency (Section 6.3). OpenMP shows excellent results, outperforming the MPI version. It is about twice as fast as the MPI version, and there is a clear advantage to using shared memory as it avoids the problem of communication overhead. OpenMP delivers better performance with both experiments.

- For experiment 1, the file size is 18mb, a speedup of 3.3 on 8 cores is achieved, while MPI achieved only 1.3.

- The overall speedup increased for both versions as the computation increased, in experiment 2 (file size = 1mb and N = 50), with 5.3 of the OpenMP and 2.6 of the MPI on 8 cores.

In MPI significant programming effort is required to pack and unpack the concordance hash table and to divide messages into suitable chunks. As a result, although the MPI program is shorter than the OpenMP counterpart, the OpenMP program is more readable and hence easier to develop (Section 7).

The outcomes confirm our expectations, namely that for this benchmark on a shared memory architecture OpenMP is a better option than MPI as the performance is better and the program is easier to develop. However unlike MPI programs, the OpenMP program does not scale beyond a shared memory platform.

9 Future Work

Both MPI and OpenMP are evolving quickly, and in future work it would be interesting to explore some of the new technologies, e.g. single data copying [37], or reducing communication time through message pre-fetching [30].

The parallelisation reported here has focused on the computing the intensive concordance construction phase. Overall performance could be improved by parallelising other phases of the benchmark, such as sorting the hash table. The benchmark could also be measured on larger multicore and NUMA architectures to investigate scalability.

Acknowledgements

The authors would like to thank Hans-Wolfgang Loidl for his technical support. Thanks also to Mustafa Aswad, Evgenij Belikov and Greg Michaelson for their valuable comments which significantly improved the paper. We also thank the participants in the SICSA multicore challenge for stimulating discussions.

This work has been partially supported by the European Union grants RII3-CT-2005-026133 “SCIEnce: Symbolic Computing Infrastructure in Europe”, IST-2011-287510 “RELEASE: A High-Level Paradigm for Reliable Large-scale Server Software”, and by the UK’s Engineering

and Physical Sciences Research Council grants EP/G055181/1 “HPC-GAP: High Performance Computational Algebra and Discrete Mathematics”.

References

[1] *mpiP: Lightweight, Scalable MPI Profiling ,Version 3.2.1.*

[2] *SICSA MultiCore Challenge*, Available at http://www.macs.hw.ac.uk/sicsawiki/index.php/MultiCoreChallenge.

[3] *The OpenMP Profiler ompP: User Guide and Manual,Version 0.7.0.*

[4] S. Akhter and J. Roberts, *Using OpenMP for Programming Parallel Threads in Multicore Applications: Part 3*, EE Times Design (2007).

[5] P. Balaji, A. Chan, W. Gropp, R. Thakur, and E. Lusk, *The Importance of Non-Data-Communication Overheads in MPI*, Int. J. High Perform. Comput. Appl. **24** (2010), 5–15.

[6] R. Bordawekar, J. M. del Rosario, and A. Choudhary, *Design and Evaluation of Primitives for Parallel I/O*, Proceedings of the 1993 ACM/IEEE conference on Supercomputing (New York, NY, USA), Supercomputing ’93, ACM, 1993, pp. 452–461.

[7] D. Buntinas and G. Mercier, *Implementation and Shared-memory Evaluation of MPICH2 over the Nemesis Communication Subsystem*, Proceedings of the Euro PVM/MPI Conference, Springer, 2006.

[8] D. Buntinas, G. Mercier, and W. Gropp, *Design and Evaluation of Nemesis, a Scalable, Low-Latency, Message-Passing Communication Subsystem*, Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (Washington, DC, USA), CCGRID ’06, IEEE Computer Society, 2006, pp. 521–530.

[9] Darius Buntinas, Guillaume Mercier, and William Gropp, *Implementation and Evaluation of Shared-memory Communication and Synchronization Operations in MPICH2 using the Nemesis Communication Subsystem*, Parallel Comput. **33** (2007), 634–644.

[10] S. Byna, X. Sun, R. Thakur, and W. Gropp, *Automatic Memory Optimizations for Improving MPI Derived Datatype Performance*, Proceedings of the 13th European PVM/MPI User’s Group conference on Recent advances in parallel virtual machine and message passing interface (Berlin, Heidelberg), EuroPVM/MPI’06, Springer-Verlag, 2006, pp. 238–246.

[11] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel Programming in OpenMP*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[12] B. Chapman, G. Jost, and R. Pas, *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*, The MIT Press, 2007.

[13] T. Copeland, *Manage C Data using the GLib Collections*, 2005.

[14] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to algorithms*, MIT Press, 2001.

[15] Matthew Curtis-Maury, Xiaoning Ding, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos, *An evaluation of openmp on current and emerging multithreaded/multicore processors*, Proceedings of the 2005 and 2006 international conference on OpenMP shared memory parallel programming (Berlin, Heidelberg), IWOMP'05/IWOMP'06, Springer-Verlag, 2008, pp. 133–144.

[16] Inderjit S. Dhillon, James Fan, and Yuqiang Guan, *Efficient clustering of very large document collections*, 2001.

[17] D. Eadline, *MPI on Multicore, an OpenMP Alternative?*, The Linux Magazine (2007).

[18] D. Eadline, *The Multicore Programming Challenge*, The Linux Magazine (2007).

[19] Eadline, D., *Comparing MPI and OpenMP*, The Linux Magazine (2009).

[20] A. Fachat and K. Hoffmann, *Blocking vs. Non-blocking Communication under MPI on a Master-Worker Problem*, 1998.

[21] N. Fredrickson, A. Afsahi, and Y. Qian, *Performance Characteristics of OpenMP Constructs, and Application Benchmarks on a Large Symmetric Multiprocessor*, Proceedings of the 17th annual international conference on Supercomputing (New York, NY, USA), ICS '03, ACM, 2003, pp. 140–149.

[22] K. Fürlinger and M. Gerndt, *Analyzing Overheads and Scalability Characteristics of OpenMP Applications*, Proceedings of the 7th international conference on High performance computing for computational science (Berlin, Heidelberg), VECPAR'06, Springer-Verlag, 2007, pp. 39–51.

[23] K. Fürlinger, M. Gerndt, and J. Dongarra, *Scalability Analysis of the SPEC OpenMP Benchmarks on Large-Scale Shared Memory Multiprocessors*, Proceedings of the 7th international conference on Computational Science, Part II (Berlin, Heidelberg), ICCS '07, Springer-Verlag, 2007, pp. 815–822.

[24] J. Gailly and M. Adler, *Zlib 1.2.5 Manual*, Tech. report, Jean-loup Gailly and Mark Adler, 2009.

[25] W. Gropp and E. Lusk, *A High-performance MPI Implementation on a Shared-Memory Vector Supercomputer*, Parallel Comput. **22** (1997), 1513–1526.

[26] Steffen Heinz and Justin Zobel, *Performance of data structures for small sets of strings*, Proceedings of the twenty-fifth Australasian conference on Computer science - Volume 4 (Darlinghurst, Australia, Australia), ACSC '02, Australian Computer Society, Inc., 2002, pp. 87–94.

[27] Alan J., *SPMD OpenMP versus MPI for Ocean Models*, Concurrency - Practice and Experience **12** (2000), no. 12, 1155–1164.

[28] G. Jost, H. Jin, D. An Mey, and F. Hatay, *Comparing the OpenMP, MPI, and Hybrid Programming Paradigms on an SMP Cluster 1.*

[29] H. Kasim, V. March, R. Zhang, and S. See, *Survey on Parallel Programming Model*, Proceedings of the IFIP International Conference on Network and Parallel Computing (Berlin, Heidelberg), NPC '08, Springer-Verlag, 2008, pp. 266–275.

[30] Jian Ke, Martin Burtscher, and Evan Speight, *Reducing Communication Time through Message Prefetching*, Intl. Conf. on Parallel and Distributed Processing Techniques and Applications, 2005, pp. 6–2005.

[31] J. A. Keane, A. J. Grant, and M. Q. Xu, *Comparing Distributed Memory and Virtual Shared Memory Parallel Programming Models*, Future Gener. Comput. Syst. **11** (1995), 233–243.

[32] G. Krawezik, *Performance Comparison of MPI and three OpenMP Programming Styles on Shared Memory Multiprocessors*, Proceedings of the fifteenth annual ACM symposium on Parallel Algorithms and Architectures (New York, NY, USA), SPAA '03, ACM, 2003, pp. 118–127.

[33] G. Krawezik, G. Alléon, and F. Cappello, *SPMD OpenMP versus MPI on a IBM SMP for 3 Kernels of the NAS Benchmarks*, Proceedings of the 4th International Symposium on High Performance Computing (Berlin, Heidelberg), Springer-Verlag, 2002, pp. 425–436.

[34] Graud Krawezik and Franck Cappello, *Performance comparison of mpi and openmp on shared memory multiprocessors*, Concurrency and Computation: Practice and Experience **18** (2006), no. 1, 29–61.

[35] D. Mallón, G. Taboada, C. Teijeiro, J. Touriño, B. Fraguela, A. Gómez, R. Doallo, and J. Mouriño, *Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures*, Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface (Berlin, Heidelberg), Springer-Verlag, 2009, pp. 174–184.

[36] T. Mattson, B. Sanders, and B. Massingill, *Patterns for Parallel Programming*, first ed., Addison-Wesley Professional, 2004.

[37] Q. Miao, G. Sun, J. Shan, and G. Chen, *Single Data Copying for MPI Communication Optimization on Shared Memory System*, Proceedings of the 7th international conference on Computational Science, Part I: ICCS 2007 (Berlin, Heidelberg), ICCS '07, Springer-Verlag, 2007, pp. 700–707.

[38] MPI Forum, *MPI 2: Extensions to the Message-Passing Interface*, Tech. report, University of Tennessee, Knoxville, 1997.

[39] P. Pacheco, *Parallel Programming with MPI*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[40] Peter S. Pacheco, *A Users Guide to MPI*, March 1998.

[41] R. Pas, *An Overview of OpenMP 3.0*, 2009.

[42] Keshav Pingali, *Parallel Programming Languages*, 1998.

[43] *Parallel Virtual Machine Reference Manual, Version 3.2*, University of Tennessee, August 1993.

[44] C. Quammen, *Introduction to Programming Shared-Memory and Distributed-Memory Parallel Computers*, Crossroads **12** (2005), 2–2.

[45] M. Quinn, *Parallel Programming in C with MPI and OpenMP*, McGraw-Hill Education Group, 2003.

[46] M. Randall and A. Lewis, *A Parallel Implementation of Ant Colony Optimization*, J. Parallel Distrib. Comput. **62** (2002), 1421–1432.

[47] Juan-Antonio Rico-Gallego and Juan-Carlos Díaz-Martín, *Performance evaluation of thread-based mpi in shared memory*, Proceedings of the 18th European MPI Users' Group conference on Recent advances in the message passing interface (Berlin, Heidelberg), EuroMPI'11, Springer-Verlag, 2011, pp. 337–338.

[48] S. Skiena, *The Algorithm Design Manual*, Springer-Verlag New York, Inc., New York, NY, USA, 1998.

[49] L. A. Smith, *Mixed mode mpi/openmp programming*, UK High-End Computing Technology Report (2000).

[50] B. Surendra, *Improving the Performance of MPI Derived Datatypes by Optimizing Memory-Access Cost*, In Proceedings of the IEEE International Conference on Cluster Computing, 2003, pp. 412–419.

[51] M. Süßand C. Leopold, *Common Mistakes in OpenMP and How to Avoid them: A Collection of Best Practices*, Proceedings of the 2005 and 2006 international conference on OpenMP shared memory parallel programming (Berlin, Heidelberg), IWOMP'05/IWOMP'06, Springer-Verlag, 2008, pp. 312–323.

[52] J. Theron, *MPI vs. OpenMP*, 2004.