Modeling and Predicting the Efficiency of Application Execution in Distributed Environments

AHMED M. AL KINDI a,*, KHAMIS N. AL GharBI b, GRAHAM NUDD c

a Department of Computer Science, College of Science
b Information Systems Department, College of Commerce and Economics
Sultan Qaboos University, P.O. Box 36, PC 123, Oman
c High Performance Systems Laboratory, Department of Computer Science,
University of Warwick, Coventry CV4 7AL, UK

Abstract: One of the main objectives in the study of performance prediction is to give researchers the option to select from a variety of distributed systems/resources based on what would optimise the performance of the application. This paper introduces an optimization scheme for modelling and predicting the efficiency of an application execution when run in a distributed environment. The results obtained from this research show that the suggested modelling and prediction technique can provide the same quality of information as a measurement process for application optimization, but in a fraction of the time.

Keywords: Performance Optimization, Dynamic Performance Prediction/Modelling, DFT, FFT, FFTW, PACE.

1 Introduction

The promise of Information Power Grid (IPG) is based on the concept of selecting distributed resources to perform a single or multiple computational tasks with performance optimisation in mind [1]. It relies on the availability of systems and the utilisation of performance information to guide application execution (i.e. communication and synchronization overhead of an application execution when run in a distributed environment [2]). Instead of writing optimized codes by hand, there have been several examples of automated code generators geared towards obtaining the maximum performance on different platforms. One such library is the Portable High Performance ANSI-C (PHiPAC), developed at Berkeley and intended for producing high performance matrix-vector libraries in the Basic Linear Algebraic Subprograms (BLAS) suite [3], [4], and [5]. In a similar way, an ‘adaptive’ application called The Fastest Furrier Transform in the West (FFTW) [6] uses automatically generated blocks of highly optimized C codes called codelets, to calculate a FFT in a minimum amount of time on any system [7].

This paper utilizes and models this ‘adaptive’ application and shows that efficiency of application execution in distributed environments can be predicted with reasonable quality of information. This research uses a performance evaluation tool developed by the University of Warwick, called Performance Analysis and Characterization Environment (PACE), which was developed as a modelling toolset for high performance and distributed applications. The work is an extension of the sequential performance evaluation techniques presented in [8] and [9] which led to further enhancement of the FFTW. The experiments and evaluations were conducted using machines with equal CPU and memory sizes (Intel Pentium 2.4 GHz and 512 MB RAM running under the Red Hat LINUX operating system). The next section introduces PACE while Section 3 discusses the FFTW package. These overviews are deliberately brief due to the nature of this work being conducted in these environments.

---

1 This work was conducted at the University of Warwick (UK) and funded in part by the Sultan Qaboos University, Oman
in part to space constraint. Interested researchers are advised to seek further information in literatures referenced at the end of this paper. Sections 4 through 7 present the implementation and results.

2 Performance Analysis and Characterization Environment (PACE)

PACE was developed as a modeling toolset for high performance and distributed applications. It includes tools for the definition and creation of the application model and application performance analysis [9]. It uses associative objects organized in a layered framework as a basis for representing each of a system’s components [10]. PACE supports any application that depends on message passing using MPI. In fact, any hardware or platform that uses these programming tools can be analyzed within PACE [11].

A key feature of the object organization is the independent representation of computation, parallelization, and hardware in a layered framework. The functions of the layers are:

- **Application Layer**: describes an application in terms of a sequence of subtasks. It acts as the entry point to the performance study, and includes an interface that can be used to modify parameters of a performance scenario.

- **Application Subtask Layer**: describes the sequential part of every subtask within an application that can be executed in parallel. An application might consist of a number of different subtasks, each representing the application’s parallel kernel. A combination of these subtasks by the application object forms a functionality model of the whole application.

- **Parallel Template Layer**: describes the parallel characteristics of subtasks in terms of expected computation/communication interactions between processors. This is done through the use of models in the hardware layer.

- **Hardware Layer**: collects system specification parameters, micro-benchmark results (e.g. atomic language operations), statistical models (e.g. regression communication models), analytical models (e.g. cache contention), and heuristics that characterize the communication and computation abilities of a particular system.

When application source code is available, PACE performs a static analysis of the code to produce the control flow of the application, operation counts, and the communication structure. A compiler translates C code to scripts that are then linked with an evaluation engine and the hardware models. The final output is a binary file that can be executed rapidly. The user determines the system/application configuration and the type of output that is required as command line arguments. The model binary performs all the necessary model evaluations and produces the requested results.

An MPI version of PACE is also available where it has the ability to measure and evaluate MPI-Send, MPI-Receive, and other MPI commands. MPI-PACE takes into account additional overhead that is incurred when sending and receiving the data from and to the processors. This communication/synchronization overhead is also modeled and predicted and then added to the overall computational cost of the evaluated application.

3 The Fastest Fourier Transform in the West (FFTW)

The FFTW is based on the divide-and-conquer approach of the recursive FFT. To illustrate how a DFT is computed using the recursive FFT, let us assume that the following polynomial:

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

is to be evaluated at the n complex roots of unity of degree n: \(\omega_0^n, \omega_1^n, \omega_2^n, \ldots, \omega_{n-1}^n\) (n is assured to be a power of 2).

Polynomial A can be used to represent any complex valued vector \(a\) where:

$$a = (a_0, a_1, \ldots, a_{n-1})$$

Then the results can be defined as follows:

$$y_k = A(\omega^n_k) = \sum_{j=0}^{n-1} a_j \omega_n^{kj}$$

where \(k = 0, 1, 2, \ldots, n-1\).

The DFT of vector \(a\) is the vector:

$$y = (y_0, y_1, \ldots, y_{n-1})$$

To compute the DFT of the vector \(a\), the array is divided into an even-index sub-array, and an odd-index sub-array. The two polynomials representing the divided arrays are hence as follows:

$$A_{n/2}^{(0)}(x) = a_0 + a_2 x + a_4 x^2 + \ldots + a_{n-2}$$

$$A_{n/2}^{(1)}(x) = a_1 + a_3 x + a_5 x^2 + \ldots + a_{n-1}$$

where \(x = \omega_n^k\).
\[ A^{[1]}(x) = a_1 + a_3 x + a_5 x^2 + \ldots + a_{n-1} x^{n-1} \tag{6} \]

This divide-and-conquer method is based on the following formula:

\[ A(x) = A^{[0]}(x^2) + A^{[1]}(x^2) \times x \tag{7} \]

Therefore, the computation of the DFT of \( a \) is now reduced to two steps:

- Evaluating the polynomials: \( A^{[0]}(x) \) and \( A^{[1]}(x) \) of degree \( n/2 \) at the \( n/2 \) complex roots of unity of degree \( n/2 \): \( \omega_n^0, \omega_n^2, \omega_n^4, \ldots, \omega_n^{n-2} \), and then applying formula (7).

Since each root occurs twice, the equations in (5 and 6) may now be recursively computed at the \( n/2 \) complex roots of unity of degree \( n/2 \). Therefore, an \( n \) element DFT computation is successfully divided into two \( n/2 \) element DFT computations. This forms the basis of the portable C package called the FFTW which computes one or multi-dimensional complex DFT. It is claimed that due to its self-configuring and dynamic nature, the FFTW can compute the transform faster than any other software. It requires a sequence of measurements to be made for its software components (codelets) on the target system, and then uses a combination of these codelets to determine how best to compute the transform of any given size. There are three essential components that make up the FFTW:

- **Codelets**: These are a collection of highly optimized blocks of C code used to compute the transform. The codelets are generated automatically. There are two types of codelets:
  - **Non-Twiddle**: used to compute small-sized transforms (\( N \leq 64 \)), and
  - **Twiddle**: used to compute larger transforms by the combination of smaller ones.

- **Planner**: The planner determines the most efficient way in which the codelets can be combined together to calculate the required transform. Each individual combination of codelets is called a plan. Typically dozens of plans are available for a specific FFT size.

- **Executor**: The execution of each plan is performed and measured by the executor. This results in the run-time cost of each plan on a specific system. From this, the best plan can be determined – i.e. the plan with the minimum execution time.

The initial stage of the FFTW operation involves the planner, which finds out the fastest way to compute the DFT of a given complex array. To illustrate the operation of the planner, an example of performing a 16-point FFT is considered. The planner uses the divide-and-conquer approach to partition the data set and to construct all possible combinations of codelets. The resulting instructions contained within the plan may look as follows:

\[
\begin{align*}
\text{FFT (16)} &= \text{Divide & Conquer (16,8)} \gg 8 \\
& \text{or} \quad \text{Divide & Conquer (16,4)} \gg 4 \\
& \text{or} \quad \text{Divide & Conquer (16,2)} \gg 2 \\
& \text{or} \quad \text{Solve (16)}
\end{align*}
\]

In this example, the planner has selected four types of instructions or sequences to send to the executor. One is to solve using a non-twiddle codelet 16, and the rest are to solve using a combination of twiddle and non-twiddle codelets (e.g. twiddle codelet 8 and a non-twiddle codelet 2 for the first instruction). To benefit from the recursive nature of the algorithm, solutions to individual instructions could also be stored in tables during this real-time computation for reusability.

The different alternatives that are available for the calculation using this approach in the FFTW result in different performance due, in part, to the use of the memory hierarchy including caches. For instance, the data required for computing an FFT of a small size may fit into the system’s primary cache, resulting in a better processing time, whereas the FFT of a larger size may not. Thus combinations of smaller FFTs may out-perform larger FFTs. By initially measuring all these combinations, a ‘best plan’ can be determined, which will be different on different systems.

The FFTW provides two parallelization techniques: Multithread FFTW and MPI-FFTW. Because PACE relies on MPI for its prediction analysis of parallel environments, MPI-FFTW is selected for use in this research. Further detail on the FFTW may be found at [http://www.fftw.org](http://www.fftw.org).

### 4 Modeling the FFTW

The measurement procedure required by the FFTW to select the best plan for a specific system can be replaced by a set of performance models. To
overcome the costly measurements of its initial stage, separate models are created to represent all individual FFTW codelets. The codelets are dynamically profiled and analyzed, and the models for the individual codelets are generated. The operation of the FFTW planner is then simulated in order to select the minimum predicted execution time of an appropriate set of codelet models. Finally, the predicted optimum plan is passed to the FFTW executor to compute the transformation.

The evaluation of each plan can be performed for a range of systems and problem sizes. The evaluation time for this process is faster than the corresponding real-time FFTW measurements. The validity of this optimization methodology depends on the selection pattern of the optimum solution (or best plan). To achieve this, the performance models require accurate prediction of every plan’s execution time.

The performances of six selected non-twiddle codelets are illustrated in Table 1 which shows their modeled performance compared to their real-time execution on a single Pentium machine (P64 stands for plan for FFT of size 64).

<table>
<thead>
<tr>
<th>Plans</th>
<th>PACE (µs)</th>
<th>FFTW (µs)</th>
<th>PACE / FFTW (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P64</td>
<td>1.68697</td>
<td>1.534912</td>
<td>109.9066</td>
</tr>
<tr>
<td>P32</td>
<td>0.695336</td>
<td>0.796448</td>
<td>87.30463</td>
</tr>
<tr>
<td>P16</td>
<td>0.277505</td>
<td>0.235214</td>
<td>117.9798</td>
</tr>
<tr>
<td>P8</td>
<td>0.107272</td>
<td>0.120674</td>
<td>88.89405</td>
</tr>
<tr>
<td>P4</td>
<td>0.041807</td>
<td>0.068794</td>
<td>60.77129</td>
</tr>
<tr>
<td>P2</td>
<td>0.01859</td>
<td>0.061836</td>
<td>30.06339</td>
</tr>
</tbody>
</table>

Table 1. Execution vs Prediction for six non-twiddle codelets

Table 2 holds the predicted times for the modeled twiddle and non-twiddle codelets.

<table>
<thead>
<tr>
<th>Twiddle Codelets</th>
<th>Time (µs)</th>
<th>Non-twiddle Codelets</th>
<th>Time (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.0337442</td>
<td>1</td>
<td>0.0111672</td>
</tr>
<tr>
<td>3</td>
<td>0.0621405</td>
<td>2</td>
<td>0.01859</td>
</tr>
<tr>
<td>4</td>
<td>0.0748073</td>
<td>3</td>
<td>0.0380635</td>
</tr>
<tr>
<td>5</td>
<td>0.123439</td>
<td>4</td>
<td>0.0418074</td>
</tr>
<tr>
<td>6</td>
<td>0.136066</td>
<td>5</td>
<td>0.0815163</td>
</tr>
<tr>
<td>7</td>
<td>0.194844</td>
<td>6</td>
<td>0.0852202</td>
</tr>
<tr>
<td>8</td>
<td>0.175963</td>
<td>7</td>
<td>0.134986</td>
</tr>
<tr>
<td>9</td>
<td>0.271823</td>
<td>8</td>
<td>0.107272</td>
</tr>
<tr>
<td>10</td>
<td>0.267594</td>
<td>9</td>
<td>0.194209</td>
</tr>
<tr>
<td>16</td>
<td>0.417579</td>
<td>10</td>
<td>0.181058</td>
</tr>
<tr>
<td>32</td>
<td>0.978175</td>
<td>11</td>
<td>0.279649</td>
</tr>
<tr>
<td>64</td>
<td>2.25334</td>
<td>12</td>
<td>0.192931</td>
</tr>
</tbody>
</table>

Once the models have been created and depending on the required FFT size, a performance prediction can be extracted.

The next phase in the analysis requires the creation of the communication and synchronization models which predict the overhead cost of transferring and receiving the FFT data among processors. The MPI-FFTW uses a number of known MPI calls (such as MPI-Send, MPI-Receive, MPI-Gather, etc) to communicate and synchronize the parallelization of the transform process. For example, it uses MPI_Bcast to broadcast a message (such as transform size) to all used processors, and uses MPI_Send to send data. These MPI calls form the main communication/synchronization overhead of the MPI-FFTW.

The calls are evaluated using PACE’s parallel template where the number of processors to be used is also declared. The application object (Fig.1) is configured, compiled and linked with the subtask object (Fig.2) along with the parallel template (Fig.3). The parallel template passes the number of processors through variable Nproc. The resulting binary would now produce the communication cost estimates depending on the required number of processors.

```
application foo_appl
{
  include foo_subt;
  include hardware;
  var numeric:    N = Nproc = 1;
  link {
    hardware: Nproc = Nproc; foo_subt: N = N/Nproc;
  }
  option {
    hrduse="IntelP4_2400";
  }
  proc exec init { call foo_subt;}
}
```

Fig.1. Application object (foo_appl) used in MPI-PACE

Table 3 shows the predicted communication / synchronization costs for the MPI-FFTW on the given number of machines. The ability to evaluate and predict the cost of communication when using parallel processing, gives researchers a considerable advantage since there need not be any supporting application execution. This is especially true if the application at hand is big and takes a long time to execute.
subtask foo_subt
{
    include async;
    include hardware;
    var numeric: N;
    link {
        async: Tx = main(), N = N;
    }
}
(* * CHIP3S * Application Characterization Tool
* Source : foo_subt * RU Type: clc *)
proc cflow main {
    compute <is clc, FCAL>;
}
Fig.2. Application subtask (foo_subt) used in MPI-PACE

#include <mpidefs.h>
(* * async.la - Parallel template*)
partmp async {
    include hardware;
    var numeric: N;
    var compute: Tx; (*execution time *)
    option {
        nstage = 1, seval = 1;
    }
    proc exec init {
        var numeric : i;
        step mpcicom {
            for (i = 2; i <= hardware.Nproc; i = i + 1) {
                call MPI_Barrier (MPI_COMM_WORLD);
                call MPI_Bcast (4,1,MPI_COMM_WORLD);
                call MPI_Gather (4,4,1,MPI_COMM_WORLD);
                confdev 1, i, 4, MPI_COMM_WORLD;
            }
        }
        (* print "async: Tx=", Tx; *)
        step cpu {
            confdev Tx;
        }
    }
}
Fig.3. Parallel template object used in MPI-PACE

One additional advantage is that a user can also evaluate the parallelization of more than the number of machines currently available. For example, in this study, up to 256 processors (which is more than the available resources) were defined in the evaluation models. This helps in trying to foresee the outcome of the parallelization of computing certain FFT and parallelization sizes where it was impossible to do because of either huge communication overhead that incur unacceptable bottleneck, or the fact that there weren’t enough machines to do the experiments on.

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Communication Overhead (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>637.776</td>
</tr>
<tr>
<td>4</td>
<td>2305.48</td>
</tr>
<tr>
<td>8</td>
<td>7074.2</td>
</tr>
<tr>
<td>16</td>
<td>19549.7</td>
</tr>
<tr>
<td>32</td>
<td>49949.4</td>
</tr>
<tr>
<td>64</td>
<td>121680</td>
</tr>
<tr>
<td>128</td>
<td>356368</td>
</tr>
<tr>
<td>256</td>
<td>1.30159e+06</td>
</tr>
</tbody>
</table>

Table 3. Predicted communication/synchronization costs for MPI-FFTW

After creating the models for the communication / synchronization overheads and the codelets, an analysis and performance evaluation may now be conducted for any FFT size and for any number of processors in a parallelized environment.

Given the fact that all machines used in this study are of the same performance characterization, then it is safe to assume that if the predicted execution time for an n point FFT equals Tseq (predicted sequential time), then the predicted execution time for an n/p point FFT is (Tseq / p). If this is then applied to a distributed memory environment where p represents the number of processors, and add the predicted overhead cost, then the predicted computation is as follows:

\[
\text{Predicted Tpar (p)} = \frac{\text{Tseq}}{p} + T_{(\text{commu.+sync})}
\]  

where Tpar is the predicted time for computing a transform by the FFTW in a parallel environment, Tseq is the predicted sequential time for the same transform, p is the number of processors used, and T (commu.+ sync) is the overhead for communication / synchronization.

MPI-FFTW uses a speedup measurement to evaluate its performance enhancement through parallelism. Therefore, the experiments conducted here are also measured accordingly so that comparisons could be made in terms of speedups as well as execution times. This measurement is given by the factor of sequential execution time over the time obtained when run in a distributed environment:

\[
\text{Speedup} = \frac{T_{\text{seq}}}{T_{\text{par}}}
\]

or:

\[
\text{Speedup} = \frac{T_{\text{seq}}}{[T_{\text{seq}} / p] + T_{(\text{commu.+sync})}}
\]
5 A Parallelized Performance Prediction and Evaluation of the MPI-FFTW

Up to 16 concurrent processors were used to predict and measure in real-time the enhancement gained by using the MPI-FFTW to compute small transforms. Results show that for small FFTs, both the predicted performance as well as the FFTW real-time execution degrades instantly when distributed memory is used. This is because the initial cost of communication overhead is greater than the total computation cost of the FFT in a single machine. Fig.4 shows the predicted speedup for computing a 64 point FFT on 2, 4, 8, and 16 processors in comparison to the real-time speedup gained by the MPI-FFTW. The speedups for one processor (which is 1 for all FFT sizes) are not plotted so that the rest of the speedups can be clearly viewed.

Other FFT sizes (256, 512, K, 2K, and 4K) were used in order to compare the MPI-FFTW with the MPI-PACE. The transforms were computed using 16 distributed machines. Fig.5 shows the predicted speedups, while Fig.6 shows the real-time measurements. The behavior of the MPI-FFTW in the beginning is similar to the predicted one. The initial increase in the number of processors results in the degradation of the performance. The behavior changes a little as the real-time speedups indicate a better performance than predicted when 8 processors are used for sizes 2K and 4K, but the overall pattern remains the same.

While the real-time speedups depend on the availability of the machines (which is not always possible), and the possibility of affecting other users due to communication overhead, it is much easier to conduct performance evaluations and produce reliable predictions. To illustrate this point further, a number of evaluation experiments were conducted using 1 to 256 processors and for FFT sizes ranging from 32 to 128K. As Fig.7 shows, it was possible to conduct a prediction analysis for up to 256 machines. The figure shows that for the user to benefit from the available distributed resources in the machines, the transform must be at least of the size 16K. It also indicates that for that size, only using 2 processors would be deemed successful in terms of speedup. Transforms of sizes 32K, 64K, and 256K may be computed faster when 4 machines are used in parallel.

Otherwise the communication/synchronization overhead is proving much costlier and the speedups start to degrade again as the number of processors increase. However, although the performance prediction of the MPI-FFTW is close to that of the real-time execution when the size of the FFT is small, a different scenario occurs when a comparison is made for larger transforms. A number of factors contribute to this discrepancy. The FFTW is an adaptive package and changes its behavior according to the circumstances of the computation, the final outcome of the performance of the FFTW depends also on the FFT size, speed in which individual/combined transforms (plans) are computed, and the load on the system. As these factors increase in size and number, so would the difference between the predicted and the real-time performance. An increase in the number of plans means an increase in the variations of best plans chosen for each run and hence the possibility of sending a different plan to compute the transform then the one being analyzed and predicted.

Furthermore, the load affecting the machines in the real-time environment also contributes to the overall performance of the FFTW, and to have multiple machines running at the same time while being affected by external variables means that the chance of predicting precisely what the performance would be is more difficult when the size of the problem is larger. Another factor is the probability of having the communication/synchronization cost overestimated, which may indicate a need for further research towards improving the MPI analysis especially when...
dynamically optimized applications are studied. However, for most practical sizes of FFTs, the user can greatly benefit from the modeling techniques as well as the dynamic performance evaluation presented here.

6 Modeling and Predicting the Efficiency

The efficiency of parallel computation is calculated as the ratio of speedup resulting from distributing a given problem over a number of processors. Efficiency information in this case study is helpful in determining if the predicted speedups are efficient enough for the size of the given problem. The efficiency of using \( p \) number of processors equals the speedup gained when using \( p \) processors in a parallel environment divided by \( p \), and it is written as follows:

\[
\text{Efficiency}(p) = \frac{\text{Speedup}(p)}{p}
\] (11)

For example, if the speedup of using 4 machines in parallel to compute a given FFT is 2, then the efficiency of this computation process is \( 2/4 = 1/2 \). That is, the utilisation power of such environment is only 50\% of the total computational resources available. Therefore, while speedups in Fig.7 are up to 2.5 faster on certain sizes, this is still not the best performance the machines can offer. Unless the machines are being used in solving other problems, 1/2 of the resources remain idle while the computation is going on. Fig.8 illustrates that the efficiency predicted is similar to that gained by the MPI-FFTW. The figure represents prediction and execution comparison for the efficiency of computing a 64 point transform in a distributed environment consisting of up to 16 processors.

Fig.8. Efficiency comparisons for computing a 64 point FFT

Fig.9 predicts that using one machine to solve up to 128K FFT is more efficient than parallelising the computation even if there are as many as 256 powerful processors. One reason for this is the large communication overhead cost that is involved when computing this size of FFT. The fact that the data must be divided and then distributed among processors for the computation then they must be reallocated back contributes to this large overhead. For comparison purposes, the efficiency of the parallelisation process of the real-time MPI-FFTW is also measured (although less machines used). As predicted and as illustrated by Fig.10, there isn’t much efficiency in trying to parallelise the computation of the FFTs using a fast machine like the Pentium. The decrease in efficiency is as sharp as the one predicted by the previous figure.

Fig.9. Predicted efficiency for computing various FFT sizes using up to 256 machines
Experiments carried out here also show that parallelization schemes can be analyzed and their performances can be predicted where either the cost of doing real-time studies is expensive and/or when there aren’t enough resources available. The results also highlight MPI-PACE overestimation of the communication / synchronization in distributed environments and the need to further re-examine it so that a more precise evaluation is conducted.

References:


7 Summary

This paper presented a modeling technique for evaluating the performance and efficiency of an application execution in distributed environment. Besides predicting the behaviour pattern of the parallelised computations, the paper also analysed and predicted speedups as well as efficiency gained through multiprocessing. Results showed a reliable prediction process with a huge reduction in the overhead cost of the learning process. Predicted speedups suggest a performance enhancement that is not efficient because the communication and synchronisation costs are too big for the computations at hand. This is especially true when large number of processors is used. These predictions and overall behaviour patterns are matched by the real-time implementation.