Measuring Naira: a Symbolic Program with Irregular Parallelism

Sahalu B. Junaidu and Phil W. Trinder

Information and Computer Science Department, King Fahd University of Petroleum and Minerals, Dhahran 31261, Saudi Arabia.
Department of Computer Science and Electrical Engineering, Herriot-Watt Unviversity, Edingburgh, Scotland.

Abstract

Naira is a compiler for Haskell, written in Glasgow parallel Haskell. It exhibits modest, but irregular parallelism that is determined by properties of the program being compiled, e.g. the complexity of the types and of the pattern-matching. We report four experiments into Naira's parallel behaviour using a set of realistic inputs: namely the 18 Haskell modules of Naira itself. The issues investigated are:

- Does increasing input size improve sequential efficiency and speedup?
- To what extent do high communications latencies reduce average parallelism and speedup?
- Does migrating running threads between processors improve average parallelism and speedup at all latencies?

1 INTRODUCTION

The parallel behaviour of programs with regular parallelism is well understood: there are good simulation tools [?], a variety of analytical cost models [?, ?] and hybrid methods [?]. Unfortunately many useful programs lack such a regular structure, for example the number and size of tasks may be determined by the input. It is much harder to construct analytical models of the behaviour of programs with such irregular parallelism, and cost models are far less well developed [?, ?, ?]. Instead we must rely on simulation and measurement, and for these to be meaningful the program simulated must be realistic, and applied to real input data.

Parallel functional languages with dynamic models of parallelism are well suited to constructing programs with irregular parallelism. In the implementation of these languages many aspects of parallel behaviour, e.g. how many threads to create, and where to execute a thread, are automatically managed by a sophisticated runtime system. In consequence the runtime system can adapt at runtime to irregular parallel behaviour. Glasgow parallel Haskell (GpH) is one such language [?], and several irregularly-parallel programs have been constructed [?].

This paper reports a series of experiments into the irregular parallel behaviour of the Naira compiler. Naira comprises approximately five thousand lines of GpH, and a thousand lines of C. As a compiler Naira performs symbolic manipulation of program text. Although the results of our experiments only apply directly to Naira, we hope that the principles established will hold for other symbolic programs with irregular parallelism. Naira is a suitable basis for these experiments because of the availability of a suite of real input data, and more importantly it is a large, real program that achieves wall-clock speedups, as reported in section **??**.

2 NAIRA

Naira compiles from a substantial subset of Haskell to a RISC-like target language that has been extended with special parallel constructs [?]. The front end of the compiler comprises about 5K lines of GpH code organised in 18 modules. Input is in the form of a Haskell module, i.e. a file containing function definitions. The phases of the compiler, and the data dependencies between them, are depicted in Figure **??**. The first phase, analysis, consists of the lexical analyser and the parser. The next four phases implement the pattern matching compiler, the lambda lifter, the type checker and the intermediate language optimiser respectively. The detailed organisation and implementation of these phases is described elsewhere [?].

The main sources of parallelism in Naira are as follows. Data-parallel compilation of each function definition in the input. Pipelining of data between the compilation phases identified in Figure **??**. Control parallelism in the pattern-matcher and lambda-lifter.

Naira's modest parallelism has been measured on the GranSim simulator, and

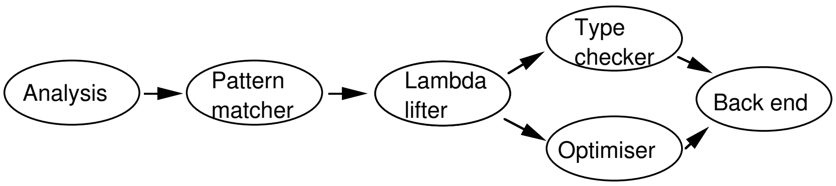

FIGURE 1: Naira Architecture

subsequently on a network of workstations. GranSim is a simulator that can be parameterised to emulate different parallel architectures [**?**]. The best simulated speedups achieved are 10.9 on an idealised parallel machine with an unbounded number of processors; 6.3 on an 8-processor shared-memory architecture; 5.8 on an 8-processor distributed-memory architecture. On a network of 5 Sun workstations Naira achieves a maximum speedup of 2.73 relative to its execution on a single workstation. However parallel GpH code carries an execution overhead compared to sequential code, and Naira's sequential efficiency is 89%. As a result the wallclock speedup achieved 2.46 on 5 processors. These speedups agree with GranSim's predicted speedup of 3.01 for the workstation-network. Detailed descriptions of the parallelisation, and measurement of Naira can be found in [**?**, **?**].

3 EXPERIMENTAL APPARATUS

The first experiment below measures Naira's performance using GpH's GUM runtime system. The other experiments use the GranSim parameterisable simulator [**?**] because it provides a cheap and convenient platform. GranSim is closely based on the GUM runtime system that manages execution of GpH programs on real parallel architectures. GranSim takes parameters describing many aspects of a parallel architecture, e.g. thread creation costs, communication latency etc. As outlined above, the accuracy of GranSim predictions for Naira has been corroborated, and it's accuracy for several other programs has been validated on several architectures [**?**].

The parallel program metrics used are standard [**?**], as follows. **Runtime:** the time to execute a program. **Sequential efficiency:** the ratio between runtime under the sequential runtime system, and runtime under the parallel runtime system on a single processor. Sequential efficiency measures the additional work that the parallel language must perform to synchronise with concurrent threads. **Average Parallelism:** the average number of active threads during a program's execution. **Speedup** (at N processors): the ratio between sequential runtime and parallel runtime (on N processors).

Input Module	**Sequential**	**Seq. for par. exec.**	**Efficiency**
MyPrelude	2.5	3.0	83%
DataTypes	2.8	3.2	89%
Tables	3.4	4.1	83%
PrintUtils	1.7	2.5	70%
Printer	3.3	4.0	82%
LexUtils	3.4	3.8	89%
Lexer	3.0	3.6	83%
SyntaxUtils	22.2	23.3	95%
Syntax	4.9	5.4	90%
MatchUtils	4.5	5.2	86%
Matcher	5.0	5.5	91%
LambdaUtils	2.1	2.6	79%
LambdaLift	7.2	8.2	88%
TCheckUtils	10.0	11.8	84%
TChecker	2.8	3.7	75%
OptmiseUtils	7.7	9.4	82%
Optimiser	10.6	12.6	84%
Main	6.9	8.4	82%
Minimum efficiency			70%
Maximum efficiency			90%
Mean efficiency			**86%**

TABLE 1: Naira's Runtime and Sequential Efficiency for each module

4 EXPERIMENT 1: IMPACT OF INPUT SIZE

The first experiment investigates the impact of input size on both the sequential efficiency, and speedups achieved by Naira.

4.1 Input Size vs Sequential Efficiency

What is the impact of input size on the sequential efficiency of symbolic irregularly-parallel GpH programs? Table 1 shows the runtimes for Naira compiling each of it's 18 GpH modules. The runtime figures in this table are the wall-clock (real) timings taken to compile each input module. These times (measured in seconds) are averaged over several program executions. The second column of Table 1 records the Naira runtime to compile of each module, when Naira itself is running, fully-optimised, under the standard sequential runtime system. The third column gives the Naira runtime to compile each module, under the parallel runtime system on a single processor. The fourth column contains the efficiency calculated by dividing the second column by the third, and shows that the overhead imposed by parallel execution some can be high for some inputs, e.g. PrintUtils, or TChecker.. The last three rows contain a summary of information in the table.

Naira's input size can be increased by concatenating modules, for example the module `Preface` in Table 2 is a result of concatenating the contents of the `MyPrelude` and `DataTypes` modules from Table 1. Table 2 reports the sequential efficiency of Naira compiling these larger inputs. Similarly Table 3 reports the sequential efficiency of Naira compiling files constructed by concatenating 4 of the original modules.

The effect of increasing input size can be observed by comparing Tables 1, 2 and 3, and we note that the mean sequential efficiency of Naira modules increases

Input Module	**Sequential**	**Seq. for par. exec.**	**Efficiency**
Preface	14.4	21.8	66%
Printing	7.7	8.7	88%
Lexing	8.4	9.1	92%
Parsing	33.3	35.8	93%
PMatching	13.2	14.6	90%
LLifting	6.5	7.2	90%
TChecking	15.5	17.9	86%
Optimising	41.1	45.1	91%
Minimum efficiency			66%
Maximum efficiency			93%
mean efficiency			**87%**

TABLE 2: Naira's Runtime and Sequential Efficiency for 2-module inputs

Input Module	**Sequential**	**Seq. for par. exec.**	**Efficiency**
MyPrelToPrinter	63.9	76.2	84%
LexUToSyntax	67.1	74.8	90%
MatchUToLLift	33.4	37.2	90%
MAIN	115.3	120.3	96%
Minimum efficiency			84%
Maximum efficiency			96%
mean efficiency			**91%**

TABLE 3: Naira's Runtime and Sequential Efficiency for 4-module inputs

from 86% to 91%. Maximum efficiency likewise increases uniformly, from 90% to 93%, however, it is not clear why minimal efficiency drops to 66% in Table 2. Sequential efficiency probably improves because fixed parallel execution overheads, like program start-up costs, are amortised over a longer runtime.
Conclusion 1.1: *Sequential efficiency improves with input size.*

4.2 Input Size vs Speedup

What is the impact of input size on the average parallelism and speedups achieved by symbolic irregularly-parallel GpH programs? Naira's 18 GpH modules are a varied collection of real compiler inputs. Several size metrics are possible for compiler input, each with a different variability in the 18 modules. For example Lines of source code, varying from 100 to 500 lines; Number of function definitions, varying from 10 to 90 functions; Number of output lines produced, varying from 100 to 1200 lines (measures input size in the sense that the number of output lines reflects the complexity of the functions compiled). The measure selected is the number of function definitions as it provides a suitably abstract view of compiler input.

Figure **??** plots the size of 18 modules against speedup, together with line of best fit. We see a weak correlation between size and speedup, as reflected by the regression coefficient of X. There are several reasons why speedup may be improved with larger input, most probably the overheads of parallel structures, like pipeline startup and shutdown, are amortised over a longer runtime. For irregularly-parallel programs, however, a special factor may be at work: a longer runtime allows the runtime system more time to adapt, and take advantage of the adaption.

Tentative Conclusion 1.2: *There is a weak correlation between input size and speedup.*

Note to reviewers: *Regression coefficient will be supplied, and line of best fit plotted*

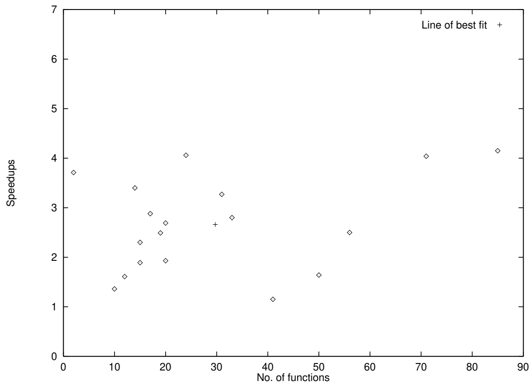

FIGURE 2: Number of functions in Input vs Speedups

5 EXPERIMENT 2: IMPACT OF LATENCY

To what extent do high communication latencies reduce average parallelism and speedup of symbolic irregularly-parallel GpH programs? A key issue in parallel computing is achieving a high ratio of computation time to communication time. To discuss this issue in an architecture independent way the time to send a message, or communications *latency* is expressed in terms of the number of (computational) machine cycles. There are a whole range of parallel architectures, ranginf from: shared-memory machines with a latency of around 5 or 10 cycles; fast distributed-memory machines (e.g. Meiko CS2) with a latency of 100 to 500 cycles; conventional distributed memory machines (e.g. IBM SP2) with a latency of 1000-5000 cycles; to networked machines (e.g. Suns on an ethernet) with a latency of 25000-100000 cycles.

Parallel language implementations in general, and GpH in particular, use a variety of mechanisms to hide latency costs, and high latencies expose the limitations of these mechanisms. GranSim allows us to measure a program on simulated architectures with varying latencies. Similar experiments have been reported elsewhere [**?**, **?**], but Naira is a larger program, and is measured on a set of real input data.

Table **??**, and Figures 3a and 3b summarise the average parallelism and speedups of Naira on a range of loosely-coupled architectures with varying communication latencies. In the table the maximum and minimum average parallelism and speedup figures at each latency are boxed. We note that compiling some modules typically produces the greatest speedup, or the greatest average parallelism. Likewise some modules typically produce the least speedup, or average parallelism.

The table shows that the average parallelism and speedup for each program falls linearly as latency increases. Figures 3a and 3b give a depiction of the trend for all programs, plotting mean, minimum and maximal speedup and average parallelism figures. This result is in contrast to earlier work on smaller irregular symbolic programs (approximately 500 lines) with greater speedups (approximately 28) report an exponential reduction in speedup [**?**, **?**]. The difference may be because speedup and average parallelism are already low in Naira.

The figures also show that the difference between average parallelism and speedup is dramatic at a latency of 400 cycles, however as latency increases the difference between average parallelism and speedup diminishes. This indicates that at high latencies less unnecessary work is performed, but why this is so is currently unclear.

Conclusion 2.1: *Increased latency linearly reduces both average parallelism and speedup.*

Conclusion 2.2: *Increased latency reduces the difference between average parallelism and speedup.*

Note to reviewers: *Figures 3 & 4 will use a log. scale in the final version*

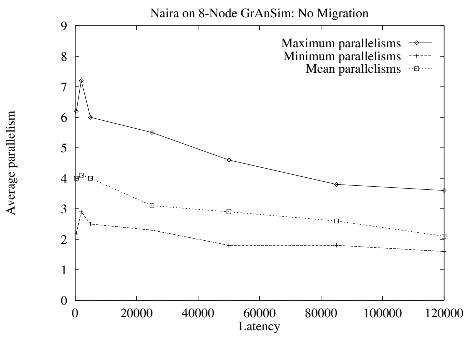

Figure 3a: parallelism.

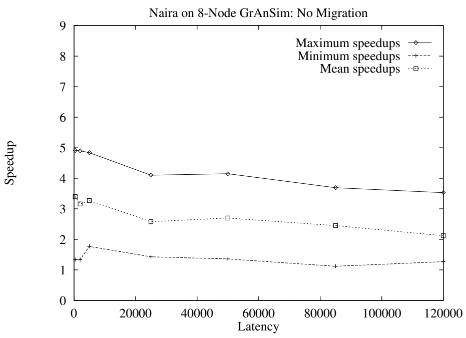

Figure 3b: speedups.

6 EXPERIMENT 3: IMPACT OF THREAD MIGRATION

What is the impact of thread migration on the average parallelism and speedups achieved by symbolic irregularly-parallel GpH programs at a range of latencies? Thread migration is an adaptive load-balancing technique for parallel languages where a thread of computation started on a heavily-loaded processor may be moved to an idle processor. It is especially useful for correcting poor load distributions where one processor is overburdened with work. Implementing thread migration in a runtime is complex undertaking, and it is not currently available in GpH. Part of the motivation for this experiment is to determine whether the benefits of thread migration for real programs are worth the implementation effort.

The program measurements reported in Table **??** have been repeated with GranSim parameterised to permit thread migration. The results are summarised in Figures 4a and 4b. At high latencies, i.e. above 5K cycles, both the speedups and average parallelism achieved with migration, and without it, are very similar. Differences are in the order of a few percent, typically in favour of with-migration, with occasional exceptions. This contradicts a body of earlier work on smaller irregular programs that indicates that thread migration gives significant improvements [**?**]. There may be several reasons why migration does not improve Naira speedups at high latencies. Naira may have naturally generate a good load distribution, never requiring migration. Alternately the loss of data locality caused by migration may offset the benefits of migration, e.g. the cost of communicating large data structures like the symbol table outweigh the improved load distribution. Further investigation might establish whether Naira is typical of large irregular programs, or whether there are special considerations.

For most inputs at lower latencies speedup and average parallelism are similar with migration and without it. However, for a few inputs at lower latencies, speedup and average parallelism are dramatically improved by migration. This is reflected in the variability of Figures 3a and 3b below 5K cycles, as compared with Figures 4a and 4b. Example modules with dramatic improvements are Syntax, and LexUtils, and we expect that poor load distributions are generated for these inputs. This result is consistent with the earlier work thread migration [**?**], and extends it to medium-scale programs at low latencies . It also confirms earlier observations suggesting that for low-latency architectures a good load distribution is more important than good data locality [**?**].

Conclusion 3.1: *At high latencies thread migration makes little difference to speedup or average parallelism.*

Conclusion 3.2: *At low latencies thread migration can make performance more predictable: giving dramatic improvements for some inputs.*

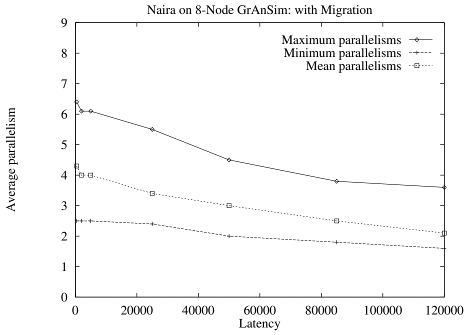

Figure 4a: parallelism.

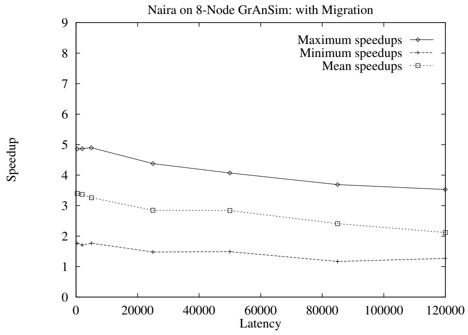

Figure 4b: speedups.

7 CONCLUSIONS

We have measured several aspects of the irregularly-parallel behaviour of the Naira compiler using the GranSim simulator. GranSim's prediction of Naira's behaviour has previously been validated on a network of workstations. We make the following conclusions: increasing input size improves both sequential efficiency and speedups. Raising communication latency reduces average parallelism and speedup linearly, and also reduces the difference between average parallelism and speedup. At low latencies we confirm earlier work on smaller programs showing that thread migration gives more predictable speedups and average parallelism, with dramatic improvements for some inputs. A new and more controversial result is that, for Naira at least, thread migration makes little difference at high latencies.

The experiments reported here are the first measurements of a medium-scale GpH symbolic and irregularly-parallel program using a substantial set of real input data. We hope that future work will show that the principles established above hold for a class of irregularly-parallel programs. Further investigation may resolve some of the questions raised by the experiments, for example why does increased communication latency reduce the difference between average parallelism and speedup?

Acknowledgements We would like to thank Tony Davie, Kevin Hammond, and Hans Wolfgang Loidl for stimulating discussions and other contributions to this work.

REFERENCES

[1] DL Eager, J Zahorhan, and ED Lazowska, "Speedup Versus Efficiency in Parallel Systems", *IEEE Transactions on Computers* 38(3), pp. 408-423, March 1989.

[2] T Fahringer *Automatic Performance Prediction of Parallel Programs*, Kluwer Academic Press, 1996.

[3] K Hammond, H-W Loidl and AS Partridge, "Visualising Granularity in Parallel Programs: A Graphical Winnowing System for Haskell". In *HPFC'95—Conf. on High Performance Functional Computing*, pp. 208-221, Denver, CO, April 10-12, 1995.

[4] S Junaidu, A Davie and K Hammond, "Naira: A Parallel^2 Haskell Compiler" In *9th Intl Workshop on Implementing Functional Languages*, pp. 214-230, London, UK, September 1997.

[5] S Junaidu, *A Parallel Functional Language Compiler for Message Passing Multicomputers*. PhD thesis, School of Mathematical and Computational Sciences, St Andrews University, Scotland, 1998.

[6] H-W Loidl and K Hammond, "Making a Packet: Cost-Effective Communication for a Parallel Graph Reducer", In *IFL'96 — International Workshop on the Implementation of Functional Languages*, Springer-Verlag LNCS 1268, pp. 184-199, Bad Godesberg, Germany, September 1996.

[7] H-W. Loidl, *Granularity in Large-Scale Parallel Functional Programming*, PhD Thesis, Department of Computer Science, University of Glasgow, 1998.

[8] H.-W. Loidl, and P.W. Trinder, K. Hammond, S.B. Junaidu R.G. Morgan and S.L. Peyton Jones, "Engineering Parallel Symbolic Programs in GPH", *to appear in Concurrency Practice and Experience*, 2000.

[9] G Ostheimer, *Parallel Functional Programming for Message Passing Multiprocessors*. PhD Thesis, Department of Mathematical and Computational Sciences, St. Andrews University, Scotland, 1993.

[10] V Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*, MIT Press, Cambridge, Mass., USA, 1989

[11] PW Trinder, K Hammond, JS Mattson Jr., AS Partridge and SL Peyton Jones, "GUM: A Portable Parallel Implementation of Haskell". In *PLDI '96 — Programming Languages Design and Implementation*, pp. 78-88, Philadelphia, PA, May 1996.

[12] KY Wang "A Framework for Static, Precise Performance Prediction for Superscalar-Based Parallel Computers". In *4th Intl Workshop on Compilers for Parallel Computers*, pp. 413-427, Delft, The Netherlands, December 1993.

[13] N Yazici-Pekergin and JM Vincent, "Stochastic Bounds on Execution Times of Parallel Programs". *IEEE Trans. on Software Engineering* 17(10), pp. 1059-1068, October 1991.

Input Module	*Communication Latencies (Distributed Memory Architectures)*													
	400 cycles		*2K cycles*		*5K cycles*		*25K cycles*		*50K cycles*		*85K cycles*		*120K cycles*	
	Paral.	**Spdup**	**Paral.**	**Spdup**	**Paral.**	**Spdup**	**Paral.**	**Spdup**	**Paral.**	**Spdup**	**Paral.**	**Spdup**	**Paral.**	**Spdup**
MyPrelude	4.6	3.41	4.2	3.00	4.3	3.17	3.0	2.18	2.5	1.80	$\boxed{1.8}$	1.50	$\boxed{1.6}$	1.39
DataTypes	4.7	3.86	3.8	3.12	4.3	3.53	3.9	3.25	3.6	3.01	2.0	1.66	1.7	1.51
Tables	5.0	3.21	4.6	2.93	4.9	3.13	4.1	2.64	4.0	2.77	2.7	1.79	2.6	1.86
PrintUtils	$\boxed{2.5}$	2.66	$\boxed{2.5}$	2.62	$\boxed{2.5}$	2.57	$\boxed{2.4}$	2.52	$\boxed{2.0}$	2.09	1.9	2.06	1.8	2.04
Printer	3.8	3.98	3.8	3.90	3.7	3.87	3.3	3.43	2.4	2.52	2.6	2.71	1.8	1.88
LexUtils	$\boxed{6.4}$	3.18	5.6	4.32	5.4	3.32	5.1	$\boxed{4.38}$	$\boxed{4.5}$	3.89	3.7	2.99	2.7	2.50
Lexer	4.9	3.92	4.7	3.94	4.5	3.68	3.2	3.25	2.6	2.87	2.2	2.84	2.2	2.78
SyntaxUtils	6.2	2.46	$\boxed{6.1}$	2.54	$\boxed{6.1}$	2.31	$\boxed{5.5}$	2.10	3.7	4.33	3.6	4.55	2.7	3.19
Syntax	4.3	2.62	3.9	2.41	4.1	2.54	3.5	2.12	2.9	1.81	2.3	1.43	2.2	1.34
MatchUtils	3.9	4.05	3.6	3.75	3.7	3.86	2.9	3.04	3.1	4.06	1.8	1.87	1.7	2.04
Matcher	4.2	3.66	4.1	3.68	4.2	2.98	3.4	2.41	3.1	3.27	2.8	2.77	2.0	1.50
LambdaUtils	3.7	2.32	3.5	2.21	3.4	2.14	2.9	1.82	2.6	1.60	2.4	1.57	2.1	1.34
LambdaLift	4.6	4.26	4.3	4.67	4.0	4.84	3.4	3.06	2.8	2.72	2.6	2.59	1.9	2.00
TCheckUtils	5.1	$\boxed{4.86}$	5.2	$\boxed{4.87}$	5.2	$\boxed{4.90}$	4.3	4.10	4.3	$\boxed{4.07}$	$\boxed{3.8}$	$\boxed{3.69}$	$\boxed{3.6}$	$\boxed{3.53}$
TChecker	2.7	$\boxed{1.77}$	2.8	$\boxed{1.71}$	2.6	$\boxed{1.77}$	2.4	$\boxed{1.48}$	2.3	$\boxed{1.49}$	1.9	$\boxed{1.17}$	2.0	$\boxed{1.27}$
OptmiseUtils	3.0	3.59	3.0	3.55	2.6	3.11	2.4	2.92	2.3	2.72	2.0	2.33	1.8	2.23
Optimiser	3.4	4.76	3.3	4.67	3.3	4.59	3.0	4.23	2.9	4.05	2.3	3.88	2.0	3.66
Main	3.6	2.69	3.5	2.70	3.1	2.31	3.1	2.33	2.5	1.98	2.6	1.97	2.1	2.03
MEAN	4.3	3.40	4.0	3.37	4.0	3.26	3.4	2.85	3.0	2.84	2.5	2.41	2.1	2.12

TABLE 4: Naira on 8-node GranSim with varying latencies: **With Migration**.