Dror Feitelson
Larry Rudolph
Uwe Schwiegelshohn (Eds.)

LNCS 3277

Job Scheduling Strategies for Parallel Processing

10th International Workshop, JSSPP 2004
New York, NY, USA, June 2004
Revised Selected Papers

Springer

Dror Feitelson Larry Rudolph
Uwe Schwiegelshohn (Eds.)

Job Scheduling Strategies for Parallel Processing

10th International Workshop, JSSPP 2004
New York, NY, USA, June 13, 2004
Revised Selected Papers

Springer

Volume Editors

Dror Feitelson
The Hebrew University
School of Computer Science and Engineering
91904 Jerusalem, Israel
E-mail: feit@cs.huji.ac.il

Larry Rudolph
CSAIL – Massachusetts Institute of Technology
32 Vassar Street, Cambridge, MA 02139, USA
E-mail: rudolph@csail.mit.edu

Uwe Schwiegelshohn
University of Dortmund
Computer Engineering Institute
44221 Dortmund, Germany
E-mail: uwe.schwiegelshohn@udo.edu

Library of Congress Control Number: 2005925176

CR Subject Classification (1998): D.4, D.1.3, F.2.2, C.1.2, B.2.1, B.6, F.1.2

ISSN 0302-9743
ISBN-10 3-540-25330-0 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-25330-3 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Boller Mediendesign
Printed on acid-free paper SPIN: 11407522 06/3142 5 4 3 2 1 0

Lecture Notes in Computer Science 3277

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

Lecture Notes in Computer Science

For information about Vols. 1–3360

please contact your bookseller or Springer

Vol. 3492: P. Blache, E. Stabler (Eds.), Logical Aspects of Computational Linguistics. X, 363 pages. 2005. (Subseries LNAI).

Vol. 3467: J. Giesl (Ed.), Term Rewriting and Applications. XIII, 517 pages. 2005.

Vol. 3465: M. Bernardo, A. Bogliolo (Eds.), Formal Methods for Mobile Computing. VII, 271 pages. 2005.

Vol. 3463: M. Dal Cin, M. Kaâniche, A. Pataricza (Eds.), Dependable Computing - EDCC 2005. XVI, 472 pages. 2005.

Vol. 3461: P. Urzyczyn (Ed.), Typed Lambda Calculi and Applications. XI, 433 pages. 2005.

Vol. 3459: R. Kimmel, N. Sochen, J. Weickert (Eds.), Scale Space and PDE Methods in Computer Vision. XI, 634 pages. 2005.

Vol. 3456: H. Rust, Operational Semantics for Timed Systems. XII, 223 pages. 2005.

Vol. 3455: H. Treharne, S. King, M. Henson, S. Schneider (Eds.), ZB 2005: Formal Specification and Development in Z and B. XV, 493 pages. 2005.

Vol. 3454: J.-M. Jacquet, G.P. Picco (Eds.), Coordination Models and Languages. X, 299 pages. 2005.

Vol. 3453: L. Zhou, B.C. Ooi, X. Meng (Eds.), Database Systems for Advanced Applications. XXVII, 929 pages. 2005.

Vol. 3452: F. Baader, A. Voronkov (Eds.), Logic for Programming, Artificial Intelligence, and Reasoning. XI, 562 pages. 2005. (Subseries LNAI).

Vol. 3450: D. Hutter, M. Ullmann (Eds.), Security in Pervasive Computing. XI, 239 pages. 2005.

Vol. 3449: F. Rothlauf, J. Branke, S. Cagnoni, D.W. Corne, R. Drechsler, Y. Jin, P. Machado, E. Marchiori, J. Romero, G.D. Smith, G. Squillero (Eds.), Applications on Evolutionary Computing. XX, 631 pages. 2005.

Vol. 3448: G.R. Raidl, J. Gottlieb (Eds.), Evolutionary Computation in Combinatorial Optimization. XI, 271 pages. 2005.

Vol. 3447: M. Keijzer, A. Tettamanzi, P. Collet, J.v. Hemert, M. Tomassini (Eds.), Genetic Programming. XIII, 382 pages. 2005.

Vol. 3444: M. Sagiv (Ed.), Programming Languages and Systems. XIII, 439 pages. 2005.

Vol. 3443: R. Bodik (Ed.), Compiler Construction. XI, 305 pages. 2005.

Vol. 3442: M. Cerioli (Ed.), Fundamental Approaches to Software Engineering. XIII, 373 pages. 2005.

Vol. 3441: V. Sassone (Ed.), Foundations of Software Science and Computational Structures. XVIII, 521 pages. 2005.

Vol. 3440: N. Halbwachs, L.D. Zuck (Eds.), Tools and Algorithms for the Construction and Analysis of Systems. XVII, 588 pages. 2005.

Vol. 3439: R.H. Deng, F. Bao, H. Pang, J. Zhou (Eds.), Information Security Practice and Experience. XII, 424 pages. 2005.

Vol. 3437: T. Gschwind, C. Mascolo (Eds.), Software Engineering and Middleware. X, 245 pages. 2005.

Vol. 3436: B. Bouyssounouse, J. Sifakis (Eds.), Embedded Systems Design. XV, 492 pages. 2005.

Vol. 3434: L. Brun, M. Vento (Eds.), Graph-Based Representations in Pattern Recognition. XII, 384 pages. 2005.

Vol. 3433: S. Bhalla (Ed.), Databases in Networked Information Systems. VII, 319 pages. 2005.

Vol. 3432: M. Beigl, P. Lukowicz (Eds.), Systems Aspects in Organic and Pervasive Computing - ARCS 2005. X, 265 pages. 2005.

Vol. 3431: C. Dovrolis (Ed.), Passive and Active Network Measurement. XII, 374 pages. 2005.

Vol. 3429: E. Andres, G. Damiand, P. Lienhardt (Eds.), Discrete Geometry for Computer Imagery. X, 428 pages. 2005.

Vol. 3427: G. Kotsis, O. Spaniol (Eds.), Wireless Systems and Mobility in Next Generation Internet. VIII, 249 pages. 2005.

Vol. 3423: J.L. Fiadeiro, P.D. Mosses, F. Orejas (Eds.), Recent Trends in Algebraic Development Techniques. VIII, 271 pages. 2005.

Vol. 3422: R.T. Mittermeir (Ed.), From Computer Literacy to Informatics Fundamentals. X, 203 pages. 2005.

Vol. 3421: P. Lorenz, P. Dini (Eds.), Networking - ICN 2005, Part II. XXXV, 1153 pages. 2005.

Vol. 3420: P. Lorenz, P. Dini (Eds.), Networking - ICN 2005, Part I. XXXV, 933 pages. 2005.

Vol. 3419: B. Faltings, A. Petcu, F. Fages, F. Rossi (Eds.), Constraint Satisfaction and Constraint Logic Programming. X, 217 pages. 2005. (Subseries LNAI).

Vol. 3418: U. Brandes, T. Erlebach (Eds.), Network Analysis. XII, 471 pages. 2005.

Vol. 3416: M. Böhlen, J. Gamper, W. Polasek, M.A. Wimmer (Eds.), E-Government: Towards Electronic Democracy. XIII, 311 pages. 2005. (Subseries LNAI).

Vol. 3415: P. Davidsson, B. Logan, K. Takadama (Eds.), Multi-Agent and Multi-Agent-Based Simulation. X, 265 pages. 2005. (Subseries LNAI).

Vol. 3414: M. Morari, L. Thiele (Eds.), Hybrid Systems: Computation and Control. XII, 684 pages. 2005.

Vol. 3412: X. Franch, D. Port (Eds.), COTS-Based Software Systems. XVI, 312 pages. 2005.

Vol. 3411: S.H. Myaeng, M. Zhou, K.-F. Wong, H.-J. Zhang (Eds.), Information Retrieval Technology. XIII, 337 pages. 2005.

Vol. 3410: C.A. Coello Coello, A. Hernández Aguirre, E. Zitzler (Eds.), Evolutionary Multi-Criterion Optimization. XVI, 912 pages. 2005.

Vol. 3409: N. Guelfi, G. Reggio, A. Romanovsky (Eds.), Scientific Engineering of Distributed Java Applications. X, 127 pages. 2005.

Vol. 3408: D.E. Losada, J.M. Fernández-Luna (Eds.), Advances in Information Retrieval. XVII, 572 pages. 2005.

Vol. 3407: Z. Liu, K. Araki (Eds.), Theoretical Aspects of Computing - ICTAC 2004. XIV, 562 pages. 2005.

Vol. 3406: A. Gelbukh (Ed.), Computational Linguistics and Intelligent Text Processing. XVII, 829 pages. 2005.

Vol. 3404: V. Diekert, B. Durand (Eds.), STACS 2005. XVI, 706 pages. 2005.

Vol. 3403: B. Ganter, R. Godin (Eds.), Formal Concept Analysis. XI, 419 pages. 2005. (Subseries LNAI).

Vol. 3401: Z. Li, L.G. Vulkov, J. Waśniewski (Eds.), Numerical Analysis and Its Applications. XIII, 630 pages. 2005.

Vol. 3399: Y. Zhang, K. Tanaka, J.X. Yu, S. Wang, M. Li (Eds.), Web Technologies Research and Development - APWeb 2005. XXII, 1082 pages. 2005.

Vol. 3398: D.-K. Baik (Ed.), Systems Modeling and Simulation: Theory and Applications. XIV, 733 pages. 2005. (Subseries LNAI).

Vol. 3397: T.G. Kim (Ed.), Artificial Intelligence and Simulation. XV, 711 pages. 2005. (Subseries LNAI).

Vol. 3396: R.M. van Eijk, M.-P. Huget, F. Dignum (Eds.), Agent Communication. X, 261 pages. 2005. (Subseries LNAI).

Vol. 3395: J. Grabowski, B. Nielsen (Eds.), Formal Approaches to Software Testing. X, 225 pages. 2005.

Vol. 3394: D. Kudenko, D. Kazakov, E. Alonso (Eds.), Adaptive Agents and Multi-Agent Systems II. VIII, 313 pages. 2005. (Subseries LNAI).

Vol. 3393: H.-J. Kreowski, U. Montanari, F. Orejas, G. Rozenberg, G. Taentzer (Eds.), Formal Methods in Software and Systems Modeling. XXVII, 413 pages. 2005.

Vol. 3392: D. Seipel, M. Hanus, U. Geske, O. Bartenstein (Eds.), Applications of Declarative Programming and Knowledge Management. X, 309 pages. 2005. (Subseries LNAI).

Vol. 3391: C. Kim (Ed.), Information Networking. XVII, 936 pages. 2005.

Vol. 3390: R. Choren, A. Garcia, C. Lucena, A. Romanovsky (Eds.), Software Engineering for Multi-Agent Systems III. XII, 291 pages. 2005.

Vol. 3389: P. Van Roy (Ed.), Multiparadigm Programming in Mozart/Oz. XV, 329 pages. 2005.

Vol. 3388: J. Lagergren (Ed.), Comparative Genomics. VII, 133 pages. 2005. (Subseries LNBI).

Vol. 3387: J. Cardoso, A. Sheth (Eds.), Semantic Web Services and Web Process Composition. VIII, 147 pages. 2005.

Vol. 3386: S. Vaudenay (Ed.), Public Key Cryptography - PKC 2005. IX, 436 pages. 2005.

Vol. 3385: R. Cousot (Ed.), Verification, Model Checking, and Abstract Interpretation. XII, 483 pages. 2005.

Vol. 3383: J. Pach (Ed.), Graph Drawing. XII, 536 pages. 2005.

Vol. 3382: J. Odell, P. Giorgini, J.P. Müller (Eds.), Agent-Oriented Software Engineering V. X, 239 pages. 2005.

Vol. 3381: P. Vojtáš, M. Bieliková, B. Charron-Bost, O. Sýkora (Eds.), SOFSEM 2005: Theory and Practice of Computer Science. XV, 448 pages. 2005.

Vol. 3380: C. Priami (Ed.), Transactions on Computational Systems Biology I. IX, 111 pages. 2005. (Subseries LNBI).

Vol. 3379: M. Hemmje, C. Niederee, T. Risse (Eds.), From Integrated Publication and Information Systems to Information and Knowledge Environments. XXIV, 321 pages. 2005.

Vol. 3378: J. Kilian (Ed.), Theory of Cryptography. XII, 621 pages. 2005.

Vol. 3377: B. Goethals, A. Siebes (Eds.), Knowledge Discovery in Inductive Databases. VII, 190 pages. 2005.

Vol. 3376: A. Menezes (Ed.), Topics in Cryptology – CT-RSA 2005. X, 385 pages. 2005.

Vol. 3375: M.A. Marsan, G. Bianchi, M. Listanti, M. Meo (Eds.), Quality of Service in Multiservice IP Networks. XIII, 656 pages. 2005.

Vol. 3374: D. Weyns, H.V.D. Parunak, F. Michel (Eds.), Environments for Multi-Agent Systems. X, 279 pages. 2005. (Subseries LNAI).

Vol. 3372: C. Bussler, V. Tannen, I. Fundulaki (Eds.), Semantic Web and Databases. X, 227 pages. 2005.

Vol. 3371: M.W. Barley, N. Kasabov (Eds.), Intelligent Agents and Multi-Agent Systems. X, 329 pages. 2005. (Subseries LNAI).

Vol. 3370: A. Konagaya, K. Satou (Eds.), Grid Computing in Life Science. X, 188 pages. 2005. (Subseries LNBI).

Vol. 3369: V.R. Benjamins, P. Casanovas, J. Breuker, A. Gangemi (Eds.), Law and the Semantic Web. XII, 249 pages. 2005. (Subseries LNAI).

Vol. 3368: L. Paletta, J.K. Tsotsos, E. Rome, G.W. Humphreys (Eds.), Attention and Performance in Computational Vision. VIII, 231 pages. 2005.

Vol. 3367: W.S. Ng, B.C. Ooi, A. Ouksel, C. Sartori (Eds.), Databases, Information Systems, and Peer-to-Peer Computing. X, 231 pages. 2005.

Vol. 3366: I. Rahwan, P. Moraitis, C. Reed (Eds.), Argumentation in Multi-Agent Systems. XII, 263 pages. 2005. (Subseries LNAI).

Vol. 3365: G. Mauri, G. Păun, M.J. Pérez-Jiménez, G. Rozenberg, A. Salomaa (Eds.), Membrane Computing. IX, 415 pages. 2005.

Vol. 3363: T. Eiter, L. Libkin (Eds.), Database Theory - ICDT 2005. XI, 413 pages. 2004.

Vol. 3362: G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, T. Muntean (Eds.), Construction and Analysis of Safe, Secure, and Interoperable Smart Devices. IX, 257 pages. 2005.

Vol. 3361: S. Bengio, H. Bourlard (Eds.), Machine Learning for Multimodal Interaction. XII, 362 pages. 2005.

Preface

This volume contains the papers presented at the 10th Anniversary Workshop on Job Scheduling Strategies for Parallel Processing. The workshop was held in New York City, on June 13, 2004, at Columbia University, in conjunction with the SIGMETRICS 2004 conference.

Although it is a workshop, the papers were conference-reviewed, with the full versions being read and evaluated by at least five and usually seven members of the Program Committee. We refer to it as a workshop because of the very fast turnaround time, the intimate nature of the actual presentations, and the ability of the authors to revise their papers after getting feedback from workshop attendees. On the other hand, it was actually a conference in that the papers were accepted solely on their merits as decided upon by the Program Committee.

We would like to thank the Program Committee members, Su-Hui Chiang, Walfredo Cirne, Allen Downey, Eitan Frachtenberg, Wolfgang Gentzsch, Allan Gottlieb, Moe Jette, Richard Lagerstrom, Virginia Lo, Reagan Moore, Bill Nitzberg, Mark Squillante, and John Towns, for an excellent job.

Thanks are also due to the authors for their submissions, presentations, and final revisions for this volume. Finally, we would like to thank the MIT Computer Science and Artificial Intelligence Laboratory (CSAIL), The Hebrew University, and Columbia University for the use of their facilities in the preparation of the workshop and these proceedings.

This year saw a continued interest in scheduling in grid and cluster environments, with a growing representation of real-system issues such as workload studies, network topology issues, and the effect of failures. At the same time, there was also a strong representation of research relating to classical multiprocessor systems, and lively discussions contrasting the academic point of view with that of administrators of 'real' systems. We hope that the papers in this volume capture this range of interests and approaches, and that you, the reader, find them interesting and useful.

This was the tenth annual workshop in this series, which reflects a consistent and ongoing interest; the organizers believe that the workshop satisfies a real need. The proceedings of previous workshops are available from Springer as LNCS volumes 949, 1162, 1291, 1459, 1659, 1911, 2221, 2537, and 2862 (and since 1998 they have also been available online). We look forward to the next workshop in 2005, and perhaps even to the next decade of workshops!

September 2004

Dror Feitelson
Larry Rudolph
Uwe Schwiegelshohn

Table of Contents

Parallel Job Scheduling — A Status Report

Dror G. Feitelson[1], Larry Rudolph[2], and Uwe Schwiegelshohn[3]

[1] School of Computer Science and Engineering
The Hebrew University of Jerusalem
91904 Jerusalem, Israel
[2] Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139, USA
[3] Computer Engineering Institute
Universität Dortmund
44221 Dortmund, Germany

1 Introduction

The popularity of research on the scheduling of parallel jobs demands a periodic review of the status of the field. Indeed, several surveys have been written on this topic in the context of parallel supercomputers [17, 20]. The purpose of the present paper is to update that material, and to extend it to include work concerning clusters and the grid.

The paper is divided into three major parts. The first part addresses algorithmic and research issues covering the two main approaches: backfilling and gang scheduling. For each, recent advances are reviewed, both in terms of how to perform the scheduling and in terms of understanding the performance results. An underlying theme of the surveyed results is the shift from dogmatic use of rigid formulations to a more flexible approach. This reflects a maturation of the field and improved concern for real-world issues.

The second part of the paper addresses current usage. It presents a short overview of vendor offerings, and then reviews the scheduling frameworks used by top-ranking parallel systems. For vendor offerings, we highlight the distinction between what is done in a research setting and what is actually developed for production use. Regarding actual usage, we consider the alternative options of procurement of an existing system vs. the development of an in-house solution that more directly reflects desired attributes.

The third part of the paper looks both back and forward in time. As with any field, the success, popularity, and influence of a particular approach depends on a range of factors. We review some less successful ones. It is possible that some of these techniques may only be relevant to future machines. The paper, therefor concludes with some observations about the near-term future.

This paper contains a large number of references. In order to highlight the more recent results, i.e. those with publication dates in this millennium, their citation will be superscripted with the last two digits of the publication date.

D. Feitelson, L. Rudolph, and U. Schwiegelshohn (Eds.): JSSPP 2004, LNCS 3277, pp. 1–16, 2005.

2 Advances in Parallel Job Scheduling Research

There are many different ways to schedule parallel jobs and their constituent threads [17], but only a few mechanisms are used in practice and studied in detail. This section reviews backfilling and gang scheduling strategies, their variants, and their connections. The special requirements and strategies for scheduling parallel jobs on a grid are addressed as well.

2.1 Backfilling

The most basic batch scheduling algorithm is First-Come-First-Serve (FCFS) [43] where jobs are considered in order of arrival. Each job specifies the number of processors it requires and is placed in a FIFO queue upon arrival. If there are sufficient available processors to run the job at the head of the queue, the processors are allocated and the job is started. If there are not enough, the scheduler waits for some currently running job to terminate and free additional processors.

Backfilling is an optimization that tries to balance the goals of utilization and maintaining FCFS order. It requires that each job also specifies its maximum execution time. While the job at the head of the queue is waiting, it is possible for other, smaller jobs, to be scheduled, especially if they would not delay the start of the job on the head of the queue. Processors get to be used that would otherwise remain idle.

By letting some jobs execute out of order, other jobs may get delayed. Backfilling will never completely violate the FCFS order where some jobs are never run (a phenomenon known as "starvation"). In particular, jobs that need to wait are typically given a reservation for some future time.

The use of reservations was included in several early batch schedulers [29, 8]. Backfilling, in which small jobs move forward to utilize the idle resources, was introduced by Lifka [33]. This was done in the context of EASY, the Extensible Argonne Scheduling sYstem, which was developed for the first large IBM SP1 installation at Argonne National Lab.

Variations on Backfilling While the concept of backfilling is quite simple, it nevertheless has several variants with subtle differences. We generalize the behavior of backfilling by parameterizing several constants. Judicial choice of parameter values lead to improved performance.

One parameter is the *number of reservations*. In the original EASY backfilling algorithm, only the first queued job received a reservation. Jobs may be scheduled out of order only if they do not delay the job at the head of the queue. The scheduler estimates when a sufficient number of processors will be available for that job and reserves them for this job. Other backfilled jobs may not violate this reservation, they must either terminate before the time of the reservation (known as the "shadow time"), or use only processors that are not required by the first job [33].

Backfilling may cause delays in the execution of other waiting jobs (which are not the first, and therefore do not get a reservation). The obvious alternative is to make reservations for all jobs. This approach has been named "conservative backfilling" [37][01]. Simulation results indicate, however, that delaying other jobs is rarely a problem, and that conservative backfilling tends to achieve reduced performance in comparison with the more aggressive EASY backfilling. The MAUI scheduler includes a parameter that allows system administrators to set the number of reservations [30][01]. Chiang et al. suggest that making up to four reservations is a good compromise [6][02].

An intriguing recent suggestion is adaptive reservations depending on the extent different jobs have been delayed by previous backfilling decisions. If a job is delayed by too much, a reservation is made for this job [50][02]. This is essentially equivalent to the earlier "flexible backfilling", in which all jobs have reservations, but backfilling is allowed to violate these reservations up to a certain slack [51]. Setting the slack to the threshold used by adaptive reservations is equivalent to only making a reservation if the delay exceeds this threshold.

Another parameter is the *order of queued jobs.* The original EASY scheduler, and many other systems and designs, use a first come, first served (FCFS) order [33]. A general alternative is to prioritize jobs in some way, and select jobs for scheduling (including as candidates for backfilling) according to this priority order. Flexible backfilling combines three types of priorities: an administrative priority set to favor certain users or projects, a user priority used to differentiate among the jobs of the same user, and a scheduler priority used to guarantee that no job is starved [51]. The Maui scheduler has a priority function that includes even more components [30][01].

A special type of prioritization depends on job characteristics. In particular, Chiang et al. have proposed a whole set of criteria based on resource consumption, that are generalizations of the well-known Shortest Job First (SJF) scheduling algorithm [6][02]. These have been shown to improve performance metrics, especially those that are particularly sensitive to the performance of short jobs, such as slowdown.

A final parameter is the amount of *lookahead into the queue.* All previous backfilling algorithms consider the queued jobs one at a time, and try to schedule them. But the order in which jobs are scheduled may lead to loss of resources to fragmentation. The alternative is to consider the whole queue at once, and try to find the set of jobs that together maximize desired performance metrics. This can be done using dynamic programming, leading to optimal packing and improved performance [47][03].

Effect of User Runtime Estimates Backfilling depends on estimates of how long each job will run to figure out when additional processors will become available, and to verify that backfilled jobs will terminate in time so as not to violate reservations. The source of the estimates is typically the user who submits the job. Jobs that execute beyond their estimated runtime are usually

terminated by the system. Many users therefore regard these estimates as upper bounds, rather than as tight estimates.

Initial expectations were that user runtime estimates will nevertheless be tight, as low estimates improve the chance for backfilling. However, comparisons of user estimates with real runtimes show that they tend to be inaccurate, even when users are requested to provide their best possible estimate with no danger of having their job killed if the estimate is too low [18],[37]^{01},[32]^{04}. Attempts to derive better estimates automatically based on historical information from previous runs have not been successful, as they suffered from too many underestimations (which in backfilling would lead to killed jobs).

Probably, the most surprising result demonstrated by several studies has shown that inaccurate runtime estimates actually lead to *improved* average performance [18, 61],[37]^{01}. This is not simply the result of more backfilling due to more holes in the schedule, because inflated runtime estimates not only create holes in the schedule, but also enlarge potential backfill jobs, making it harder for them to fit into the holes. Rather, it is the result of a sequence of events where small backfill jobs prevent the holes from closing up, leading to a strong preference for short jobs and the automatic production of an SJF-like schedule [53]^{04}. This also motivates the construction of algorithms that explicitly favor short jobs such as those proposed by Chiang et al. [6]^{02}.

This does not necessarily indicate that more accurate runtime estimates are impossible and useless. Not all estimates are bad; in most cases, some users provide reasonably accurate estimates while others do not. Some studies indicate that those users who do provide reliable estimates do indeed benefit, as their jobs receive better service from the scheduler [6]^{02}. Also, while it seems that deriving good estimates automatically is not possible for all jobs, it might be possible to do so for short jobs and for jobs that have exhibited especially small variability in the past.

Incidently, inaccurate user runtime estimates have been shown to have surprising effects on performance evaluations [16]^{03},[15]^{03}. In a nutshell, it was seen that for workloads with numerous long single-process jobs, the inaccurate estimates allow for significant backfilling of these jobs under the aggressive EASY backfilling, but not under conservative backfilling. This in turn was detrimental for the performance of short jobs that were delayed by the long backfilled jobs. But if accurate estimates were used the effect was reversed, leading to a situation where short jobs were favored over long ones. This has more to do with evaluation methodology than will scheduling technology.

2.2 Gang Scheduling

The main alternative to batch scheduling is gang scheduling, where jobs are preempted and re-scheduled as a unit, across all involved processors. The notion was introduced by Ousterhout, using the analogy of a working set of memory pages to argue that a "working set" of processes should be co-scheduled for the application to make efficient progress [38]. Subsequent work emphasized gang

scheduling, which is an all-or-nothing affair, i.e. either all of the job's processes run or none do.

The point of gang scheduling is that it provides an environment similar to a dedicated machine, in which all a job's threads progress together, and at the same time allows resources to be shared. In particular, preemption is used to improve performance in face of unknown runtimes. This prevents short jobs from being stuck in the queue waiting for long ones, and improves fairness [44][00].

Flexible Algorithms One problem with gang scheduling is that the requirement that all a job's processes always run together causes too much fragmentation. This has led to several proposals for more flexible variants.

One such variant, called "paired gang scheduling" is designed to alleviate inefficiencies caused by I/O activity [56][03]. In conventional gang scheduling, processors running processes that perform I/O remain idle for the duration of the I/O operation. In paired gang scheduling jobs with complementary characteristics are paired together, so that when the processes of one perform I/O, those of the other can compute. Given a good job mix, this can lead to improved resource utilization at little penalty to individual jobs.

A more general approach is to monitor the communication behavior of all applications, and try to determine whether they really benefit for gang scheduling [24][03]. Gang scheduling is then used for those that need it. Processes belonging to other jobs are used as filler to reduce the fragmentation cause by the gang scheduled jobs.

Dealing with Memory Pressure Early evaluations of gang scheduling assumed that all arriving jobs can be started immediately. Under high loads this could lead to situations where dozens of jobs share each processor. This is unrealistic as all these jobs would need to be memory resident or else suffer from paging, which would interfere with the synchronization among the job's threads.

A simple approach for avoiding this problem is to use admission controls, and only allow additional jobs to start if enough memory is available [3][00]. An alternative is placing an oblivious cap on the multiprogramming level (MPL), usually in the range of 3–5 jobs [35]. While this avoids the need to estimate how much memory a new job will need, it is more vulnerable to situations in which memory becomes overcommitted causing excessive paging.

When admission controls are used and jobs wait in the queue the question of queue order presents itself. The simplest option is to use a FCFS order. Improved performance is obtained by using backfilling, and allowing small jobs to move ahead in the queue [59][00],[58][03]. In fact, using backfilling fully compensates for the loss of performance due to the limited number of jobs that are actually run concurrently [23][03].

All the above schemes may suffer from situations in which long jobs are allocated resources while short jobs remain in the queue and await their turn. The solution is to use a preemptive long-range scheduling scheme. With this construction, the long term scheduler allocates memory to waiting jobs, and

then the short term scheduler decides which jobs will actually run out of those that are memory resident. The long term scheduler may decide to swap out a job that has been in memory for a long time, to make room for a queued job that has been waiting for a long time. Such a scheme was designed for Tera (Cray) MTA machine [1].

System Integration The only commercially successful implementation of gang scheduling that we know of so far was the one on the Connection Machine CM-5. Other implementations, e.g. on the Intel Paragon, never moved beyond experimentation because of significant performance overheads, probably due to the cost of gang synchronization and coordination. Recent advances in the implementation of gang scheduling in experimental systems promise to reduce these overheads.

Gang scheduling requires the context switching to be synchronized across the nodes of the machine, and software-implemented synchronization on large machines is expensive. But some modern interconnection networks provide hardware support for global operations, and this can be exploited also in the runtime system. For example, in the STORM, where all parallel system activities are expressed in terms of three basic primitives, which in turn are supported by the hardware of the Quadrics network. In particular, this design has resulted in a very scalable implementation of gang scheduling [25][02].

While high performance networks enable efficient implementation of system primitives, they may cause problems with multiprogramming. The difficulty arises due to the use of user-level communication, in which user processes access the network interface cards (NICs) directly so as to avoid the overheads involved in trapping into the operating system. As a result no protection is available, and only one job can use the NICs. This can be solved by switching communication buffers as part of the gang scheduling's context switch operation [14][01]. It is also possible that this problem will be reduced in the future, as the memory available on NICs continues to grow.

Even tighter integration between communication and scheduling is used in the "buffered coscheduling" scheme proposed by Petrini and Feng [39][00],[40][00]. In this scheme the execution of all jobs is partitioned by the system into phases. In each phase, communication operations are buffered and at the end of the phase all the required communications is scheduled and and performed during the next phase. This leads to complete overlap of computation and communication.

Gang scheduling was originally developed in order to support fine-grain synchronization of parallel applications [19]. But an even greater benefit may be its contribution to reducing interference. The problem is that the nodes of parallel machines and clusters typically run a full operating system, with various user-level daemons that are required for various system services. These daemons may wake up at unpredictable times in order to perform their function. Obviously this interferes with the application process running on the node [36]. If such interferences are not synchronized across nodes, the application will be slowed considerably as different processes are delayed. But with gang scheduling

it is possible to run all the daemons on the different nodes at the same time, and eliminate their interference when user jobs are running [41][03]. When this is done, the full capabilities of the hardware are achieved.

2.3 Parallel Job Scheduling and the Grid

More recently, parallel computers are becoming part of a so called computational grid. The name grid has been chosen in analogy to the electrical power grid where several power plants provide numerous consumers with electrical power without the consumer being aware of the origin of the power. Similarly, it is the goal of a computational grid or simply Grid to allow users to run their jobs on any suitable computer belonging to the Grid. This way the computational load is balanced across many machines. Clearly, the Grid is mainly of interest for large computational jobs or jobs using a large data set as smaller jobs will usually run locally. However, the Grid is not restricted to this kind of jobs but will cover a wide range of general services. Nevertheless at the moment large computational jobs form the dominant grid application.

Before addressing the scheduling problem in a grid it is necessary to point out some differences between a parallel computer and the grid. A parallel computer has a central resource management system that can control all individual processors. However in a grid, the compute resources typically have different owners and as in most distributed systems there is no central control. Therefore, a compute resource typically has its own local resource management system that implements the policy of its owner. Hence, a grid scheduling architecture must be built on top of those existing local resource management systems. This requires communication between those different layers of the scheduling system in a grid [45][03],[55]. As in a distributed system the use of a central grid scheduler may result in a performance bottleneck and lead to a failure of the whole system if the scheduler fails. It is therefore appropriate to use a decentralized grid scheduler architecture and distributed algorithms.

Further, grid resources are heterogeneous in hardware and software which imposes constraints on the suitability of a resource for a given job. In addition, not every user may be accepted on every machine due to the implemented owner policy. A grid scheduler must determine which resources can be used for a specific submitted job while such a problem is usually not encountered in a parallel processor or even in a cluster of computers [10][02],[12][02]. Moreover, the grid is subject to frequent changes as some compute resources may be temporarily withdrawn from the grid due to maintenance or privileged non-grid use on request of the owner. To obtain these data, the grid scheduler needs a specific grid information service while the necessary up-to-date information is always assumed to be available in a parallel computer.

Today, the main purpose of grid computing is considered to be in the area of cross-domain load balancing. To support this idea the Globus Toolbox provides basic services that allow the construction of a grid scheduler [22]. With the help of those basic services grid schedulers are constructed that run on top of commercial resource management systems, like LSF, PBS or Loadleveler. Further, existing

Systems, like Condor [34, 42], are adapted to include grid scheduling abilities or allow integration with a grid scheduler.

If a parallel computer is embedded in a grid, a large variety of jobs from different users will be run on this machine. Then it will become increasingly difficult to implement the usage policy of an owner with the help of those simple scheduling criteria that are used today, like utilization and response time. Therefore, it can be assumed that the grid will also change job scheduling strategies for parallel computers. However in practice such an effect has not been observed yet.

Large grid application projects, like LCG, Datagrid, GriPhyn, frequently include the construction of some grid scheduler. Unfortunately, the scope of such a scheduler is usually restricted to the corresponding application project. On the other hand, there are academic projects that specifically address scheduling issues like the generation, distribution and selection of resource offerings. To this end various means are used, for instance economic methods.

In another approach, the job itself is responsible for its scheduling. Then we speak of an application scheduler. This is important for jobs which have a complex workflow and are subject to complex parallelization constraints. For example, this is the approach taken in the AppLes project [7][00].

As a continuation of some metacomputing ideas it is sometimes considered to use a computational grid as a single parallel processor, where many computational resources, that is parallel computers in the grid, are combined to solve a single very large problem . In this situation, the network performance varies greatly from communication within a parallel computer to communication between two parallel computers. Some models have been derived to evaluate the performance of so called multi site computing [26, 4, 9, 11, 13][03]. However in practice, such an approach has not been implemented with the possible exception of the preplanned combination of a few specific parallel computers for a specific purpose.

An important component of using the grid as a single parallel resource is co-allocation [5, 4, 2, 48][04]. This means that resources on several different machines need to be allocated to the same job at the same time. This is hard to accomplish due to the fact that the different resources belong to different owners, and do not have a common resource management infrastructure. The way to circumvent this problem is to try and reserve resources on the different machines, and then to use them only if all required reservations are successful [49][00].

3 Parallel Job Scheduling Practice

3.1 Vendor Offerings

Commercial scheduling software for parallel jobs comes in two types: portable, standalone systems, and components in a specific system.

There are two main competitors in the market for scheduling software. One is the Platform Computing Load Sharing Facility (LSF), which is based on the