Design of Efficient Java Communications for High Performance Computing

Guillermo López Taboada

Department of Electronics and Systems
University of A Coruña, Spain

Department of Electronics and Systems

University of A Coruña, Spain

PhD Thesis

Design of Efficient Java Communications for High Performance Computing

Guillermo López Taboada

May 2009

PhD Advisors:
Juan Touriño Domínguez
Ramón Doallo Biempica

Dr. Juan Touriño Domínguez
Catedrático de Universidad
Dpto. de Electrónica y Sistemas
Universidade da Coruña

Dr. Ramón Doallo Biempica
Catedrático de Universidad
Dpto. de Electrónica y Sistemas
Universidade da Coruña

CERTIFICAN

Que la memoria titulada "*Design of Efficient Java Communications for High Performance Computing*" ha sido realizada por D. Guillermo López Taboada bajo nuestra dirección en el Departamento de Electrónica y Sistemas de la Universidad de A Coruña y concluye la Tesis Doctoral que presenta para optar al grado de Doctor en Ingeniería Informática con la Mención de Doctor Europeo.

En A Coruña, a 26 de Febrero de 2009

Fdo.: Juan Touriño Domínguez
Director de la Tesis Doctoral

Fdo.: Ramón Doallo Biempica
Director de la Tesis Doctoral

V^o B^o: Luis Castedo Ribas
Director del Dpto. de Electrónica y Sistemas

Resumen de la Tesis

Introducción

El lenguaje de programación Java se ha convertido rápidamente en uno de los más populares tras su aparición a mediados de los años noventa. Entre las causas de su pronta aceptación en numerosos ámbitos de aplicación están el tratarse de un lenguaje orientado a objetos, ser independiente de la plataforma, su portabilidad, ser inherentemente seguro, contar con una API muy extensa, y finalmente figurar como el principal lenguaje de programación por número de desarrolladores y por formación ofertada, tanto en el mundo académico como en el profesional. Su penetración en el mundo de la web e Internet es especialmente importante, tanto en aplicaciones cliente-servidor como en computación distribuida. No obstante, la presencia de Java no está restringida a estos ámbitos, siendo una opción consolidada en muchos otros (por ejemplo, en aplicaciones multimedia y de escritorio) e incluso una alternativa emergente en la actualidad para su aplicación en computación de altas prestaciones (High Performance Computing o HPC).

El creciente interés en Java para computación paralela está motivado por las especiales características del lenguaje, de enorme utilidad en este ámbito de aplicación. Así, su inherente soporte para aplicaciones en red, el tratarse de un lenguaje multithread (lo cual permite aprovechar de forma directa los sistemas multi-núcleo), y el ser portable y multiplataforma, entre otras ventajas de Java, están contribuyendo a su adopción en el campo de la computación de altas prestaciones. El principal inconveniente de Java en este ámbito ha sido, hasta ahora, su rendimiento inferior al de los lenguajes que son compilados a código nativo de cada plataforma, los cuales están más establecidos en este campo, especialmente Fortran y C. No obstante, en la actualidad el rendimiento ya no constituye un obstáculo insalvable. La tradicional diferencia de rendimiento entre Java y los lenguajes compilados específicamente para cada plataforma se ha ido estrechando en los últimos años gracias al Just-In-Time (JIT), la técnica de compilación de la Máquina Virtual de Java (JVM) que proporciona al bytecode de Java rendimientos comparables al del código nativo.

Sin embargo, aunque Java obtiene rendimientos similares en códigos secuenciales a los lenguajes compilados, en aplicaciones paralelas su escalabilidad depende en gran medida de la intensidad y eficiencia de las comunicaciones y/o accesos a memoria compartida. De hecho, la inexistencia de mecanismos de comunicación eficientes en Java tiene como consecuencia que sus aplicaciones paralelas presenten peor escalabilidad que las que utilizan código nativo, ralentizando de este modo el desarrollo de soluciones Java en computación de altas prestaciones.

En cuanto a la arquitectura y configuración de los sistemas utilizados en HPC cabe destacar que actualmente se está incrementando significativamente el número de núcleos de procesador instalados a fin de satisfacer la creciente demanda de potencia de cálculo. Además, las arquitecturas clúster siguen siendo las más demandadas, debido a su excelente ratio coste/rendimiento. Finalmente, la necesidad de que las aplicaciones escalen con un mayor número de núcleos favorece la utilización de redes de baja latencia (por ejemplo InfiniBand, Myrinet o SCI). Este hecho, la popularización de los clusters con redes de baja latencia y con un número cada vez mayor de núcleos por nodo, subraya la importancia del paralelismo y de la programación multithread e híbrida utilizando ambos paradigmas de memoria compartida/distribuida.

Así, el actual escenario demanda lenguajes de programación que permitan un desarrollo más productivo y que proporcionen un completo soporte multithread y de servicios de red. Además, la agregación de nodos multi-núcleo en clusters por medio de redes de altas prestaciones debe ir pareja del uso de un middleware de comunicación que permita a las aplicaciones aprovechar el rendimiento de dichas redes. El lenguaje de programación Java puede satisfacer estas demandas y constituir una alternativa atractiva para la programación paralela en computación de altas prestaciones, siempre y cuando sea capaz de dotarse de mecanismos de comunicación eficientes en las actuales arquitecturas de computación.

La presente Tesis Doctoral, *"Design of Efficient Java Communications for High Performance Computing"*, parte de la hipótesis inicial de que es posible desarrollar aplicaciones Java en computación de altas prestaciones, un ámbito en el que el rendimiento es crucial, siempre que esté disponible un middleware de comunicación eficiente. Entre los objetivos de esta Tesis se hayan la evaluación del rendimiento de Java en el marco de la computación de altas prestaciones y proporcionar nuevas herramientas para mejorar este proceso de evaluación y posterior análisis de los resultados. Además, otra finalidad de este trabajo es el diseño y desarrollo de un middleware de comunicación optimizado para Java, basado en sockets, así como dotar a Java de bibliotecas de comunicación que proporcionen mayores rendimientos que alternativas previamente existentes. Entre los desarrollos implementados destacan un middleware de comunicación no bloqueante a bajo nivel, una biblioteca de paso de mensajes en Java, y una optimización del protocolo de la Invocación de Métodos Remotos (RMI) para Java. Finalmente, la Tesis proporciona una guía de recomendaciones y buenas prácticas en optimización del rendimiento de aplicaciones Java en computación de altas prestaciones.

Metodología de Trabajo

La metodología de trabajo seguida en el desarrollo de la presente Tesis Doctoral, ha consistido en:

- Definir la lista de tareas a realizar en el tema de la Tesis, teniendo en cuenta los trabajos previos y los recursos disponibles.

- Determinar su secuencia u orden de ejecución, ateniéndose a las restricciones que pudiesen existir y al orden más favorable.

- Establecer su duración y la oportunidad de su desarrollo en un momento determinado.

- Organizar las acciones o tareas por bloques de cierta entidad que definan etapas.

- Definir, para cada etapa, las metas a alcanzar (u objetivos concretos a lograr en tiempo definidos), sabiendo que en cada etapa puede haber una o varias metas.
- Fijar para cada meta, dentro de cada etapa, la metodología de trabajo a emplear para lograrla (revisión bibliográfica, evaluación de proyectos previos, análisis de la eficiencia de los desarrollos existentes en el ámbito de la Tesis, análisis de aportaciones necesarias en el ámbito de trabajo, diseño e implementación de soluciones más eficientes en el área de investigación, evaluación de las aportaciones realizadas y, en caso de ser de positivas y de entidad, difusión posterior a través de congresos y revistas del ámbito de conocimiento; y, finalmente, recopilación de las principales conclusiones en la presente memoria de Tesis).

De este modo, la lista de tareas (*Tn*), agrupadas en bloques (**Bn**), desarrolladas en la presente Tesis han sido:

B1 Estado actual de las comunicaciones en Java para computación de altas prestaciones.

T1.1 Selección del paradigma más apropiado para la computación de altas prestaciones en Java. Se seleccionó el paso de mensajes debido a la mayor escalabilidad de los sistemas de memoria distribuida, en los cuales constituye el principal paradigma de programación. Además, las bibliotecas de paso de mensajes proporcionan, en general, rendimientos aceptables y Java cuenta con numerosos proyectos de bibliotecas de paso de mensajes.

T1.2 Revisión de la literatura y proyectos desarrollados en Java para paso de mensajes.

T1.3 Evaluación del rendimiento de las bibliotecas existentes de paso de mensajes en Java.

T1.4 Obtención de modelos analíticos de rendimiento de comunicaciones colectivas en paso de mensajes en Java. Estos modelos se obtuvieron en tres clusters, con redes de interconexión Fast Ethernet y las redes de baja latencia SCI y Myrinet.

T1.5 Evaluación y modelado analítico de comunicaciones MPI utilizando las tres redes de interconexión mencionadas en el punto anterior con finalidad comparativa.

T1.6 Identificación de los principales problemas de rendimiento detectados: pobre rendimiento de las bibliotecas de comunicaciones de Java en clusters, especialmente si se utiliza RMI, bibliotecas de paso de mensajes en Java poco desarrolladas y con operaciones colectivas poco escalables.

B2 Análisis y desarrollo de un middleware de comunicaciones más eficiente en Java.

T2.1 Determinación de la aproximación a seguir en el diseño de mecanismos de comunicación eficientes en Java. Se selecciona una estrategia de abajo a arriba, comenzando por el desarrollo de una biblioteca de comunicación más eficiente en Java a bajo nivel (Java sockets).

T2.2 Análisis y diseño de necesidades de una implementación más eficiente de sockets en Java (soporte de redes de baja latencia, un protocolo con menos copias de datos, tanto en Java como en las funciones nativas del sistema utilizadas, y reducción del coste que supone la serialización de los datos que han de ser transferidos en Java).

T2.3 Desarrollo de un primer prototipo de solución: implementación de una biblioteca de sockets de alto rendimiento en Java sobre TCP/IP.

T2.4 Desarrollo de la solución sobre una red de baja latencia: implementación de una biblioteca de sockets en Java sobre SCI.

T2.5 Implementación de la biblioteca de sockets para Myrinet e InfiniBand.

T2.6 Desarrollo de un protocolo de comunicación a través de sockets en sistemas de memoria compartida, para permitir que aplicaciones basadas en sockets mejoren su rendimiento en sistemas multi-núcleo.

T2.7 Identificación de necesidades en comunicaciones en Java: soporte más eficiente de comunicaciones no bloqueantes sobre Java IO sockets.

T2.8 Análisis, diseño e implementación de una biblioteca de comunicaciones no bloqueantes en Java, tanto sobre el API estándar de Java sockets como sobre la biblioteca de sockets de alto rendimiento.

- *T2.9* Optimización del protocolo Java RMI, con el objeto de trasladar a las aplicaciones basadas en RMI los beneficios del middleware desarrollado.
- *T2.10* Evaluación del rendimiento del middleware de comunicaciones desarrollado, especialmente en aplicaciones paralelas en Java.

B3 Análisis y desarrollo de una biblioteca de paso de mensajes en Java.

- *T3.1* Análisis, diseño y desarrollo de una biblioteca de paso de mensajes basada en el middleware desarrollado. Se implementa un subconjunto importante y perfectamente funcional de la API de mpiJava 1.2, la más utilizada ya que mpiJava y MPJ Express, dos de las bibliotecas de paso de mensajes en Java más extendidas, la implementan.
- *T3.2* Desarrollo de primitivas de comunicación colectiva para paso de mensajes en Java utilizando algoritmos más escalables.
- *T3.3* Validación de las bibliotecas desarrolladas.
- *T3.4* Desarrollo de benchmarks en Java para paso de mensajes.
- *T3.5* Evaluación del rendimiento de Java con los benchmarks implementados y haciendo uso del middleware y las bibliotecas desarrolladas en esta Tesis, con especial atención al impacto que presentan en el rendimiento final.
- *T3.6* Documentación y difusión del trabajo desarrollado en la aplicación de las bibliotecas desarrolladas en computación de altas prestaciones en Java.

B4 Determinación de las principales conclusiones y líneas de trabajo futuras.

- *T4.1* Determinación de las principales conclusiones.
- *T4.2* Evaluación de las principales líneas de investigación abiertas a raíz del trabajo desarrollado.
- *T4.3* Redacción de la memoria final de la Tesis Doctoral.

El trabajo llevado a cabo en estas tareas ha sido recogido en la presente memoria. Así, las tareas del primer bloque han sido desarrolladas en los capítulos 1 y 2. El segundo bloque constituye los capítulos 3 y 4. Las tareas del tercer bloque están recogidas en los capítulos 5 y 6. Finalmente, el cuarto bloque está incluido dentro del último capítulo, el de las conclusiones.

Asimismo, la lista de metas (*M*) asociadas con cada bloque (**B**) de la Tesis Doctoral ha sido:

B1 Estado actual de las comunicaciones en Java para computación de altas prestaciones.

- *M1.1* Evaluación actual de los proyectos existentes en comunicaciones en Java para computación de altas prestaciones, tanto del rendimiento proporcionado como de sus principales aportaciones y carencias.
- *M1.2* Mejora de las herramientas disponibles para dicha evaluación (nuevos modelos de rendimiento, técnicas de evaluación y desarrollo de nuevos benchmarks).

B2 Análisis y desarrollo de un middleware de comunicaciones más eficiente en Java.

- *M2.1* Middleware de comunicaciones en Java que del modo más transparente posible para el usuario proporcione mejores rendimientos tanto en redes de baja latencia (SCI, Myrinet, InfiniBand) como en sistemas de memoria compartida (sistemas multi-núcleo). Además, debe extender las funcionalidades existentes en la actualidad en Java, proporcionando comunicaciones no bloqueantes sobre la API de Java IO sockets.
- *M2.2* Desarrollo de un protocolo optimizado de Java RMI.

B3 Análisis y desarrollo de una biblioteca de paso de mensajes en Java.

- *M3.1* Implementación de una biblioteca de paso de mensajes en Java que mejore el rendimiento de otras bibliotecas existentes aprovechando el middleware de comunicaciones especificado en la meta *M2.1*.
- *M3.2* Evaluación en términos de impacto en el rendimiento final de aplicaciones Java de los proyectos desarrollados en el marco de esta Tesis, así como de sus principales aportaciones y carencias.

B4 Determinación de las principales conclusiones y líneas de trabajo futuras.

- *M4.1* Memoria final de la Tesis Doctoral que recoge las principales conclusiones y líneas futuras de investigación.

Los medios necesarios para realizar esta Tesis Doctoral, siguiendo la metodología de trabajo anteriormente descrita, han sido los siguientes:

- Material de trabajo y financiación económica proporcionados fundamentalmente por el Grupo de Arquitectura de Computadores de la Universidade da Coruña, junto con la Xunta de Galicia (beca predoctoral), el Ministerio de Educación y Ciencia (beca FPU AP2004-5984) y la Universidade da Coruña (contrato de profesor ayudante).
- Además, esta Tesis se ha financiado a través de los siguientes proyectos de investigación:
    - De financiación estatal (Ministerios de Ciencia y Tecnología, Educación y Ciencia, y Ciencia e Innovación) a través de los proyectos TIC2001-3694-C02-02, TIN2004-07797-C02, y TIN2007-67537-C03-02, además de la mencionada beca FPU AP2004-5984.
    - De financiación autonómica a través de los proyectos PGIDT01-PXI10501-PR, PGIDIT02-PXI10502IF, PGIDIT05PXIC10504PN, PGIDIT06PXIB-105228PR, Programa de Consolidación de Grupos de Investigación Competitivos (Ref. 3/2006 DOGA 13/12/2006) y Red Gallega de Computación de Altas Prestaciones (Grupos de Redes de Investigación 2007/147), además de la beca predoctoral.
- Financiación del Ministerio de Ciencia e Innovación de una ayuda de movilidad para la participación de profesores en tribunales de Tesis convocados para la mención europea en el título de doctor, TME2008-00739 (BOE 12/1/2009).
- Acceso a material bibliográfico, a través de la biblioteca de la Universidade da Coruña.
- Acceso al software desarrollado en proyectos previos de comunicaciones en Java.
- Acceso a clusters con redes de baja latencia y/o múltiples procesadores/núcleos por nodo:
    - Clúster *irixoa* (Univ. de A Coruña, 2002). 10 nodos Pentium III 933 MHz con 512 MB RAM interconectados mediante Fast Ethernet.

- Clúster *bw* (Centro de Supercomputación de Galicia, CESGA, 2003-2006). 16 nodos Pentium III a 1 GHz con 512 MB RAM interconectados mediante Myrinet.

- Clúster *muxia* (Univ. de A Coruña, 2003-actualidad). Inicialmente 8 nodos dual Xeon 1.8 GHz y 1 GB RAM interconectados mediante redes SCI y Gigabit Ethernet. Posteriormente se han incorporado 8 nodos dual Xeon a 2.8/3.2 GHz con 2 GB de RAM (2003-2004-2005) y 8 nodos dual Xeon de doble núcleo a 3.2 GHz y 4 GB de RAM (2006). La instalación del clúster se financió gracias a los proyectos PGIDT01-PXI10501PR y PGIDIT02-PXI10502IF, mientras que las sucesivas ampliaciones han sido a cargo de los proyectos TIC2001-3694-C02-02, TIN2004-07797-C02 y PGIDIT05PXIC10504PN.

- Clúster *starbug* (Univ. de Portsmouth, Reino Unido, 2005-2006). Clúster con 8 nodos dual Xeon a 2.8 GHz y 2 GB de RAM interconectados mediante SCI y Myrinet.

- Supercomputador *Finis Terrae* (CESGA, 2008-actualidad). 144 nodos con 16 núcleos de procesador Itanium2 Montvale a 1.6 GHz y 128 GB RAM interconectados mediante InfiniBand, además de contar con un sistema Superdome de memoria compartida con 128 núcleos Itanium2 Montvale a 1.6 GHz y 1 TB RAM.

- Dos estancias de investigación en el Distributed Systems Group de la Universidad de Portsmouth, Reino Unido, bajo la supervisión del Profesor Mark Baker. Estas estancias, de 3.5 meses en 2005 y de 4 meses en 2006, permitieron el desarrollo de las tareas T2.5 y T2.8. Además, brindaron la oportunidad de establecer una relación de colaboración con los Profesores Bryan Carpenter y Aamir Shafi, desarrolladores de buena parte de los proyectos más relevantes en computación de altas prestaciones en Java, como son HPJava y muy especialmente las bibliotecas de paso de mensajes en Java mpiJava y MPJ Express. La financiación de ambas estancias de investigación fue obtenida en concurrencia competitiva en sendas convocatorias públicas de la Xunta de Galicia y del Ministerio de Educación y Ciencia.

Conclusiones

La presente Tesis Doctoral, *"Design of Efficient Java Communications for High Performance Computing"*, ha puesto de manifiesto que es posible desarrollar aplicaciones Java en computación de altas prestaciones si se dispone de middleware y bibliotecas de comunicación eficientes. El análisis del estado del arte en este tema puso de manifiesto que Java carecía de dicho middleware y bibliotecas, lo cual le impedía obtener rendimientos escalables en computación de altas prestaciones. Una evaluación más profunda de la situación realizada con nuevos benchmarks y modelos desarrollados en esta Tesis identificó las causas de la falta de eficiencia en las comunicaciones:

- Carencia de soporte directo en redes de baja latencia.
- Realización de copias de datos prescindibles en transferencias entre Java y las bibliotecas de comunicaciones nativas del sistema.
- Serialización de los datos a transmitir, siendo esta operación muy costosa.
- Carencia de soporte eficiente a las comunicaciones no bloqueantes con Java sockets IO.
- Utilización de protocolos de comunicación no adaptados a clusters HPC.

Una vez analizadas las causas de la ineficiencia de las comunicaciones en Java se procedió a diseñar soluciones que incrementasen su rendimiento, desarrollando un middleware de comunicación eficiente para Java, denominado Java Fast Sockets (JFS), una implementación de sockets de alto rendimiento que proporciona soporte a redes de baja latencia (InfiniBand, Myrinet, SCI) e incrementa el rendimiento de sistemas de memoria compartida. Además, se diseñó e implementó una biblioteca, iodev, que proporciona comunicaciones no bloqueantes en Java, y que sirvió de base para el desarrollo de una biblioteca de paso de mensajes en Java. Dicha biblioteca, denominada F-MPJ, implementa algoritmos de operaciones colectivas más escalables que otras implementaciones de paso de mensajes previas. Finalmente, la optimización del protocolo de Java RMI para clusters con redes de baja latencia completó el elenco de soluciones más eficientes para comunicaciones en Java.

El análisis del impacto de las soluciones desarrolladas en el rendimiento final de las aplicaciones paralelas en Java ha validado la hipótesis inicial, ya que ha sido posible aumentar significativamente la escalabilidad de las aplicaciones Java en el ámbito de la computación de altas prestaciones. No obstante, a la hora de implementar aplicaciones paralelas en Java es importante la optimización del rendimiento del código Java, para lo cual también se ha proporcionado una guía de "buenas prácticas" para desarrollar códigos más eficientes, recopiladas durante el desarrollo de los micro-benchmarks, benchmarks de kernels y aplicaciones utilizados en las evaluaciones de rendimiento realizadas en esta Tesis.

Conviene destacar que aunque se ha demostrado que es posible el uso de Java en HPC, es necesario seguir mejorando y extendiendo los proyectos iniciados en el presente trabajo. Así, es de sumo interés el desarrollo de nuevas implementaciones de sockets de alto rendimiento en Java, como los sockets SCTP (Stream Control Transmission Protocol). Además, la orientación a mensajes del protocolo SCTP lo hacen especialmente indicado para la implementación de bibliotecas de paso de mensajes sobre él. En este sentido, sería importante el completar la implementación de la API de mpiJava 1.2 en F-MPJ. La integración de JFS y el protocolo RMI optimizado en otros middleware de comunicación es también altamente interesante. De este modo, sería posible extender de forma significativa el alcance de las mejoras de rendimiento obtenidas con las bibliotecas desarrolladas en esta Tesis.

Para concluir conviene tener presente que el auge de los sistemas multi-núcleo demanda un mayor desarrollo de las soluciones en memoria compartida. Por tanto, el desarrollo de una implementación eficiente de OpenMP en Java debe abordarse de forma apremiante. Así sería posible aunar las optimizaciones desarrolladas para bibliotecas de memoria distribuida (JFS, iodev, RMI optimizado, F-MPJ) con protocolos de memoria compartida, permitiendo explotar al máximo la arquitectura de los clusters multi-núcleo: la comunicación en memoria compartida se realizaría intra-nodo mientras que las bibliotecas de comunicación eficientes serían utilizadas para transferencias inter-nodo.

Principales Contribuciones

Las principales aportaciones de esta Tesis son:

- Una evaluación actual de Java para computación de altas prestaciones [61, 94], con especial énfasis en su rendimiento en clusters con nodos con múltiples procesadores (multi-core o con hyper-threading) interconectados mediante redes de baja latencia (InfiniBand, Myrinet y SCI).

- La mejora de las herramientas disponibles para evaluar el rendimiento de Java para programación paralela al proporcionar: (1) una suite de micro-benchmarks de operaciones colectivas en paso de mensajes [87], (2) un modelo analítico más exacto del rendimiento de dichas operaciones [94], el cual ha permitido caracterizar de forma más precisa el rendimiento de las comunicaciones en Java para paso de mensajes sobre clusters de baja latencia [94], y (3) una implementación en Java para paso de mensajes de los NAS Parallel Benchmarks [61], la suite de benchmarks estándar para la evaluación del rendimiento de sistemas y lenguajes en computación de altas prestaciones.

- El diseño y desarrollo de una biblioteca de sockets de alto rendimiento en Java, Java Fast Sockets (JFS) [92]. El uso de JFS permite, de un modo transparente para el usuario y las aplicaciones, incrementar el rendimiento de las comunicaciones en Java. Así, el protocolo de comunicaciones implementado en JFS reduce el número de copias de datos necesarias para realizar una comunicación, evita la serialización de tipos de datos primitivos y proporciona el soporte necesario para aprovechar la baja latencia y los elevados anchos de banda tanto de redes de altas prestaciones como InfiniBand, Myrinet y SCI, como de la memoria compartida en sistemas multi-núcleo.

- El diseño e implementación de un middleware de comunicación en Java que soporta de forma eficiente comunicaciones no bloqueantes sobre la API de sockets IO, iodev [93], permitiendo de este modo aprovechar el rendimiento proporcionado por JFS, en el caso de que esta biblioteca esté disponible en el sistema.

- El diseño y desarrollo de una implementación más eficiente de RMI para clusters homogéneos [86], gracias a un protocolo más eficiente que reduce la cantidad de información a transmitir, así como el coste de las comunicaciones mediante el uso de JFS. El impacto del uso de este protocolo es muy importante al estar basadas en RMI un amplio número de bibliotecas y aplicaciones distribuidas.

- El diseño e implementación de una biblioteca de paso de mensajes en Java, Fast MPJ (F-MPJ) [93], que proporciona una mayor escalabilidad a las aplicaciones paralelas en Java gracias al uso de iodev y a su implementación de las operaciones colectivas utilizando algoritmos más eficientes.

Publications from the Thesis

Journal Papers (3)

- G. L. Taboada, J. Touriño, and R. Doallo. F-MPJ: Scalable Java Message-passing Communications on Parallel Systems. *Journal of Supercomputing,* 2009 (In press).
- G. L. Taboada, J. Touriño, and R. Doallo. Performance Analysis of Message-Passing Libraries on High-Speed Clusters. *Intl. Journal of Computer Systems Science & Engineering,* 2009 (In press).
- G. L. Taboada, J. Touriño, and R. Doallo. Java Fast Sockets: Enabling High-speed Java Communications on High Performance Clusters. *Computer Communications*, 31(17):4049-4059, 2008.

International Conferences (9)

- G. L. Taboada, J. Touriño, R. Doallo, Y. Lin and J. Han. Efficient Java Communication Libraries over InfiniBand. In *Proc. 11th IEEE Intl. Conf. on High Performance Computing and Communications (HPCC'09)*, Seoul, Korea, 2009.
- D. A. Mallón, G. L. Taboada, J. Touriño, and R. Doallo. NPB-MPJ: NAS Parallel Benchmarks Implementation for Message-Passing in Java In *Proc. 17th Euromicro Intl. Conf. on Parallel, Distributed, and Network-Based Processing (PDP'09)*, pages 181-190, Weimar, Germany, 2009.

- G. L. Taboada, C. Teijeiro, and J. Touriño. High Performance Java Remote Method Invocation for Parallel Computing on Clusters. In *Proc. 12th IEEE Symposium on Computers and Communications (ISCC'07)*, pages 233-239, Aveiro, Portugal, 2007.

- G. L. Taboada, J. Touriño, and R. Doallo. High Performance Java Sockets for Parallel Computing on Clusters. In *Proc. 9th Intl. Workshop on Java and Components for Parallelism, Distribution and Concurrency (IWJacPDC'07)*, page 197b (8 pages), Long Beach, CA, USA, 2007.

- G. L. Taboada, J. Touriño, and R. Doallo. Non-blocking Java Communications Support on Clusters. In *Proc. 31st IEEE Conf. on Local Computer Networks (LCN'06)*, pages 264-271, Tampa, FL, USA, 2006.

- G. L. Taboada, J. Touriño, and R. Doallo. Non-blocking Java Communications Support on Clusters. In *Proc. 13th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'06), Lecture Notes in Computer Science vol. 4192*, pages 256-265, Bonn, Germany, 2006.

- G. L. Taboada, J. Touriño, and R. Doallo. Designing Efficient Java Communications on Clusters. In *Proc. 7th Workshop on Java for Parallel and Distributed Computing (IWJPDC'05)*, page 192 (8 pages), Denver, CO, USA, 2005.

- G. L. Taboada, J. Touriño, and R. Doallo. Performance Analysis of Java Message-Passing Libraries on Fast Ethernet, Myrinet and SCI Clusters. In *Proc. 5th IEEE Intl. Conf. on Cluster Computing (CLUSTER'03)*, pages 118-126, Hong Kong, China, 2003.

- G. L. Taboada, J. Touriño, and R. Doallo. Performance Modeling and Evaluation of Java Message-Passing Primitives on a Cluster. In *Proc. 10th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'03), Lecture Notes in Computer Science vol. 2840*, pages 29-36, Venice, Italy, 2003.

National Conferences (4)

- D. A. Mallón, G. L. Taboada, J. Touriño, and R. Doallo. Implementación de los NAS Parallel Benchmarks en Java con Paso de Mensajes. In *Actas de las XIX Jornadas de Paralelismo*, pages 405-410, Castellón, Spain, 2008.
- G. L. Taboada, J. Touriño, and R. Doallo. Bibliotecas de Comunicación Eficiente en Clusters para Códigos Java. In *Actas de las XVII Jornadas de Paralelismo*, pages 407-412, Albacete, Spain, 2006.
- G. L. Taboada, J. Touriño, and R. Doallo. Configuración y Evaluación de Bibliotecas de Paso de Mensajes Java en Clusters. In *Actas de las XIV Jornadas de Paralelismo*, pages 193-198, Leganés, Spain, 2003.
- G. L. Taboada. Estudio de la Programación de Clusters mediante Bibliotecas de Paso de Mensajes en Java. In *Actas del I Certamen Arquímedes de Introducción a la Generación de Conocimiento*, pages 1-24, Valencia, Spain, 2002.

Technical Reports (1)

- B. Amedro, V. Bodnartchouk, D. Caromel, C. Delbe, F. Huet, and G. L. Taboada. Current State of Java for HPC. In *INRIA Technical Report RT-0353*, pages 1–24, INRIA Sophia Antipolis, Nice, France, 2008, http://hal.inria.fr/-inria-00312039/en/.

Abstract

There is an increasing interest to adopt Java as the parallel programming language for the multi-core era. This interest demands scalable performance on hybrid shared/distributed memory architectures. Although Java offers important advantages, such as built-in multithreading and networking support, security, widespread use, high programming productivity, portability and platform independence, the lack of efficient communication middleware is an important drawback for its uptake in High Performance Computing (HPC). This PhD Thesis presents the design, implementation and evaluation of several solutions to improve this situation. For this, it has been designed, developed and evaluated a high performance Java sockets implementation (named JFS, Java Fast Sockets), in order to take advantage of high-speed networks (SCI, Myrinet, InfiniBand) and shared memory (e.g., multi-core) machines. Moreover, an efficient non-blocking communication support is provided through the development of the iodev library, which allows to overlap communication and computation. Finally, a Java message-passing library, Fast MPJ (F-MPJ), has been implemented, which provides more scalable algorithms in collective communication primitives. These middleware and libraries have been used in Java communication protocols (e.g., RMI) and eventually in parallel applications, enhancing their performance, especially thanks to the avoidance of the serialization of primitive data types, commonly used in HPC. Furthermore, a collection of good programming practices for performance have been gathered from the implementation of new Java parallel codes, used for the performance evaluation and validation of the middleware and libraries developed in this Thesis. The final and main conclusion is that the use of Java for HPC is feasible, and even advisable when looking for productive development, provided that efficient communication middleware is made available, such as the projects presented in this Thesis, and following the guidelines for performance optimization also suggested in this work.

A Karina
meu Amor

A meus pais
por tanto

Acknowledgments

This Thesis is not the result only of my own effort; there are many people involved in this work whose support and dedication I want to acknowledge. First and foremost, I want to acknowledge my PhD advisors Juan and Ramón for the confidence they placed in me, and the support they gave me during all these years. I cannot forget my colleagues in the Computer Architecture Group because they made my experience in the lab easier during all these years. Furthermore, Javier, Toño, o padriño, a madriña e a abuelita have always been there for me.

I gratefully thank Mark Baker for hosting me during my visits to the Distributed Systems Group, University of Portsmouth, UK, and Bryan Carpenter and especially Aamir Shafi for working side by side with me during that time and for their valuable comments. I also thank Denis Caromel and his group for their valuable collaboration.

I also gratefully thank CESGA (Supercomputing Center of Galicia, Santiago de Compostela, Spain) for providing access to the Finis Terrae supercomputer and the Myrinet cluster.

Finally, but not least, I am thankful to the following institutions for funding this work: the Computer Architecture Group and the Department of Electronics and Systems for the human and material support, the University of A Coruña for giving me the opportunity of starting a career in academia, and for financing my attendance at some conferences, the Galician Government (Xunta de Galicia) for the projects PGIDT01PXI10501PR, PGIDIT02PXI10502IF, PGIDIT05PXIC10504PN, PGIDIT06PXIB105228PR, Consolidation Program of Competitive Research Groups (Ref. 3/2006 DOGA 13/12/2006), High Performance Computing Galician Thematic Network (2007/147), and a Pre-Doctoral Grant, and the Ministry of Education and Science of Spain for the projects TIC2001-3694-C02-02, TIN2004-07797-C02-02 and TIN2007-67537-C03-02, the FPU Grant AP2004-5984, and for a mobility grant for inviting recognized professors to serve on the Thesis committee TME2008-00739.

Guillermo López Taboada

El necio es atrevido y el sabio comedido
(Fools rush in where angels fear to tread)
Spanish proverb

Contents

List of Tables

List of Figures

Preface

Java has become a leading programming language soon after its release, especially in web-based and distributed computing environments, and it is an emerging option for High Performance Computing (HPC) [3, 57, 82]. The increasing interest in Java for parallel computing is based on its appealing characteristics: built-in networking and multithreading support, object orientation, platform independence, portability, security, it has an extensive API and a wide community of developers, and finally, it is the main training language for computer science students. Moreover, performance is no longer an obstacle. The gap between Java and native languages performance has been narrowing over the last few years, thanks to the Just-in-Time (JIT) compiler of the Java Virtual Machine (JVM) that obtains native performance from Java bytecode. Nevertheless, although the performance gap is usually small for sequential applications, it can be particularly large for parallel applications when depending on communications performance. The main reason is the lack of efficient Java communication middleware, which has hindered Java adoption for HPC.

Regarding HPC platforms, new deployments are increasing significantly the number of cores installed in order to meet the ever growing computational power demand. This current trend to multi-core clusters underscores the importance of parallelism and multithreading capabilities [30]. Therefore, this scenario requires scalable parallel solutions, where communication efficiency is fundamental. This efficiency not only depends heavily on the use of high-speed networks, such as InfiniBand [45], Myrinet [14] or SCI [41], but more and more on the communication middleware [104]. Furthermore, hybrid systems (shared/distributed memory architectures) increase the complexity of communication protocols as they have to combine inter-node and intra-node communications, which may imply efficient communication overlapping. Hence, Java represents an attractive choice for the development of communication middleware for these systems as it is a multithreaded language, supports the heterogeneity of the systems and can rely on efficient communication middleware that provides support on high performance communication hardware. Thus, Java can take full advantage of hybrid architectures using shared memory for intra-node communication and relying on efficient inter-node communication.

Work Methodology

The present PhD Thesis, *"Design of Efficient Java Communications for High Performance Computing"*, deals with the initial hypothesis that it is possible to develop Java applications for High Performance Computing (HPC), where performance is essential, provided that an efficient communication middleware is made available. Thus, the main objective of this work is the design and development of such middleware for HPC.

The methodology used in this Thesis begins with the identification of the main causes of inefficiency in this field, through an extensive evaluation of Java performance in HPC. At this point, it is expected that new benchmarks and models have to be developed as the evaluation of emerging solutions usually lacks such facilities. Next, the work follows a bottom-up approach. Thus, it is first targeted at the design and development of more efficient Java middleware, based on Java sockets, as well as providing non-blocking support on Java IO sockets. Among the criteria for selecting the API or protocol to be implemented are the lack of efficient implementations, a wide range of applicability of the optimized middleware, and its user and application transparency. The selection of Java IO sockets meets these criteria, as well as the Java Remote Method Invocation (RMI) protocol and the implementation of a more efficient Message-Passing in Java (MPJ) library. Thus, an optimized Java RMI protocol for high-speed networks clusters and a more scalable MPJ library are implemented on top of the previously developed middleware.

Finally, all these efforts have served to the main goal of improving Java communications performance. In fact, the performance analysis of Java parallel benchmarks has shown that this goal has been accomplished. Additionally, not only Java communications benefit from this Thesis, but also the Java code of HPC applications is potentially improvable using the best programming practices for performance gathered during the development of these communication libraries.

Contributions

The main contributions of this Thesis are:

1 *An up-to-date performance evaluation of Java for HPC.* An up-to-date review of Java for HPC, which includes an extensive evaluation of the performance of current projects [61, 94], has been provided. It has been put a special emphasis on the analysis of the impact on performance of the use of clusters with multi-processor and multi-core nodes interconnected via high-speed networks (InfiniBand, Myrinet and SCI).

2 *A more accurate communication performance model.* A more precise model for the characterization of the performance of Java communications [94], together with a message-passing micro-benchmark suite to derive the models [87] have been proposed. The performance parameters for the model and some throughput metrics obtained on several high-speed clusters are presented in [94], showing better estimates than previous modeling techniques [88].

3 *Design and development of the high performance Java Fast Sockets (JFS).* A high performance Java IO socket library, named Java Fast Sockets (JFS), whose interoperability and transparency allow for immediate performance increases in high-speed networks and shared memory environments [89, 90, 92] has been designed and implemented. Thus, JFS:

- Enables efficient communication on clusters interconnected via high-speed networks (InfiniBand, Myrinet and SCI) through a general and easily portable solution.
- Avoids the need of primitive data type array serialization.
- Reduces buffering and unnecessary copies in the socket communication protocol.

- Provides an efficient protocol for shared memory (intra-node) communication which shows a high impact on multi-core systems performance.

JFS optimizes both user applications and communication middleware, such as RMI [86] and Java message-passing protocols [93].

4 *Design and development of the iodev low-level non-blocking communication library.* A low-level communication device which provides efficient non-blocking communication on Java IO sockets, allowing communication overlapping [93, 96] has been implemented. This library runs on top of a Java IO sockets implementation, and therefore can take advantage of JFS, if available. This library is oriented to performance-critical communication middleware.

5 *Design and development of an efficient Java RMI communication library.* A lightweight Java RMI protocol, whose main optimization is the reduction of its processing overhead and its support on high-speed networks has also been implemented.

6 *Design and development of an efficient Java message-passing library, Fast MPJ (F-MPJ).* F-MPJ [93], a scalable and efficient Java message-passing library on top of iodev has been implemented. F-MPJ especially benefits from the use of JFS as underlying layer of iodev, as this allows the avoidance of the serialization overhead of primitive data type arrays, data structures commonly used in HPC applications.

7 *Implementation of an efficient Java message-passing collective library.* F-MPJ implements several algorithms per collective primitive which allows, thanks to their selection at runtime, to significantly improve the performance of collective message-passing primitives. Thus, according to our experimental results, Java message-passing codes that without F-MPJ achieve their highest performance running on 8-16 cores, can take advantage of the use of up to 128 cores showing good scalability with F-MPJ.

8 *New benchmarks in Java for HPC.* Besides the message-passing micro-benchmark suite, a Java version of the NetPipe performance tool [102], and the MPI counterparts of Message-Passing in Java (MPJ) codes from the Java Grande Benchmark suite [84] have been implemented in order to compare

native and Java performance. Also, the NAS Parallel Benchmarks (NPB) suite for MPJ (named NPB-MPJ) [61] has been implemented in order to analyze MPJ libraries performance and to compare Java and native parallel libraries. Additionally, the Java optimization techniques for parallel programming used in the development of these benchmarks has been gathered, especially from the NPB-MPJ.

Overview of the Contents

The Thesis is organized into seven chapters whose contents are summarized in the following paragraphs.

Chapter 1, *Java Communications for High Performance Computing*, is intended to give the reader a clear description of the goals and the basic concepts behind the material presented in the Thesis. The chapter begins with a review of the Java communication libraries for High Performance Computing (HPC). These libraries allow the development of Java parallel applications, although their low performance has been the main obstacle to their being embraced in HPC. The related literature provides a detailed discussion on the reasons for the poor performance of the Java parallel applications, pointing out the inefficient communications as the main cause. Thus, the main motivation of this Thesis is the design of efficient Java communication libraries that increase the performance of Java parallel applications, contributing to the adoption of Java for HPC. This Thesis is designed to test the hypothesis that Java can be an interesting alternative for HPC, provided that an efficient communication middleware is made available, due to its appealing features of portability, inherent multithreading and networking support, and productive development.

Chapter 2, *Performance Analysis of Message-Passing Communications*, discusses current communication performance models in order to evaluate implementations of communication primitives. Due to the lack of an appropriate model for this evaluation purpose, a new model for the characterization of the communication overhead, based on the linear model but more precise, is proposed. Thus, Java and native communication performance models are derived from the experimental micro-benchmark results using this more accurate analytical model. The proposed model obtains better estimates than preceding ones and serves to identify inefficient communication primitives. Furthermore, a model-based performance optimization process is proposed, replacing an inefficient primitive by an efficient equivalent combination of primitives. This chapter finishes with an evaluation of Java communication libraries. For this purpose, representative benchmarks have been implemented in Java in order to overcome the lack of standard Java benchmarks for the characterization of the message-passing communication overhead. The impact on performance of the use of systems with multiple hyper-threaded processors has been especially analyzed.

Chapter 3, *JFS: High Performance Java Fast Sockets*, presents a high performance sockets implementation in Java. First, the design objectives of this library, which overcomes many limitations of current sockets libraries for HPC, are presented. Next, the JFS communication protocols are described, together with their implementation issues on high-speed cluster networks and shared memory systems. Experimental results that show the efficiency of JFS protocols are presented towards the end of the chapter. Finally, the impact of the use of JFS on the optimization of Java message-passing protocols is shown.

Chapter 4, *Efficient iodev Low-level Message-Passing and RMI Middleware*, presents iodev, a low-level message-passing communication library, and the design of an optimized Java RMI protocol for high-speed networks. Both solutions are implemented on top of the middleware developed in the previous chapter. The iodev communication device provides efficient non-blocking communications using Java IO sockets. A related project, MPJ Express, has implemented its low-level communication library, niodev, using Java NIO sockets. However, these latter sockets already provide non-blocking communication methods, whereas iodev has to implement the non-blocking support on top of Java IO sockets. The chapter covers the main implementation similarities and differences between both approaches. Additionally, the efficient coupling of iodev and JFS, and a comparative performance evaluation of low-level communication devices, both Java and native implementations, on several representative HPC scenarios is described. The analysis of the results states that the combination of iodev+JFS improves Java communications, especially when avoiding the serialization overhead. The chapter concludes with the description of the Java RMI protocol optimization, which also shows an important performance increase.

Chapter 5, *Fast MPJ: Efficient Java Message-Passing Library*, presents Fast MPJ (F-MPJ), a scalable and efficient Java message-passing library implemented on top of the middleware developed in the previous chapter. The F-MPJ development has been focused on the implementation of scalable collective primitives, based on the iodev point-to-point communication. The runtime selection of collective algorithms allows an important increase in communication performance, as shown in the micro-benchmarking of the collective primitives and in the evaluation of the impact of the use of F-MPJ+iodev+JFS on kernel/application benchmarks.

Chapter 6, *Implementation and Evaluation of Efficient MPJ Benchmarks*, presents the design, implementation and performance optimization of a suite of parallel benchmarks (the NAS Parallel Benchmarks), taken as representative codes for the development of Java message-passing applications. The development of this benchmark suite has been useful for gathering good programming practices for performance in Java for HPC. The impact of these practices is discussed in this chapter. Moreover, the chapter includes comprehensive benchmark results from the evaluation of the developed benchmarks on InfiniBand and Gigabit Ethernet multi-core clusters. Furthermore, an analysis of the impact of using different runtime configurations (number of nodes and number of processes per node) is included.

Finally, the *Conclusions* chapter summarizes the main contributions of the Thesis and outlines the main research lines that can be derived from the hypothesis proved, that Java can be a viable alternative for HPC.

Chapter 1

Java Communications for High Performance Computing

This chapter presents a review of Java communication libraries for High Performance Computing (HPC). The evaluated projects can be classified, from lower to higher level, in: Java sockets, Java Remote Method Invocation (RMI) protocols, Java message-passing implementations, and finally, other Java communication solutions. These libraries allow the development of higher level libraries and Java parallel applications. However, the analysis of the related literature and, in the most relevant cases, the direct evaluation of the projects have pointed out their usually low performance, which has been the main obstacle for their embrace in HPC. Thus, the main motivation of this Thesis is the design of efficient Java communication libraries that increase the performance of Java parallel applications, contributing to the adoption of Java for HPC.

The present review discusses each project, trying to identify the main causes of performance bottlenecks, in order to overcome them in the design and implementation of efficient communication middleware. Thus, the combination of efficient communication libraries and the appealing features of Java such as portability, inherent multithreading and networking support, and a better productivity in code development, turn Java into an interesting alternative for HPC.

Another alternative in Java for HPC is the use of Java threads or thread-based projects, among which the Java OpenMP implementations, such as JOMP [50] and

JaMP [55], are the most noticeable. These approaches are not covered in this Thesis as they are limited to shared memory systems, which provide less scalability than distributed memory machines. Moreover, they do not rely on communication middleware.

The structure of this chapter is as follows: Section 1.1 presents Java sockets implementations and high performance native sockets libraries. Section 1.2 describes the Java RMI optimization projects. The most relevant Java message-passing libraries for HPC are shown in Section 1.3. Finally, Section 1.4 covers additional Java communication libraries not included in the previous sections as they implement more specific APIs and middleware.

1.1. Java Sockets

Sockets are a low-level programming interface for networked communications, which allows sending streams of data between applications. The socket API is widely extended and can be considered the standard low-level communication layer as there are socket implementations on almost every network protocol. Thus, sockets have been the choice for implementing in Java the lowest level of networked communication.

Java sockets are, like Java, fully portable, but their operation is limited to the widely deployed TCP/IP protocol. However, although most clusters have high-speed networks, such as InfiniBand, Myrinet or SCI, to boost communication performance, Java can not take advantage of them as shown in [88] because it has to resort to inefficient TCP/IP emulations for full networking support. These emulation libraries present high start-up latency (the 0-byte message latency), low bandwidth and high CPU load as shown in [12]. The main reason behind this poor throughput is that the IP protocol was designed to cope with low speed, unreliable and prone to failure links in WAN environments, whereas current cluster networks are high-speed, hardware reliable and non-prone to failure in LAN environments. Examples of IP emulations are IPoMX and IPoGM [70] on top of the Myrinet low-level libraries MX (Myrinet eXpress) and GM, respectively, LANE driver [54] over Giganet, IP over InfiniBand (IPoIB) [44], and ScaIP [15] and SCIP [29] on SCI.

A direct implementation of native sockets on top of low-level communication libraries can avoid the TCP/IP overhead, and thus performance could be increased. Representative examples are next presented. FastSockets [79] is a socket implementation on top of Active Messages [31], a lightweight protocol with high-speed network access. SOVIA [54] has been implemented on VIA (Virtual Interface Architecture); and Sockets over Gigabit Ethernet [10] and GAMMAsockets [75] have been developed for Gigabit Ethernet. The Socket Direct Protocol (SDP) over InfiniBand [47] is the representative socket library of the Offload Sockets Framework (OSF). Sockets-MX and Sockets-GM [70] are the developments on Myrinet, where MX is intended to supersede GM thanks to a more efficient protocol implementation. The high performance native sockets library on SCI is SCI Sockets [80]. However, from these implementations only SDP, Sockets-MX/GM and SCI Sockets are currently available. The Windows Sockets Direct components for Windows platforms provide access to certain high-speed networks. A related project is XenSocket [110], an optimized socket library restricted to Xen virtual machine intra-node communication that replaces TCP/IP by shared memory transfers.

However, the previous socket libraries usually implement a subset of socket functionality on top of low-level libraries, resorting to the system socket library for unimplemented functions. Thus, some applications such as kernel-level network services and Java codes can request features not present in the underlying libraries and thus failover to system sockets, which provides poorer performance.

A pioneer project in obtaining efficient Java sockets is NBIO [106], which provides non-blocking communications in order to increase scalability in server applications. This library has led to the introduction of significant non-blocking features in Java NIO (New I/O) sockets. Nevertheless, NBIO does not provide high-speed network support nor HPC tailoring. Ibis sockets partly solve these issues adding Myrinet support and being the base of Ibis [73], a parallel and distributed Java computing framework. However, their implementation on top of the Java Virtual Machine (JVM) sockets limits the performance increase to serialization improvements.

This Thesis presents the design and development of the high performance Java IO sockets implementation named Java Fast Sockets (JFS) (see Chapter 3). This library significantly reduces communication overhead and provides efficient shared memory and high-speed networks support. Thus, JFS is tailored to HPC, especially to multi-core clusters with high-speed networks. Moreover, by optimizing the widely used socket API, parallel and distributed Java applications based on it can improve performance transparently.

1.2. Java Remote Method Invocation (RMI)

The Java Remote Method Invocation (RMI) protocol allows an object running in one JVM to invoke methods on an object running in another JVM, providing Java with remote communication between programs equivalent to Remote Procedure Calls (RPCs). The main advantage of this approach is its simplicity, although the main drawback is the poor performance shown by the RMI protocol.

ProActive [5, 77] is an RMI-based middleware for parallel, multithreaded and distributed computing focused on Grid applications. ProActive is a fully portable "pure" Java (100% Java) middleware whose programming model is based on a Meta-Object protocol. With a reduced set of simple primitives, this middleware simplifies the programming of Grid computing applications: distributed on Local Area Network (LAN), on clusters of workstations, or for the Grid. Moreover, ProActive supports fault-tolerance, load-balancing, mobility, and security. Nevertheless, the use of RMI as its default transport layer adds significant overhead to this middleware operation.

Different frameworks have been implemented with the efficiency of RMI communication on clusters as their goal. The most relevant ones are KaRMI [76], RMIX [56], Manta [60] and Ibis RMI, part of Ibis [73]. KaRMI is a drop-in replacement for the Java RMI framework that uses a completely different protocol and introduces new abstractions (such as "export points") to improve communications specifically for cluster environments. However, KaRMI suffers from performance losses when dealing with large data sets and its interoperability is limited to the cluster nodes. RMIX extends Java RMI functionality to cover a wide range of communication protocols, but the performance on high-speed clusters is not satisfactory. The Manta project is a different approach for implementing RMI, based on Java to native code compilation. This approach allows for better optimization, avoids data serialization and class information processing at runtime, and uses a lightweight communication protocol. Serialization is the process of transforming objects in byte series, in this case to be sent across the network. Finally, Ibis RMI extends Java RMI to make it more suitable for grid computing. Looking for performance, Ibis supports the Myrinet high-speed network and avoids the runtime type inspection. However, the use of specific high-level solutions with substantial protocol overhead and focused on Myrinet has restricted the applicability of these projects. In fact, their start-up latency is from several times up to an order of magnitude larger than socket latencies. Therefore, although previous Java communication middleware (e.g., message-passing libraries) was usually based on RMI, current Java communication libraries use sockets due to their lower overhead. In this case, the higher programming effort required by the lower-level API allows for higher throughput, key in HPC.

One of the objectives of this Thesis is to provide Java with a high performance RMI implementation with high-speed networks support (see Section 4.4). This can be done by optimizing the Java RMI protocol for cluster communications under some basic assumptions for the target architecture, and using a high performance sockets library that copes with the requirements of an RMI protocol for parallel computing on high-speed clusters. As Java RMI is a widely spread API, many Java parallel applications and communication libraries can benefit from this efficient Java RMI implementation. Moreover, the goal is to optimize this protocol with the minimum associated trade-offs. Thus, the solution is transparent to the user, it does not modify the source code, and it is interoperable with other systems. The trade-off is that this protocol is limited to clusters with a homogeneous configuration in terms of JVM and architecture, although most clusters are under these conditions.

1.3. Java Message-Passing Libraries

Message-passing is the most widely used parallel programming paradigm as it is highly portable, scalable and usually provides good performance. It is the preferred choice for parallel programming distributed memory systems such as clusters, which can provide higher computational power than shared memory systems. Regarding the languages compiled to native code (e.g., C, Fortran), MPI [64] is the standard interface for message-passing libraries.

Soon after the introduction of Java, there have been several implementations of Java message-passing libraries. Most of them have developed their own MPI-like binding for the Java language. The two main proposed APIs are the mpiJava 1.2 API [21], which tries to adhere to the MPI C++ interface defined in the MPI standard version 2.0, but restricted to the support of the MPI 1.1 subset, and the JGF MPJ (Message-Passing interface for Java) API [22], which is the proposal of the Message-Passing Working Group within the Java Grande Forum (JGF) [48] to standardize the MPI-like Java API. The main differences among these two APIs lie in naming conventions of variables and methods. For purposes of clarity, henceforth the term "MPJ" will denote implementations of Message-Passing in Java libraries, independently of the API they implement (mpiJava 1.2 or the JGF MPJ API). In order to avoid confusion, the Message-Passing interface for Java API proposed by the JGF will be always denoted as "JGF MPJ" API.

The Java message-passing libraries have followed different implementation approaches: (1) using Java RMI, (2) wrapping an underlying native messaging library like MPI through Java Native Interface (JNI), or (3) using low-level Java sockets. Each solution fits with specific situations, but presents associated trade-offs. The use of Java RMI, a "pure" Java (100% Java) approach, as base for MPJ libraries, ensures portability, but it might not be the most efficient solution, especially in the presence of high speed communication hardware. The use of JNI has portability problems, although usually in exchange for higher performance. The use of a low-level API, Java sockets, requires an important programming effort, especially in order to provide scalable solutions, but it significantly outperforms RMI-based communication libraries. Although most of the Java communication middleware is based on RMI, MPJ libraries looking for efficient communication have followed the latter two approaches.

The mpiJava library [7] consists of a collection of wrapper classes that call a native MPI implementation (e.g., MPICH or OpenMPI) through JNI. This wrapper-based approach provides efficient communication relying on native libraries, adding a reduced JNI overhead. However, although its performance is usually high, mpiJava currently only supports some native MPI implementations, as wrapping a wide number of functions and heterogeneous runtime environments entails an important maintaining effort. Additionally, this implementation presents instability problems, derived from the native code wrapping, and it is not thread-safe, being unable to take advantage of multi-core systems through multithreading.

As a result of these drawbacks, the mpiJava maintenance has been superseded by the development of MPJ Express [9, 81, 69], a "pure" Java message-passing implementation of the mpiJava 1.2 API specification. MPJ Express is thread-safe and presents a modular design which includes a pluggable architecture of communication devices that allows to combine the portability of the "pure" Java New I/O package (Java NIO) communications (niodev device) with the high performance Myrinet support (through the native Myrinet eXpress –MX– communication library in mxdev device).

MPJ/Ibis [16] is another MPJ library, part of the Ibis framework [73], which also includes the Ibis sockets library (see Section 1.1) and the Ibis RMI implementation (see Section 1.2). Thus, like Ibis sockets and Ibis RMI, MPJ/Ibis can use either "pure" Java communications, or native communications on Myrinet. Moreover, there are two low-level communication devices available in Ibis for MPJ/Ibis communications: TCPIbis, based on Java IO sockets (TCP), and NIOIbis, which provides blocking and non-blocking communication through Java NIO sockets. Nevertheless, MPJ/Ibis is not thread-safe, does not take advantage of non-blocking communication, and its Myrinet support is based on the GM library, which results in poorer performance than the MX library.

Currently, MPJ Express and MPJ/Ibis are the most active projects in terms of uptake by the HPC community, presence in academia and production environments, and available documentation. These projects are also stable and publicly available along with their source code.

Additionally, there have been several implementations of Java messaging libraries for HPC [88]. Although most of them raised many expectations in the past, currently they are out-of-date and their interest is quite limited. The most relevant ones (MPI-like) follow:

- JavaMPI [65], an MPI Java wrapper created with the help of JCI, a tool for generating Java-to-C interfaces. The last version was released in January 2000.
- JavaWMPI [62] is a Java wrapper version built on WMPI, a Windows-based

implementation of MPI.

- the commercial JMPI project [25] by MPI Software Technology (not to be confused with [67]) was the first project (1999) that intended to build a pure Java version of MPI specialized for commercial applications.

- MPIJ is a pure Java MPI subset developed as part of the DOGMA project (Distributed Object Group Metacomputing Architecture) [49]. MPIJ has been removed from DOGMA since release 2.0.

- JMPI [67], a pure Java implementation of the mpiJava 1.2 API developed for academic purposes at the University of Massachusetts.

- M-JavaMPI [59] is another wrapper approach with process migration support that runs on top of the standard JVM. Unlike mpiJava and JavaMPI, it does not use direct binding of Java programs and MPI. M-JavaMPI follows a client-server message redirection model that makes the system more portable, that is, independent of the MPI implementation.

- CCJ [72], a pure Java communication library with its own MPI-like API, similar to the JGF MPJ specification. It makes use of Java capabilities such as a thread-based programming model or sending of objects.

- PJMPI [63] is a pure Java message-passing implementation strongly compatible with the MPI standard and developed at the University of Adelaide in conjunction with a non-MPI message-passing environment called JUMP.

- jmpi [28] is another pure Java implementation of MPI built on top of JPVM (see at the end of this section). The project has been left idle since 1999.

- MPJava [78] is the first Java message-passing library implemented on Java NIO sockets, taking advantage of their scalability and high performance communications. It uses its own MPI-like API.

- Jcluster [109] is a message-passing library which provides both PVM-like and MPI-like APIs and is focused on automatic task load balance across large-scale heterogeneous clusters. However, its communications are based on UDP and it lacks high-speed networks support.

- Parallel Java (PJ) [51] is a "pure" Java parallel programming middleware that supports both an OpenMP-like shared memory programming (based on threads and classes from the `java.util.concurrent` package) and an MPI-like message-passing paradigm, allowing applications to take advantage of hybrid shared/distributed memory architectures. However, the use of its own API hinders its adoption.

- P2P-MPI [36] is a peer-to-peer framework for the execution of MPJ applications on the Grid. Among its features are: (1) self-configuration of peers (through JXTA peer-to-peer interconnection technology); (2) fault-tolerance, based on process replication; (3) a data management protocol for file transfers on the Grid; and (4) an MPJ implementation that can use either Java NIO or Java IO sockets for communications, although it lacks high-speed networks support. In fact, this project is tailored to grid computing systems, disregarding the performance aspects.

- JMPI (by Bang & Ahn) [11] (not to be confused with [67]) is an implementation of the JGF MPJ API which can use either Java RMI or Java sockets for communications. However, the reported performance is quite low (it only scales up to two nodes).

Far less research has been devoted to PVM-like libraries. The most representative projects were JavaPVM (renamed as jPVM [98]), a Java wrapper to PVM (last released in 1998), and JPVM [33], a pure Java implementation of PVM (last released in 1999). Performance issues of both libraries were studied in [108].

This important number of past and present projects is the result of the sustained interest in the use of Java for parallel computing. One of the main objectives of this Thesis is the design and implementation of an efficient Java message-passing library (see Chapter 5). This library, named Fast MPJ (F-MPJ), takes advantage of JFS in order to provide shared memory and high-speed networks support.

Tables 1.1 and 1.2 serve as a summary of the Java message-passing projects discussed in this chapter.

Table 1.1: Overview of Java message-passing projects

Project	**Activity**	**Observations**
jPVM	1996-1998	Java wrapper PVM implementation
JPVM	1996-1999	Pure Java PVM implementation
JavaMPI	1996-2000	Java wrapper MPI implementation
JavaWMPI	1998	Java wrapper MPI implementation on Windows
JMPI (commercial)	1998-1999	Pure Java MPI implementation
MPIJ	1999-2001	Pure Java MPI implementation
JMPI	2000-2002	Pure Java MPI implementation
M-JavaMPI	2002	Pure Java MPI implementation
CCJ	2001-2003	Pure Java MPI-like implementation
PJMPI	2000	Pure Java MPI implementation
jmpi	1998-1999	Pure Java MPI implementation on top of JPVM
MPJava	2003-2004	Pure Java MPI library
Jcluster	2002-	Pure Java MPI and PVM implementations
Parallel Java	2005-	Pure Java hybrid shared memory/MPI library
mpiJava	1998-	Java wrapper MPI implementation
P2P-MPI	2005-	Pure Java MPI implementation
MPJ Express	2005-	Pure Java MPI implementation (+MX support)
MPJ/Ibis	2005-	Pure Java MPI implementation (+GM support)
JMPI (Bang&Ahn)	2007	Pure Java MPI implementation
F-MPJ	2008-	Pure Java MPI implementation (+JFS support)

Table 1.2: Characteristics of Java message-passing projects

Project	**Pure Java**	**High-speed network support**			**API**			**URL**
		Myrinet	**InfiniBand**	**SCI**	**mpiJava 1.2**	**JGF MPJ**	**other APIs**	
jPVM							✓	
JPVM	✓						✓	http://www.cs.virginia.edu/˜ajf2j/jpvm.html
JavaMPI							✓	
JavaWMPI							✓	
JMPI (commercial)	✓						✓	
MPIJ	✓						✓	
JMPI	✓						✓	http://euler.ecs.umass.edu/jmpi/
M-JavaMPI	✓						✓	
CCJ	✓	✓					✓	http://www.cs.vu.nl/manta/ccj.html
PJMPI	✓						✓	
jmpi	✓						✓	
MPJava	✓						✓	
Jcluster	✓						✓	http://vip.6to23.com/jcluster/
Parallel Java	✓						✓	http://www.cs.rit.edu/˜ark/pj.shtml
mpiJava		✓	✓	✓	✓			http://www.hpjava.org/mpiJava.html
P2P-MPI	✓				✓			http://www.p2pmpi.org
MPJ Express	✓	✓			✓			http://mpj-express.org
MPJ/Ibis	✓	✓				✓		http://www.cs.vu.nl/ibis/mpj.html
JMPI (Bang&Ahn)	✓						✓	
F-MPJ	✓	✓	✓	✓	✓			http://jfs.des.udc.es

1.4. Additional Java Communication Libraries

Apart from Java sockets, RMI and message-passing implementations, several additional projects, with more specific APIs, have been developed. These projects have been focused on providing Java with full and more efficient support on high-speed networks. Thus, several approaches have been followed: (1) VIA-based projects, (2) Java Distributed Shared Memory (DSM) middleware on clusters, and (3) low-level libraries on high-speed networks.

Javia [24] and Jaguar [107] provide access to high-speed cluster interconnects through VIA, communication library implemented on Giganet, Myrinet, Gigabit Ethernet and SCI [38], among others. More specifically, Javia reduces data copying using native buffers, and Jaguar acts as a replacement of the JNI. Their main drawbacks are the use of particular APIs, the need of modified Java compilers and the lack of non-VIA communication support. Additionally Javia exposes programmers to buffer management and uses a specific garbage collector.

Java DSM projects worth mentioning are CoJVM [58], JESSICA2 [111] and JavaSplit [32]. As these are socket-based projects, they benefit from socket optimizations, especially in shared memory communication [52]. However, they share unsuitable characteristics such as the use of modified JVMs, the need of source code modification and limited interoperability.

Other approaches are low-level Java libraries restricted to specific networks. For instance, Jdib [42] accesses InfiniBand through the Mellanox Verbs Interface (VAPI) or the OpenFabrics Alliance InfiniBand Verbs (IBV), which provide a low-level API which directly exploits RDMA and communication queues. Thus, this library achieves almost native performance on InfiniBand. Moreover, CORBA, an RPC mechanism supported in Java and quite similar to RMI, has also been optimized on high-speed networks [39].

Chapter 2

Performance Analysis of Message-Passing Communications

In this chapter, Java and native message-passing libraries are analyzed on high-speed clusters in order to estimate overheads. The goal of this task is to evaluate the current state of Java for HPC, particularly for Myrinet and SCI clusters, and compare its performance with native libraries results. A second objective is to identify inefficient primitive implementations. Thus, this analysis can guide developers to improve the performance of their parallel applications. Additionally, a proposal of an accurate analytical model for high-speed cluster communications as well as a micro-benchmark suite [87] are made available to parallel programmers. These tools provide a useful way to quantify the influence of the message-passing libraries and system configuration on the overall application performance. This influence has been validated through a kernel benchmarking. The obtained analytical performance models are also useful for optimizing message-passing performance. Thus, communication overhead can be reduced through replacing inefficient communication primitives by more efficient equivalent combinations of primitives.

The chapter is organized as follows: the next section introduces existing message-passing performance models and analyzes their suitability for evaluation purposes. As the accuracy and simplicity of these models have not been as expected, a new linear model is proposed, focused on obtaining higher accuracy on high-speed clusters. Section 2.2 presents the formulation of this model, some performance metrics derived from it, the micro-benchmarking process and a preliminary accuracy analysis. Section 2.3 presents experimental results: the communication performance of two clusters with representative high-speed interconnects (SCI and Myrinet) has been modeled and analyzed. A further discussion on the experimental results and performance estimate is the focus of Section 2.4, together with a proposal of a model-based performance optimization. Section 2.5 presents an analysis of the influence of message-passing overhead on applications through a kernel benchmarking. Then, Section 2.6 evaluates the influence of enabling the Simultaneous MultiThreading (SMT), also known as hyper-threading on some Intel processors, on the overall performance of a cluster with dual-processor nodes. This analysis has been performed upon the results of the kernel benchmarking of Section 2.5. Finally, Section 2.7 concludes the chapter with a summary of its main contributions.

2.1. Message-Passing Performance Models

The appropriateness of existing communication models has been evaluated in terms of their simplicity and accuracy for high-speed clusters. Models discussed in this chapter can be classified into LogP-based and linear-based models.

The LogP model [26] characterizes communications by four parameters: network communication time L, overhead o, gap g and number of processors P. Some LogP variants have been proposed to support additional characteristics by adding parameters to the model. Thus, LogGP [1] introduces G, gap per byte, to support long messages; LoPC [35] and LoGPC [68] add C to model resource contention; LogGPS [46] incorporates synchronization costs by adding S; and LogPQ [101] introduces Q, referring to communication queues. Additional models are memory logP [20] which applies and augments the original LogP model to estimate overheads in a hierarchical memory subsystem; parametrized LogP (P-LogP) [53], that presents a gap $g(m)$ that depends on the message size m; log$_n$P [19], where n is the number of layers with different communication overheads taken into account, e.g., log$_3$P can address the communication cost as a sum of middleware, memory and interconnection network overheads; and HLogP [105], that is targeted to model Grid systems.

Regarding the appropriateness of these models, LogP is too basic to perform a thorough analysis. This model assumes single processor nodes and small messages, determining that it is only effective when L dominates the overall communication overhead. In this case, the influence of message size and the memory access performance on communication overhead is negligible. The need to include these parameters has led models to include G, gap per byte, or the data size. However, this is effective only in tightly synchronized communication patterns. In fact, the contention C for message-processing resources is a significant factor in the total application runtime for many fine-grain message-passing algorithms, particularly on clusters. Nevertheless, LogP with additional G and/or C parameters usually omits significant costs, such as the influence of the memory gap on performance. Memory logP models this influence, although only for shared memory architectures. The model $\log_n$P extends memory logP (in fact, memory logP is $\log_1$P) taking into account the number of communication steps. Thus, $\log_3$P would describe communications on high-speed clusters: (1) communication memory/Network Interface Card (NIC), (2) communication NIC/NIC, and (3) communication NIC/memory. Experimental results from characterizing communication overhead using these models on high-speed clusters report average absolute relative errors of 28% for LogGP predictions, and of 5% for $\log_3$P [19]. Nevertheless, these accurate results are limited to regular access patterns.

Linear models are also a popular method to characterize message-passing overhead. These models are usually based on Hockney's model for point-to-point communications and on Xu and Wang's model for collective primitives [100]. Thus, message latency (T) of point-to-point communications is modeled as an affine function of the message length n: $T(n) = t_0 + t_b n$, where t_0 is the start-up time (the time taken for the minimum message size, usually a zero length message) and t_b is the transfer time per byte. Communication bandwidth is easily derived as $Bw(n) = n/T(n)$. A generalization of the point-to-point model is used to characterize collective communications: $T(n, p) = t_0(p) + t_b(p)n$, where p is the number of processors involved in the communication. This characterization of message-passing overhead is relatively easy to develop and usually provides good predictions, but its simplicity is thought to be a restricting factor to its accuracy.

The lack of accuracy of linear models on high-speed clusters affects both t_0 and

t_b parameters. The combination into a unique parameter t_0 of the overhead (o) and network communication time (L) differentiated in the LogP model is considered to be only appropriate for long messages, not giving enough detail for short messages [1]. Moreover, as linear models usually assume t_b constant, the accuracy of the models turned out to be much better on Ethernet-based than on high-speed clusters, where different high performance communication protocols, with different t_b, are used depending on the message size. A previous work on modeling communication performance on high-speed clusters [88] has shown the limitations of the Hockney's model to predict performance accurately. In fact, Hockney's model on Fast Ethernet predicts performance with average absolute relative errors of 13% for Send and 21% for collective communications. Hockney's model on SCI presented average absolute relative errors of 18% and 28%, respectively.

Once the linear model turned out to be unsuitable due to the dearth of accuracy, the log$_n$P model was selected as the most suitable choice among LogP-based models. Nevertheless, apart from its lack of direct collective primitive support, it exhibits a certain complexity in its formulation. Although the possibility of simplification by ignoring some parameters exists, sometimes this is not an advantageous choice. In fact, while too many parameters keep non-experts from drawing conclusions about performance, too few parameters do not provide enough information.

2.2. A New Model for Message-Passing Overhead

This Thesis aims at using a model realistic enough to characterize more accurately communication overhead despite the complexity of current communication middleware, but simple enough for programmers to design and analyze parallel algorithms overhead. As existing models do not fit completely this purpose, a new linear model is proposed to address the main challenges posed by the modeling of high-speed cluster communications. This model takes into account the influence of different protocols involved in the communication process. This is done by augmenting the linear model described in [100] with a new parameter, t_i, which is the intercept from the linear regression of $T(n) - t_0$ versus n. In high-speed clusters, t_0 is quite small and t_i is usually higher. According to the previous considerations, message latency (T) of point-to-point communications on high-speed clusters should be modeled as $T(n) = t_0 + t_i + t_b n$. Nevertheless, this tentative model predicts inaccurately T for short messages (e.g., $T(0) = t_0$ and the model predicts $T(0) = t_0 + t_i$). In order to solve this issue, t_i must be weighted by the ratio of transfer time ($t_b n$) to the latency predicted by Hockney's model ($t_0 + t_b n$). Thus, point-to-point communications are modeled as:

$$T(n) = t_0 + t_i(\frac{t_b n}{t_0 + t_b n}) + t_b n$$

and collective communications are modeled generalizing the point-to-point model:

$$T(n,p) = t_0(p) + t_i(p)(\frac{t_b(p)n}{t_0(p) + t_b(p)n}) + t_b(p)n$$

Regarding point-to-point communications, this new model predicts accurately $T(0) = t_0$, and shows higher accuracy than Hockney's model, especially for medium-size messages. In fact, the highest relative difference between this model and Hockney's model occurs at a t_0/t_b-byte message. This maximum relative difference has been obtained by setting the derivative of $(T_{proposed}(n) - T_{Hockney}(n))/T_{Hockney}(n)$ equals to zero and solving for n. This value, t_0/t_b, varies on high-speed clusters from one KB to tens of KB, in the range of medium-size messages. In fact, Hockney's model usually underestimates latency of medium-size messages on high-speed clusters. The reason for this is that message-passing libraries use different communication protocols for short and long messages. Long message protocols usually show lower t_b than short message protocols, focused on lower t_0. As t_b is obtained from a linear regression of T vs. n in which the long message performance dominates, its value is quite similar to the t_b of long message protocols. Thus, using the obtained t_b, short message latency is underestimated. In order to illustrate this scenario, an example is provided: an MPI C primitive on an SCI cluster presents $t_0 = 4\mu s$, $t_i = 13\mu s$ and $t_b = 3.89 ns/byte$ (see ScaMPI Send in Table 2.2). The estimates of the models are $T_{Hockney}(4KB) = 20\mu s$ and $T_{proposed}(4KB) = 30\mu s$. As $T_{measured}(4KB) = 33\mu s$ the proposed model estimates performance more accurately.

The addition to Hockney's model of a new explanatory variable (t_i) has shown that increasing slightly the complexity of the model, higher accuracy can be obtained, specially for medium-size messages. A different alternative would be defining a function in pieces for each communication protocol. Nevertheless, this approach requires knowledge about protocol boundaries.

The lack of a suitable micro-benchmark suite for evaluating message-passing communication overhead, both in C and Java, has led to the implementation of a suite [87], a set of tests for both C and Java codes adapted to the modeling needs, with the following features, not found in any other suite: (1) it measures the overhead of each message-passing operation, instead of obtaining the mean time of several iterations; (2) before starting the actual benchmarking, 10,000 warm-up iterations are run in Java in order to get JVM results with the Just-In-Time (JIT) compiler enabled; (3) the C and Java codes implement the same benchmark algorithm for comparison purposes; (4) it measures only the communication overhead in Java, not including the serialization overhead when sending data other than byte arrays; and (5) it includes all message-passing collectives, as usually some of them are missing (such as Scan, Reduce_scatter and Allreduce). Regarding point-to-point primitives, a ping-pong test takes 150 measurements of the runtime varying the message size in powers of four from 0 bytes. The minimum value has been chosen as test time to avoid distortions due to timing outliers. Similar tests were applied to collective primitives, but also varying the number of processors (from two up to the number of available processors in the testbed). The parameter $t_0(p)$ was derived from a linear regression of start-up times vs. p. The parameters $t_i(p)$ and $t_b(p)$ were derived from a regression of $T(n, p) - t_0(p)$ vs. n and p. A Barrier was included to avoid a pipelined effect and to prevent the network contention that might appear by the overlap of collective communications executed on different iterations of the test. Double precision addition was the operation used in reduction primitives (Reduce, Allreduce, Reduce_scatter and Scan).

In order to test the accuracy of the proposed model the average absolute relative error of 20 random messages has been calculated for each primitive on SCI. The results, a 7% error for Send and up to 7% for collective operations, depending on the primitive and the number of processors, are much better than the 18% and 28% error for Hockney's model for Send and collective primitives, respectively.

Figure 2.1 illustrates, through bandwidth graphs, the better fitting of the bandwidth experimentally measured by the proposed model compared to Hockney's model. Graph (a) shows Send bandwidth on Myrinet, and Graph (b) Broadcast bandwidth on SCI. The complete details of the experimental results and models are presented in Section 2.3. It can be seen that the estimates improve especially on the native message-passing library (MPI C), as there are larger differences among native communication protocols than among Java communication protocols for Java message-passing (MPJ). Moreover, the proposed model is quite accurate for a t_0/t_b-byte message (t_0/t_b is 1KB for MPI C and 11KB for MPJ), estimating significantly better than Hockney's model.

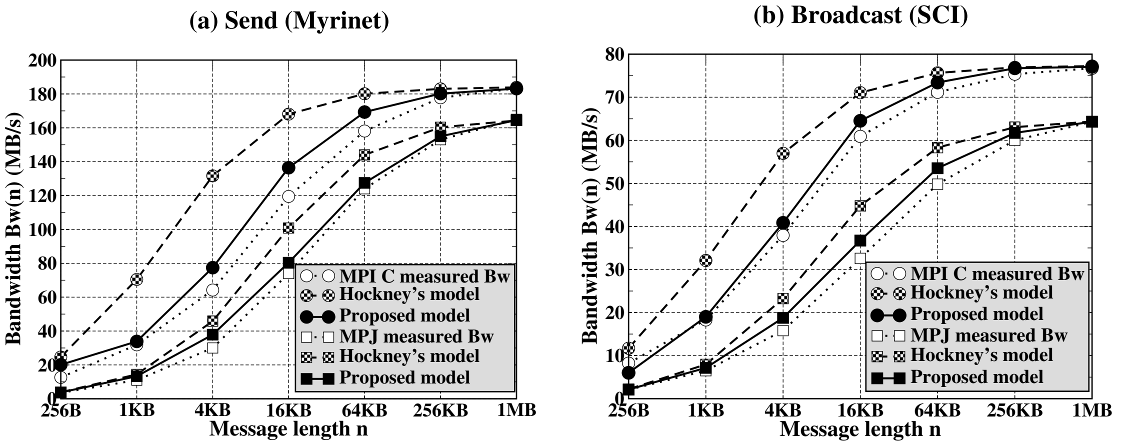


Figure 2.1: Comparison of Hockney's model vs. proposed model

Two metrics are derived from the model: the asymptotic bandwidth $Bw_{as}(p) = 1/t_b(p)$, the maximum throughput achievable when $n \to \infty$, and the specific performance $\pi_0(p) = 1/t_0(p)$. Bw_{as} shows long message performance, whereas π_0 characterizes short message bandwidth. Another metric is the aggregated asymptotic bandwidth $Bw_{as}^{ag}(p) = f(p)Bw_{as}(p)$, defined as the ratio of the total number of bytes transferred in the collective operation to the time required to perform the operation, as $n \to \infty$. The function $f(p)$ is the relationship between the total number of bytes transferred in the collective primitive and the message length. $f(p)$ depends on the communication pattern of each primitive: e.g., a Broadcast of n bytes to p processors implemented with a binomial tree sends $p-1$ messages of n bytes. Thus, $f(p) = p - 1$ for Broadcast, Alltoall, Reduce and Scan; $f(p) = (p-1)/p$ for Scatter and Gather; $f(p) = 2(p-1)$ for Allreduce; and $f(p) = (p^2-1)/p$ for Allgather and Reduce_scatter. Similarly, the aggregated specific performance is defined as $\pi_0^{ag}(p) = f(p)\pi_0(p)$ to show the performance of a collective operation for short messages. All these metrics for collective primitives are functions depending on p. In order to have numbers rather than functions to straightforwardly compare the performance of the different message-passing libraries, peak metrics have also been used in our experimental results (see Tables 2.1–2.4): the peak aggregated bandwidth $Bw_{as}^{pag} = \max_{2 \leq p \leq p^{max}} Bw_{as}^{ag}(p)$, and the peak aggregated specific performance $\pi_0^{pag} = \max_{2 \leq p \leq p^{max}} \pi_0^{ag}(p)$, being p^{max} the maximum p available in the testbed. For point-to-point communications $Bw_{as}^{pag} = 1/t_b$ and $\pi_0^{pag} = 1/t_0$.

2.3. Message-Passing Micro-benchmarking

2.3.1. Cluster Hardware/Software Configuration

The performance analytical models have been obtained from two high-speed clusters. The first cluster consists of 16 single-processor nodes (Pentium III at 1 GHz and 512 MB of memory) interconnected via Myrinet 2000 cards plugged into 64-bit 33 MHz PCI slots. The OS is Linux Red Hat 7.1, kernel 2.4, C compiler gcc 2.96, and Java Virtual Machines (JVM) Sun 1.4.2 and 1.5.0. The second cluster consists of 8 dual-processor nodes (Pentium IV Xeon with hyper-threading at 1.8 GHz and 1 GB of memory) interconnected via D334 SCI cards plugged into 64-bit 66 MHz PCI slots in a 2-D torus topology. The OS is Red Hat 7.3, kernel 2.4, C compiler gcc 3.2.2, and JVM Sun 1.5.0. Three different hardware configurations have been used for the SCI cluster: SCI-single, running one message-passing process on each node; SCI-dual, running two message-passing processes on each node; and SCI-dual SMT (with Simultaneous MultiThreading –hyper-threading– enabled), running four message-passing processes on each node. The hyper-threading allows one processor to operate as two processors internally, with a potential increase in performance claimed to be of about 30%, according to the manufacturer, Intel. Thus, a dual node with hyper-threading enabled has 4 “virtual” processors. The other two configurations, SCI-single and SCI-dual, have hyper-threading disabled.

Two MPI C libraries have been analyzed on the SCI cluster: ScaMPI (version 1.13.8), and SCI-MPICH (version 1.2.1), an MPICH implementation for SCI. Although both MPI implementations show similar performance on SCI-single, ScaMPI clearly outperforms SCI-MPICH on SCI-dual and especially on SCI-dual SMT. Therefore, for clarity purposes, SCI-MPICH results on SCI-dual SMT are not shown. MPICH-GM (version 1.2.4..8), a port of MPICH on top of GM (a low-level message-passing system for Myrinet) was selected for the Myrinet cluster.

Three representative Java message-passing libraries (see Section 1.3) have been selected: mpiJava [7] (version 1.2.5), used with Sun JVM 1.4.2 as mpiJava obtains the best performance with this JVM, MPJ/Ibis [16] (version 1.4) and MPJ Express [81] (version 0.26), these latter two libraries used with Sun JVM 1.5.0 as they require a JDK 1.5 or higher. The mpiJava library consists of a collection of wrapper classes that call a native MPI implementation through Java Native Interface (JNI). On Myrinet, mpiJava calls MPICH-GM, whereas on SCI, it calls ScaMPI. This wrapper-based approach provides efficient communication relying on native libraries, adding just a small JNI overhead. Nevertheless, its major drawback is the lack of portability, caused by the need of a native MPI implementation. This problem is overcome with the use of "pure" Java message-passing libraries that implement the whole messaging system in Java. Nevertheless, these libraries are less efficient than their native counterparts. MPJ/Ibis is an MPI-like "pure" Java message-passing implementation integrated in the Ibis framework [73]. It is implemented on top of TCPIbis sockets (based on Java IO sockets). MPJ Express is another MPI-like "pure" Java message-passing implementation based on Java NIO sockets. It implements higher level MPI features than MPJ/Ibis, like derived datatypes, virtual topologies and inter-communicators. It also includes a runtime execution environment. Despite these differences, in terms of performance both "pure" Java libraries behave similarly. Nevertheless, for conciseness, only one "pure" Java message-passing library has been modeled, MPJ/Ibis. This implementation has been selected as the representative library for showing slightly better performance than MPJ Express, both for the micro-benchmarking (results of MPJ Express not shown for clarity purposes) and the kernel benchmarking (see Section 2.5).

2.3.2. Analytical Models and Metrics

Table 2.1 presents the parameters of the latency models ($t_0(p)$, $t_i(p)$ and $t_b(p)$) for the standard Send and for collective communications on the Myrinet cluster. Two peak metrics derived from the models (π_0^{pag} and Bw_{as}^{pag}, see Section 2.2) are also provided in order to show short and long message performance, respectively, as well as to compare among libraries for each primitive. Regarding these two metrics, the higher, the better. Tables 2.2, 2.3 and 2.4 present the same results for the different SCI configurations: SCI-single, SCI-dual and SCI-dual SMT, respectively. These models are valid for communications from two nodes up to the total number of processors of the cluster. Thus, the models are valid for $2 \leq p \leq 16$ on Myrinet, for $2 \leq p \leq 8$ on SCI-single, for $4 \leq p \leq 16$ on SCI-dual, and for $8 \leq p \leq 32$ on SCI-dual SMT. Both $t_0(p)$ and $t_i(p)$ usually present $O(p)$ complexities. However, transfer times, $t_b(p)$, show $O(\log_2 p)$ complexity in almost all collective communications, which reveals a binomial tree-structured implementation of the primitives. Nevertheless, inefficient communication patterns have been detected on ScaMPI and MPJ/Ibis Scan (they are $O(p)$). Other implementations, e.g., MPJ/Ibis Allreduce, performs badly. In this particular case a Reduce followed by a Broadcast performs better than the equivalent Allreduce. This statement can be obtained from the values of $t_0(p)$ and $t_b(p)$ from the tables (e.g., $t_{b\_Allreduce}(p) > t_{b\_Reduce}(p) + t_{b\_Broadcast}(p)$).

Native Communication Libraries

As can be observed from Tables 2.1–2.4, native primitives on the SCI cluster show, in general, lower start-ups and transfer times per byte than on the Myrinet cluster. These differences can be attributed to: (1) the lower theoretical start-up of the network: $1.46\mu s$ for SCI and $7\mu s$ for Myrinet, (2) the higher theoretical bandwidth of the PCI bus, 528 MB/s on the SCI cluster and 264 MB/s on the Myrinet cluster, and (3) the higher computational power of the nodes, dual Pentium IV Xeon at 1.8 GHz on the SCI cluster and Pentium III at 1 GHz on the Myrinet cluster.

Regarding the performance metrics Bw_{as}^{pag} and π_0^{pag}, it can be seen that ScaMPI outperforms SCI-MPICH, except for Reduce_scatter and Scan. Generally, these metrics present the highest values (best performance) on SCI-dual, although communication primitives with more complex communication patterns, such as Alltoall, present the highest values on SCI-single.

Table 2.1: Myrinet: analytical models and peak aggregated metrics ($lp = \log_2 p$)

Primitive	Library	$t_0(p)$	$t_i(p)$	$t_b(p)$	π_0^{pag}	Bw_{as}^{pag}
		$\{\mu s\}$	$\{\mu s\}$	$\{ns/byte\}$	$\{KB/s\}$	$\{MB/s\}$
Send	MPICH-GM	9	20	5.330	111.1	187.6
	mpiJava	15	20	5.360	66.67	186.6
	MPJ/Ibis	65	69	5.951	15.38	168.0
Barrier	MPICH-GM	$-3 + 16\lceil lp \rceil$	N/A	N/A	245.9	N/A
	mpiJava	$5 + 15\lceil lp \rceil$	N/A	N/A	230.8	N/A
	MPJ/Ibis	$194 + 73p$	N/A	N/A	11.01	N/A
Broadcast	MPICH-GM	$3 + 8\lceil lp \rceil$	$17 + 23\lceil lp \rceil$	$0.017 + 5.649\lceil lp \rceil$	428.6	663.3
	mpiJava	$20 + 17\lceil lp \rceil$	$33 + 31\lceil lp \rceil$	$0.136 + 5.741\lceil lp \rceil$	170.5	649.4
	MPJ/Ibis	$22 + 21p$	$3 + 24p$	$3.006 + 6.670\lceil lp \rceil$	41.90	505.3
Scatter	MPICH-GM	$-7 + 9p$	$1 + 11p$	$4.271 + 0.412\lceil lp \rceil$	45.45	158.9
	mpiJava	$42 + 10p$	$39 + 13p$	$4.336 + 0.421\lceil lp \rceil$	9.146	156.3
	MPJ/Ibis	$37 + 19p$	$8 + 23p$	$4.421 + 0.673\lceil lp \rceil$	6.667	135.9
Gather	MPICH-GM	$7 + 5p$	$13 + 7p$	$3.782 + 0.503\lceil lp \rceil$	29.41	165.4
	mpiJava	$47 + 5p$	$44 + 7p$	$4.981 + 0.174\lceil lp \rceil$	11.19	140.4
	MPJ/Ibis	$78 + 8p$	$83 + 16\lceil lp \rceil$	$6.216 + 0.487\lceil lp \rceil$	6.818	114.8
Allgather	MPICH-GM	$-10 + 15p$	$3 + 19p$	$5.272 + 1.093\lceil lp \rceil$	75.00	1653
	mpiJava	$30 + 17p$	$41 + 23p$	$8.489 + 0.479\lceil lp \rceil$	52.77	1532
	MPJ/Ibis	$17 + 61p$	$4 + 72p$	$4.096 + 2.970\lceil lp \rceil$	16.05	997.6
Alltoall	MPICH-GM	$-10 + 13p$	$-6 + 16p$	$4.182 + 2.690\lceil lp \rceil$	75.76	1004
	mpiJava	$37 + 15p$	$28 + 19p$	$7.371 + 1.83\lceil lp \rceil$	54.15	1014
	MPJ/Ibis	$296 + 523p$	$213 + 465p$	$5.810 + 3.857\lceil lp \rceil$	1.731	706.3
Reduce	MPICH-GM	$12 + 3p$	$9 + 5p$	$2.698 + 10.83\lceil lp \rceil$	250.0	326.0
	mpiJava	$45 + 4p$	$29 + 6p$	$5.161 + 11.16\lceil lp \rceil$	137.6	301.2
	MPJ/Ibis	$107 + 98\lceil lp \rceil$	$63 + 100\lceil lp \rceil$	$7.618 + 15.38\lceil lp \rceil$	30.06	217.0
Allreduce	MPICH-GM	$18 + 4p$	$21 + 6p$	$3.219 + 16.35\lceil lp \rceil$	365.9	437.2
	mpiJava	$44 + 6p$	$58 + 8p$	$4.319 + 15.39\lceil lp \rceil$	214.3	455.4
	MPJ/Ibis	$223 + 290\lceil lp \rceil$	$381 + 256\lceil lp \rceil$	$5.536 + 22.03\lceil lp \rceil$	21.69	320.3
Reducesctr	MPICH-GM	$-3 + 13p$	$2 + 16p$	$9.326 + 10.81\lceil lp \rceil$	77.97	303.2
	mpiJava	$24 + 15p$	$18 + 19p$	$11.37 + 11.51\lceil lp \rceil$	60.37	277.6
	MPJ/Ibis	$13 + 76p$	$7 + 89p$	$13.91 + 17.83\lceil lp \rceil$	12.97	187.0
Scan	MPICH-GM	$13 + 4p$	$31 + 6p$	$-4.487 + 9.284\lfloor 2lp \rfloor$	194.8	357.7
	mpiJava	$50 + 6p$	$67 + 8p$	$-0.234 + 10.15\lfloor 2lp \rfloor$	102.7	296.9
	MPJ/Ibis	$-1 + 97p$	$9 + 112p$	$3.380 + 21.62p$	9.671	42.94

Table 2.2: SCI-single: analytical models and peak aggregated metrics ($lp = \log_2 p$)

Primitive	Library	$t_0(p)$	$t_i(p)$	$t_b(p)$	π_0^{pag}	Bw_{as}^{pag}
		$\{\mu s\}$	$\{\mu s\}$	$\{ns/byte\}$	$\{KB/s\}$	$\{MB/s\}$
Send	ScaMPI	4	13	3.890	250.0	257.1
	SCI-MPICH	6	5	4.560	166.7	219.3
	mpiJava	10	11	3.924	100.0	254.8
	MPJ/Ibis	49	43	4.272	20.41	234.1
Barrier	ScaMPI	$7 + 0.4p$	N/A	N/A	686.2	N/A
	SCI-MPICH	$-2 + 9\lceil lp \rceil$	N/A	N/A	280.0	N/A
	mpiJava	$8 + 1.2p$	N/A	N/A	397.7	N/A
	MPJ/Ibis	$133 + 48p$	N/A	N/A	13.54	N/A
Broadcast	ScaMPI	$6\lceil lp \rceil$	$12 + 8\lceil lp \rceil$	$-0.093 + 4.099\lceil lp \rceil$	388.9	573.6
	SCI-MPICH	$6\lceil lp \rceil$	$17 + 7\lceil lp \rceil$	$3.403 + 2.987\lceil lp \rceil$	388.9	566.1
	mpiJava	$33 + 7\lceil lp \rceil$	$59 + 9\lceil lp \rceil$	$0.391 + 4.451\lceil lp \rceil$	129.6	509.3
	MPJ/Ibis	$-7 + 15p$	$-9 + 16p$	$0.720 + 4.870\lceil lp \rceil$	61.95	456.6
Scatter	ScaMPI	$-5 + 6p$	$2 + 8p$	$2.714 + 0.251\lceil lp \rceil$	71.43	252.4
	SCI-MPICH	$5 + 2p$	$19 + 5p$	$2.011 + 0.718\lceil lp \rceil$	57.69	217.6
	mpiJava	$27 + 6p$	$58 + 11p$	$2.443 + 0.394\lceil lp \rceil$	14.71	241.4
	MPJ/Ibis	$11p$	$-16 + 16p$	$2.412 + 0.511\lceil lp \rceil$	19.23	221.8
Gather	ScaMPI	$4 + p$	$18 + 2p$	$0.612 + 1.222\lceil lp \rceil$	93.75	272.6
	SCI-MPICH	$2 + 2p$	$22 + 3p$	$2.139 + 0.719\lceil lp \rceil$	83.33	209.7
	mpiJava	$36 + p$	$53 + 4p$	$1.411 + 0.989\lceil lp \rceil$	19.89	221.3
	MPJ/Ibis	$54 + p$	$6 + 2\lceil lp \rceil$	$1.333 + 0.970\lceil lp \rceil$	14.11	229.1
Allgather	ScaMPI	$-6 + 14\lceil lp \rceil$	$12 + 18\lceil lp \rceil$	$3.510 + 1.327\lceil lp \rceil$	218.8	1051
	SCI-MPICH	$-1 + 5p$	$13 + 9p$	$1.936 + 2.571\lceil lp \rceil$	201.9	816.1
	mpiJava	$23 + 16\lceil lp \rceil$	$49 + 22\lceil lp \rceil$	$2.963 + 1.603\lceil lp \rceil$	110.9	1013
	MPJ/Ibis	$32p$	$-15 + 37p$	$1.101 + 2.679\lceil lp \rceil$	30.76	861.8
Alltoall	ScaMPI	$-10 + 8p$	$3 + 11p$	$1.693 + 2.310\lceil lp \rceil$	166.7	811.8
	SCI-MPICH	$-6 + 9p$	$12 + 12p$	$2.412 + 2.230\lceil lp \rceil$	106.1	769.1
	mpiJava	$22 + 9p$	$39 + 14p$	$2.347 + 2.120\lceil lp \rceil$	74.47	804.0
	MPJ/Ibis	$92 + 307p$	$73 + 271p$	$1.408 + 2.658\lceil lp \rceil$	2.747	746.1
Reduce	ScaMPI	$1 + 6\lceil lp \rceil$	$7 + 9\lceil lp \rceil$	$9.834 + 1.761\lceil lp \rceil$	368.4	463.1
	SCI-MPICH	$7 + 2p$	$18 + 4p$	$-3.718 + 6.381\lceil lp \rceil$	304.3	453.8
	mpiJava	$13 + 8\lceil lp \rceil$	$24 + 11\lceil lp \rceil$	$9.681 + 1.911\lceil lp \rceil$	189.2	454.1
	MPJ/Ibis	$26 + 42\lceil lp \rceil$	$6 + 42\lceil lp \rceil$	$1.507 + 9.299\lceil lp \rceil$	46.05	238.1
Allreduce	ScaMPI	$-1 + 12\lceil lp \rceil$	$11 + 15\lceil lp \rceil$	$9.281 + 2.536\lceil lp \rceil$	400.0	828.9
	SCI-MPICH	$11 + 5p$	$14 + 6p$	$5.591 + 3.859\lceil lp \rceil$	274.5	815.5
	mpiJava	$7 + 15\lceil lp \rceil$	$26 + 18\lceil lp \rceil$	$8.819 + 3.048\lceil lp \rceil$	269.2	779.4
	MPJ/Ibis	$-60 + 258\lceil lp \rceil$	$39 + 165\lceil lp \rceil$	$0.666 + 15.49\lceil lp \rceil$	19.61	297.0
Reducesctr	ScaMPI	$-1 + 8p$	$17 + 10p$	$12.51 + 2.068\lceil lp \rceil$	125.0	420.8
	SCI-MPICH	$-6 + 9p$	$5 + 12p$	$9.138 + 2.345\lceil lp \rceil$	125.0	486.9
	mpiJava	$23 + 9p$	$39 + 12p$	$13.04 + 2.149\lceil lp \rceil$	82.89	404.1
	MPJ/Ibis	$42 + 25p$	$31 + 29p$	$4.267 + 10.05\lceil lp \rceil$	32.54	228.8
Scan	ScaMPI	$-9 + 6p$	$13 + 9p$	$-3.361 + 5.183p$	333.3	183.7
	SCI-MPICH	$-1 + 4p$	$10 + 10p$	$3.799 + 1.544\lfloor 2lp \rfloor$	225.8	701.8
	mpiJava	$19 + 7p$	$42 + 12p$	$-5.423 + 8.299p$	93.33	114.8
	MPJ/Ibis	$-62 + 39p$	$-77 + 43p$	$-5.650 + 8.989p$	62.50	105.6

Table 2.3: SCI-dual: analytical models and peak aggregated metrics ($lp = \log_2 p$)

Primitive	Library	$t_0(p)$	$t_i(p)$	$t_b(p)$	π_0^{pag}	Bw_{as}^{pag}
		$\{\mu s\}$	$\{\mu s\}$	$\{ns/byte\}$	$\{KB/s\}$	$\{MB/s\}$
Barrier	ScaMPI	$5 + 2\lceil lp \rceil$	N/A	N/A	1154	N/A
	SCI-MPICH	$-169 + 140\lceil lp \rceil$	N/A	N/A	38.36	N/A
	mpiJava	$11 + \lceil lp \rceil$	N/A	N/A	1000	N/A
	MPJ/Ibis	$204 + 42p$	N/A	N/A	17.12	N/A
Broadcast	ScaMPI	$-3 + 6\lceil lp \rceil$	$7 + 9\lceil lp \rceil$	$-0.605 + 4.297\lceil lp \rceil$	714.2	904.6
	SCI-MPICH	$-11 + 11\lceil lp \rceil$	$3 + 18\lceil lp \rceil$	$-0.531 + 4.919\lceil lp \rceil$	454.5	783.5
	mpiJava	$21 + 7\lceil lp \rceil$	$39 + 12\lceil lp \rceil$	$-0.629 + 4.371\lceil lp \rceil$	306.1	889.9
	MPJ/Ibis	$6 + 15p$	$2 + 15p$	$-2.096 + 4.887\lceil lp \rceil$	60.97	859.5
Scatter	ScaMPI	$-12 + 6p$	$3 + 9p$	$2.199 + 0.339\lceil lp \rceil$	62.50	272.1
	SCI-MPICH	$6 + 2p$	$21 + 7p$	$2.158 + 1.702\lceil lp \rceil$	53.57	134.8
	mpiJava	$17 + 6p$	$38 + 13p$	$2.833 + 0.212\lceil lp \rceil$	18.29	254.7
	MPJ/Ibis	$-1 + 12p$	$-21 + 17p$	$2.417 + 0.408\lceil lp \rceil$	15.96	240.3
Gather	ScaMPI	$7 + 2p$	$34 + 4p$	$0.921 + 0.949\lceil lp \rceil$	50.00	266.1
	SCI-MPICH	$-41 + 35p$	$-3 + 53p$	$0.941 + 1.778\lceil lp \rceil$	7.575	166.8
	mpiJava	$41 + p$	$51 + 6p$	$1.037 + 0.944\lceil lp \rceil$	17.85	256.4
	MPJ/Ibis	$83 + 3p$	$70 + 4\lceil lp \rceil$	$0.968 + 0.995\lceil lp \rceil$	8.177	253.5
Allgather	ScaMPI	$4 + 2p$	$24 + 4p$	$4.515 + 1.863\lceil lp \rceil$	442.7	1332
	SCI-MPICH	$55 + 28p$	$63 + 33p$	$11.42 + 3.688\lceil lp \rceil$	31.68	608.9
	mpiJava	$41 + 2p$	$49 + 4p$	$5.831 + 1.592\lceil lp \rceil$	218.3	1306
	MPJ/Ibis	$-51 + 50p$	$-105 + 55p$	$2.713 + 2.374\lceil lp \rceil$	20.13	1305
Alltoall	ScaMPI	$-24 + 14p$	$4 + 18p$	$0.369 + 4.499\lceil lp \rceil$	93.75	816.7
	SCI-MPICH	$-221 + 103p$	$-193 + 121p$	$-2.97 + 8.391\lceil lp \rceil$	15.71	490.3
	mpiJava	$8 + 14p$	$35 + 21p$	$1.190 + 4.331\lceil lp \rceil$	64.66	810.2
	MPJ/Ibis	$-57 + 377p$	$-84 + 315p$	$-3.012 + 5.481\lceil lp \rceil$	2.510	793.1
Reduce	ScaMPI	$9 + p$	$8 + 2p$	$6.519 + 3.352\lceil lp \rceil$	600.0	752.7
	SCI-MPICH	$38 + 23p$	$51 + 38p$	$8.017 + 3.695\lceil lp \rceil$	36.94	657.9
	mpiJava	$24 + p$	$38 + 3p$	$7.598 + 3.616\lceil lp \rceil$	375.0	679.9
	MPJ/Ibis	$74 + 38\lceil lp \rceil$	$1 + 48\lceil lp \rceil$	$1.748 + 8.176\lceil lp \rceil$	66.37	435.4
Allreduce	ScaMPI	$7 + 2p$	$9 + 4p$	$11.41 + 3.693\lceil lp \rceil$	769.2	1145
	SCI-MPICH	$198 + 71p$	$228 + 83p$	$-15.03 + 20.94\lceil lp \rceil$	22.48	436.4
	mpiJava	$29 + 2p$	$41 + 5p$	$11.04 + 4.177\lceil lp \rceil$	491.8	1081
	MPJ/Ibis	$-61 + 288\lceil lp \rceil$	$-3 + 192\lceil lp \rceil$	$-9.143 + 22.39\lceil lp \rceil$	27.50	373.1
Reducescttr	ScaMPI	$-10 + 9p$	$3 + 11p$	$10.48 + 3.248\lceil lp \rceil$	144.2	679.0
	SCI-MPICH	$-673 + 216p$	$-540 + 239p$	$7.711 + 3.62\lceil lp \rceil$	19.63	718.2
	mpiJava	$22 + 8p$	$41 + 14p$	$11.31 + 3.761\lceil lp \rceil$	106.2	604.7
	MPJ/Ibis	$62 + 24p$	$46 + 28p$	$3.563 + 9.245\lceil lp \rceil$	35.73	393.1
Scan	ScaMPI	$-5 + 4p$	$1 + 6p$	$-2.939 + 5.050p$	272.7	192.7
	SCI-MPICH	$-24 + 82p$	$17 + 87p$	$-0.726 + 2.015\lfloor 2lp \rfloor$	11.64	1604
	mpiJava	$16 + 5p$	$32 + 8p$	$-4.596 + 7.903p$	156.2	123.1
	MPJ/Ibis	$-85 + 49p$	$-33 + 48p$	$-11.68 + 8.462p$	27.03	135.3

Table 2.4: SCI-dual SMT: analytical models and peak aggregated metrics ($lp = \log_2 p$)

Primitive	Library	$t_0(p)$ $\{\mu s\}$	$t_i(p)$ $\{\mu s\}$	$t_b(p)$ $\{ns/byte\}$	π_0^{pag} $\{KB/s\}$	Bw_{as}^{pag} $\{MB/s\}$
Barrier	ScaMPI	$3 + 2\lceil lp \rceil$	N/A	N/A	2067	N/A
	mpiJava	$8 + 4\lceil lp \rceil$	N/A	N/A	1937	N/A
	MPJ/Ibis	$331 + 32p$	N/A	N/A	22.88	N/A
Broadcast	ScaMPI	$-7 + 7\lceil lp \rceil$	$-3 + 11\lceil lp \rceil$	$3.210 + 4.480\lceil lp \rceil$	1107	1210
	mpiJava	$45 + 9\lceil lp \rceil$	$57 + 14\lceil lp \rceil$	$-1.097 + 5.719\lceil lp \rceil$	344.4	1127
	MPJ/Ibis	$11 + 15p$	$37 + 13p$	$-0.997 + 5.397\lceil lp \rceil$	63.14	1191
Scatter	ScaMPI	$-17 + 6p$	$3 + 8p$	$0.519 + 1.150\lceil lp \rceil$	28.22	220.4
	mpiJava	$18 + 8p$	$41 + 11p$	$1.63 + 0.937\lceil lp \rceil$	10.67	197.0
	MPJ/Ibis	$9 + 13p$	$-3 + 16p$	$0.553 + 1.295\lceil lp \rceil$	7.743	197.2
Gather	ScaMPI	$15 + 2p$	$83 + 5p$	$-1.403 + 2.017\lceil lp \rceil$	28.23	188.3
	mpiJava	$55 + 2p$	$131 + 9p$	$-1.031 + 2.053\lceil lp \rceil$	12.32	170.6
	MPJ/Ibis	$71 + 3p$	$-39 + 44\lceil lp \rceil$	$0.344 + 1.835\lceil lp \rceil$	9.211	149.6
Allgather	ScaMPI	$-1 + 3p$	$45 + 5p$	$10.23 + 1.987\lceil lp \rceil$	342.4	1585
	mpiJava	$74 + 2p$	$128 + 7p$	$9.648 + 2.238\lceil lp \rceil$	231.7	1534
	MPJ/Ibis	$-68 + 67p$	$-162 + 79p$	$5.645 + 3.320\lceil lp \rceil$	16.83	1437
Alltoall	ScaMPI	$-123 + 36p$	$-83 + 43p$	$-7.585 + 12.41\lceil lp \rceil$	42.42	569.2
	mpiJava	$-114 + 44p$	$-45 + 60p$	$-5.969 + 12.10\lceil lp \rceil$	29.41	568.5
	MPJ/Ibis	$-575 + 547p$	$-773 + 465p$	$-1.950 + 11.92\lceil lp \rceil$	1.842	537.7
Reduce	ScaMPI	$14 + p$	$31 + 2p$	$13.31 + 4.690\lceil lp \rceil$	673.9	843.3
	mpiJava	$47 + p$	$77 + 3p$	$12.91 + 5.796\lceil lp \rceil$	392.4	740.0
	MPJ/Ibis	$56 + 51\lceil lp \rceil$	$-11 + 49\lceil lp \rceil$	$1.486 + 10.93\lceil lp \rceil$	99.68	552.2
Allreduce	ScaMPI	$15 + p$	$33 + 2p$	$22.31 + 5.513\lceil lp \rceil$	1319	1243
	mpiJava	$51 + 2p$	$83 + 4p$	$20.70 + 7.131\lceil lp \rceil$	539.1	1100
	MPJ/Ibis	$-135 + 358\lceil lp \rceil$	$82 + 219\lceil lp \rceil$	$-133.3 + 62.74\lceil lp \rceil$	37.46	306.3
Reducescttr	ScaMPI	$7 + 8p$	$30 + 11p$	$18.72 + 4.798\lceil lp \rceil$	121.6	748.5
	mpiJava	$44 + 8p$	$79 + 15p$	$15.92 + 6.586\lceil lp \rceil$	106.6	654.4
	MPJ/Ibis	$81 + 27p$	$75 + 28p$	$6.588 + 11.09\lceil lp \rceil$	33.83	515.3
Scan	ScaMPI	$-3 + 5p$	$6 + 9p$	$-7.813 + 5.407p$	197.4	197.5
	mpiJava	$29 + 6p$	$41 + 11p$	$-7.864 + 9.645p$	140.3	103.1
	MPJ/Ibis	$-120 + 58p$	$-325 + 78p$	$-5.730 + 8.071p$	20.35	122.8

Java Communication Libraries

From the models it can be observed that mpiJava adds little overhead to the underlying native message-passing library. In fact, mpiJava performance is quite similar to the MPI results, especially for long messages for which this overhead is almost negligible, as their peak aggregated bandwidths (Bw_{as}^{pag}) are quite similar. However, the peak aggregated bandwidth of mpiJava can be slightly overestimated as its higher $t_0(p)$ values can slightly underestimate its $t_b(p)$ results (and hence increase Bw_{as}^{pag}). With respect to MPJ/Ibis, both the transfer time and, mainly, the start-up time, increase significantly with respect to the native libraries. This overhead corresponds to: (1) the additional communication layers involved in the communication, TCPIbis sockets and Ibis Portability Layer (IPL), and (2) the interpreted nature of the JVM, basic for the portability of the library. The most immediate way of running this library on high-speed interconnects is on top of IP emulation libraries: IP over GM on Myrinet and ScaIP on SCI. Nevertheless, in order to ensure a fair comparison, MPJ/Ibis has been slightly adapted to run on top of Sockets-GM on Myrinet, and on top of SCI Sockets.

Regarding peak performance metrics, it can be observed that MPJ/Ibis collective primitives generally present the highest values (best performance) for π_0^{pag} on SCI-single configuration, except for reduction primitives (Reduce, Allreduce, Reduce_scatter and Scan). However, MPJ/Ibis obtains the highest Bw_{as}^{pag} performance on SCI-dual and SCI-dual SMT.

2.4. Analysis of the Performance Models

2.4.1. Point-to-Point Communication

In order to assess the accuracy of the performance models derived in Section 2.3 Figure 2.2 shows experimentally measured (empty symbols) and estimated (filled symbols) latencies and bandwidths of the Send primitive as a function of the message length for the different networks. Bandwidth graphs are useful to compare long message performance, whereas latency graphs serve to compare short message performance.

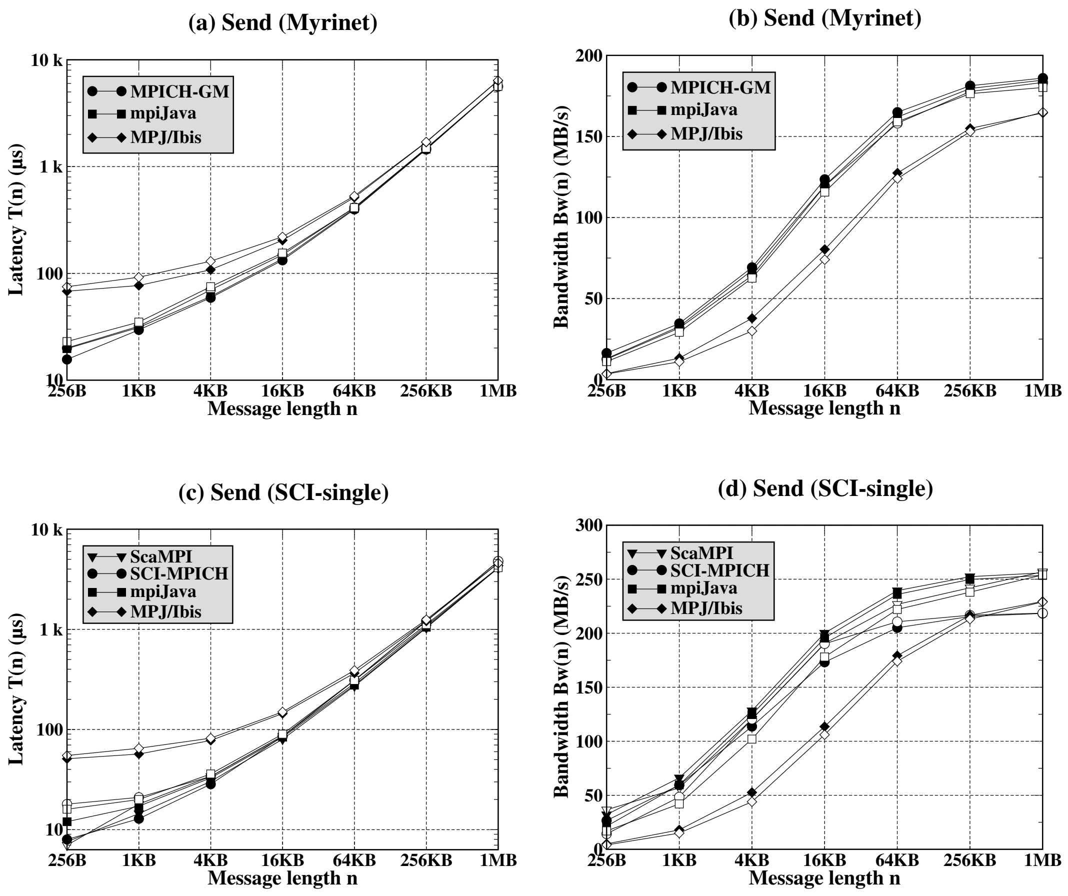


Figure 2.2: Measured and estimated latencies and bandwidths of Send

Regarding MPI C point-to-point primitives (see Tables 2.1–2.2), asymptotic bandwidths are 188 MB/s for MPICH-GM Send, and 257 MB/s for ScaMPI Send. Thus, network bandwidth is limiting the performance of MPICH-GM on Myrinet (the maximum bandwidth is 250 MB/s), but not especially on SCI (SCI maximum bandwidth is 666 MB/s). Experimentally measured MPI C point-to-point start-ups, $9\mu s$ on Myrinet and $4\mu s$ on SCI, are very close to the theoretical values of the networks (see Section 2.3.2). The different computational power of the nodes has a minor influence on these values.

With respect to the Java message-passing implementations, on the one hand, mpiJava obtains results quite similar to its underlying MPI implementation. On the other hand, MPJ/Ibis shows start-ups of $65\mu s$ on Myrinet and $49\mu s$ on SCI, and values of t_b slightly higher (around 10%) than the native library values. This overhead, quite distant from the theoretical values of the high-speed interconnects, especially for t_0, must be attributed to the messaging protocol (around $40\mu s$ overhead for t_0 both on Myrinet and SCI). The underlying communication library, TCPIbis sockets, shows $t_0 = 22\mu s$ on Myrinet, and $t_0 = 11\mu s$ on SCI, thanks to the use of high performance sockets libraries (Sockets-GM and SCI Sockets). Using IP emulation libraries TCPIbis obtains $t_0 = 196\mu s$ on Myrinet and $t_0 = 131\mu s$ on SCI. The benefits of using the high performance native sockets libraries instead of IP emulations are clear on MPJ/Ibis. However, using message-passing libraries based on RMI, such as CCJ [72] or JMPI [67], the impact of the use of high performance sockets libraries is reduced as the protocol overheads are much higher (from $0.5ms$ to $4ms$), as reported in [88].

2.4.2. Collective Communications

Measured and estimated bandwidths for some collective primitives are depicted in the graphs of Figures 2.3 and 2.4. The results were obtained using the maximum number of available processors for each cluster configuration (16 for Myrinet and SCI-dual, 8 for SCI-single and 32 for SCI-dual SMT). Note that bandwidths are not aggregated, as they are computed simply by dividing n by $T(n, p)$. In many cases, the estimated values (filled symbols) are hidden by the measured values (empty symbols), which means a good modeling. As expected, the bandwidth of the mpiJava routines and the underlying MPI C implementations are very similar (mpiJava calls to native MPI have low overhead), and pure Java primitives show lower performance. In fact, MPJ/Ibis t_b is slightly higher than the native library value, and therefore, the derived performance metric, Bw_{as}^{pag}, presents slightly lower values than the MPI libraries. Nevertheless, MPJ/Ibis shows much lower performance than the native implementations for short messages. An example is the quite poor performance, especially for short and medium-size messages, of the MPJ/Ibis Alltoall, which involves an important number of short messages (see Figures 2.4(b), 2.4(d), 2.4(f) and 2.4(h), and the metric π_0^{pag} in Tables 2.1–2.4).

A gap between Myrinet and SCI-single short message performance can be observed for all the libraries evaluated. For instance, the 4 KB MPI C Broadcast bandwidth is 3.3 times higher on SCI-single than on Myrinet (see Figures 2.3(a)

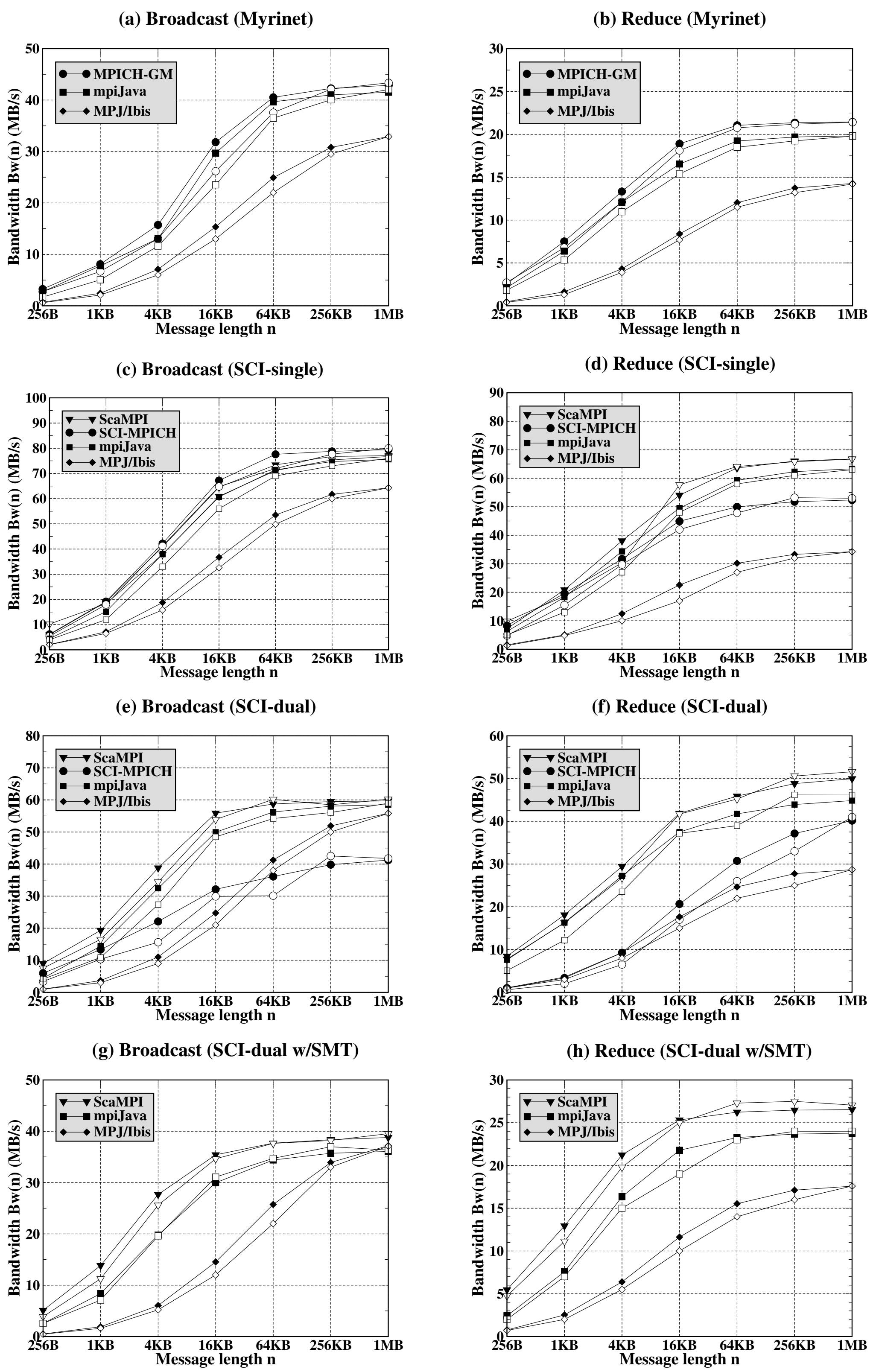


Figure 2.3: Measured and estimated bandwidths for Broadcast and Reduce

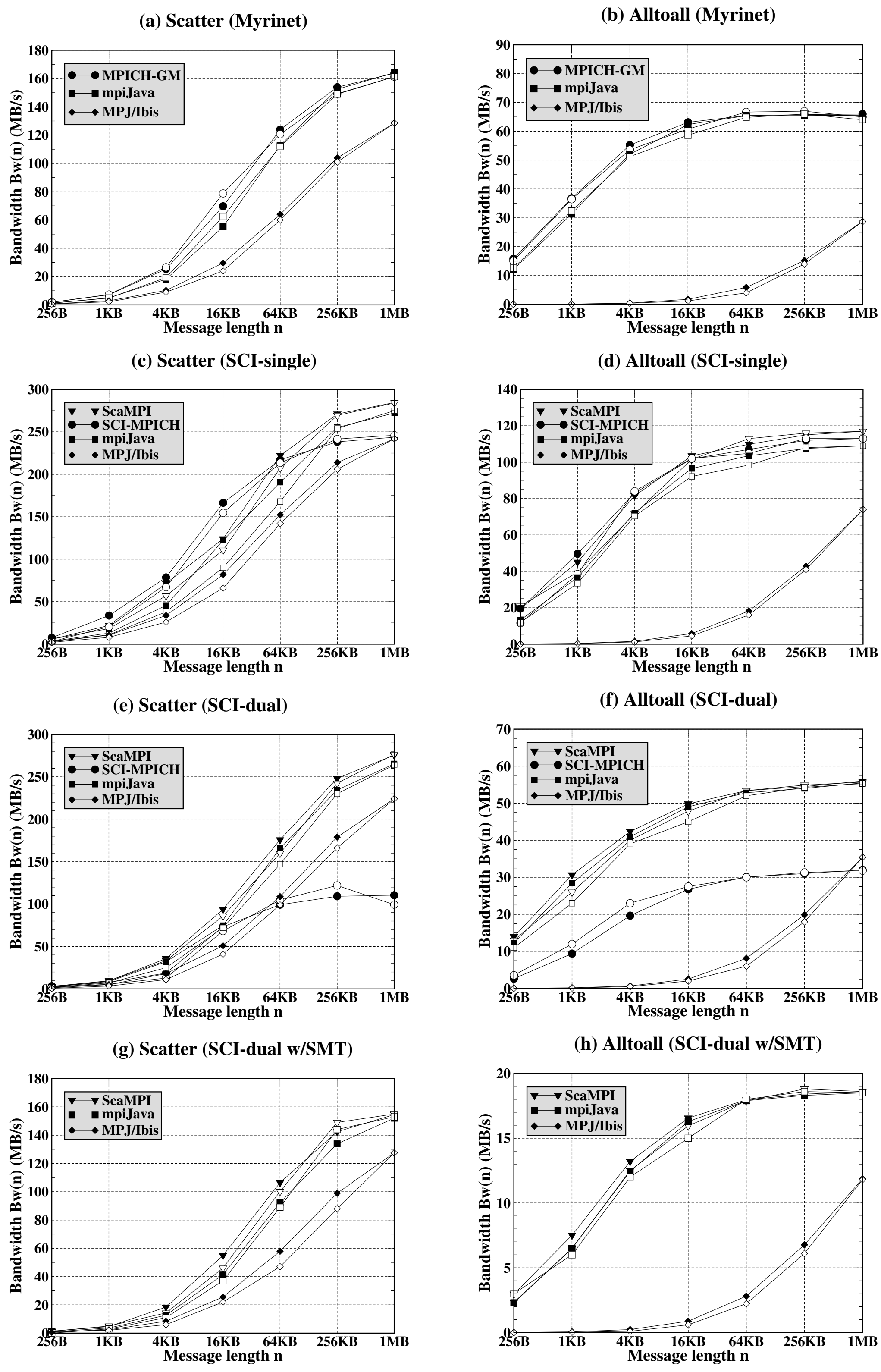


Figure 2.4: Measured and estimated bandwidths for Scatter and Alltoall

and 2.3(c)). Similarly, the 4 KB MPI C Reduce bandwidth is 2.8 times higher on SCI-single (see Figures 2.3(b) and 2.3(d)). A higher t_0 on Myrinet is the main cause of this lower performance. Regarding the different system configurations, it can be observed that the highest bandwidths are obtained by SCI-single (see Figures 2.3(c), 2.3(d), 2.4(c) and 2.4(d)), followed by SCI-dual, and finally by Myrinet and SCI-dual SMT.

2.4.3. Model-based Performance Optimization

Message-passing performance models have been used to identify inefficient communication primitives. From this process, it has been detected that ScaMPI and MPJ/Ibis Scan show a linear complexity ($O(p)$), whereas the MPICH implementations, MPICH-GM and SCI-MPICH, present a logarithmic complexity ($O(log_2p)$), and therefore a more scalable primitive implementation. Other implementations, e.g., MPJ/Ibis Allreduce, just show poor performance. To reduce the inefficiency, a primitive can be replaced by a more efficient equivalent combination of primitives. Examples of equivalences in message-passing libraries are: Broadcast=Scatter+ Allgather (Van der Geijn algorithm [97]), Allgather=Gather+Broadcast, Reduce_scatter=Reduce+Scatterv and Allreduce=Reduce+Broadcast. The replacement of primitives is done for the combinations of n and p that satisfy the following inequality: $T_{original\_primitive}(n,p) > T_{primitive\#1}(n,p) + T_{primitive\#2}(n,p)$ based on the models, where *original_primitive* is equivalent to *primitive#1+primitive#2*.

For illustrative purposes, some examples of latency reduction using this approach are presented in Table 2.5. Several combinations of n and p for replacing primitives are shown, together with some examples of replacement. The obtained latency reductions for these examples are shown in the last column. mpiJava examples have been omitted as this library performs similarly to the underlying native library. The Reduce_scatter primitive has also been omitted as it is implemented in MPICH-GM, ScaMPI and MPJ/Ibis using a Reduce followed by a Scatterv. The equivalent combination of primitives behaves as in one of the following cases: (1) can present lower start-ups than the original primitive (e.g., MPICH-GM, ScaMPI and MPJ/Ibis Allgather in Table 2.5), and thus the replacement is done for short messages; (2) can show lower transfer times than the original primitive (e.g., MPICH-GM, ScaMPI and MPJ/Ibis Broadcast), replacing the original primitive for long messages; (3) can present both situations (e.g., MPJ/Ibis Allreduce), replacing always the original primitive. The mean latency reduction of the examples shown in Table 2.5 is 37%.

This model-based performance optimization can be easily automatized. By determining cross-over points between communication primitives and its equivalent combinations, the message-passing library can replace inefficient primitives by their equivalents at runtime. Related works on automatic collective communication optimization [13, 103] use the P-LogP model, but it requires to know the algorithm used in the collective implementation. Moreover, these works present only Broadcast and Scatter optimizations, due to the higher complexity of their approaches. In fact, in order to determine the best communication pattern, the optimization procedure consists of finding out the best algorithm for each message size, and the best segment size to fragment the message. This procedure must be repeated for each number of processors considered, although the number of repetitions can be reduced with the aid of the P-LogP model. The main contribution of our higher level approach is its simplicity, as once the models of the collective primitives are obtained, the performance optimization of the collectives is straightforward, without knowledge of their concrete implementations or involving an additional and costly procedure.

2.5. Kernel Benchmarking

A kernel benchmarking has been carried out in order to analyze the impact of message-passing overhead on the overall application performance. This benchmarking has also served to analyze the influence of message-passing overhead on multiple processor nodes (see Section 2.6). As will be shown, the results of both analyses are consistent with the predictions obtained from the models. This process has been carried out on the SCI cluster described in Section 2.3.1, and the selected benchmarks have been the MPJ kernels from the Java Grande Forum (JGF) Benchmark Suite [84] and their corresponding MPI C versions. The kernels are, from higher to lower computation/communication ratio: Series, Crypt, SOR, Sparse and LUFact. For each of them there are three predetermined problem sizes: small (A), medium (B) and large (C). This benchmark suite is the most widely used in evaluation of Java for HPC.

Table 2.5: Parameter values for latency (T) reduction through primitive substitution

	Library	**Testbed**	**Parameter values{(n,p)}**	**Example(n,p)**	**T**
Broadcast	MPICH-GM	Myrinet	{(n>64KB,p=8),(n>78KB,p=16)}	n=256KB,p=16	↓ 20%
	ScaMPI	SCI-single	{(n>103KB,p=8)}	n=256KB,p=8	↓ 18%
		SCI-dual	{(n>128KB,p=16)}	n=256KB,p=16	↓ 13%
	MPJ/Ibis	Myrinet	{(n>273KB,p=8),(n>994KB,p=16)}	n=512KB,p=8	↓ 43%
		SCI-single	{(n>462KB,p=4),(n>202KB,p=8)}	n=1MB,p=8	↓ 53%
		SCI-dual	{(n>1173KB,p=16)}	n=2MB,p=16	↓ 9%
Allgather	MPICH-GM	Myrinet	{(n<256B,p=8),(n<2KB,p=16)}	n=256B,p=16	↓ 20%
	ScaMPI	SCI-single	{(n<128B,p=4),(n<256B,p=8)}	n=128B,p=8	↓ 43%
	MPJ/Ibis	Myrinet	{(n<25KB,p=8),(n<40KB,p=16)}	n=1KB,p=8	↓ 42%
		SCI-single	{(n<2KB,p=4),(n<7KB,p=8)}	n=1KB,p=8	↓ 35%
		SCI-dual	{(n<17KB,p=8),(n<45KB,p=16)}	n=1KB,p=16	↓ 50%
		SCI-SMT	{(n<66KB,p=8),(n<109KB,p=16)}	n=1KB,p=16	↓ 61%
Allreduce	MPJ/Ibis	Myrinet	Replace always	n=1KB, p=8	↓ 39%
				n=256KB, p=8	↓ 18%
		SCI-single	Replace always	n=1KB,p=8	↓ 50%
				n=256KB, p=8	↓ 24%
		SCI-dual	Replace always	n=1KB,p=16	↓ 44%
				n=256KB, p=16	↓ 41%
		SCI-SMT	Replace always	n=1KB,p=16	↓ 52%
				n=256KB, p=16	↓ 65%

Figure 2.5 shows the speedups obtained from running LUFact and Series kernels using ScaMPI, mpiJava, MPJ/Ibis and MPJ Express on the SCI cluster. These kernels have been selected as representatives of communication-intensive applications (LUFact) and computation-intensive applications (Series). In fact, LUFact performs an important number of short message broadcasts (1000, 2000 and 4000 for problem sizes A, B and C, respectively), whereas Series only involves gathering two long arrays (the size of each array is 80 KB, 800 KB and 8 MB for problem sizes A, B and C, respectively). Labels in the x-axis represent the kernel problem size (A,B,C) and the number of processes per node (1, 2 and 4; using SCI-single, SCI-dual and SCI-dual SMT configurations, respectively). Regarding the speedup results, ScaMPI shows generally the best scalability; mpiJava presents slightly lower performance than ScaMPI; and MPJ/Ibis and MPJ Express results are lower than mpiJava results. LUFact shows modest speedups, and even slowdowns for size A, especially for A4, and also for size B with MPJ Express. Series presents significantly higher speedups than LUFact, obtaining almost linear speedups (i.e., the speedups are similar to the number of processes used) except for 32 processes. With respect to the "pure" Java libraries, MPJ Express shows slightly better performance than MPJ/Ibis for Series, whereas MPJ/Ibis performs better for LUFact. These differences can be explained by the fact that MPJ/Ibis uses TCPIbis sockets as communication technology, which has lower t_0 but higher t_b than Java NIO sockets, base of MPJ Express. Thus, MPJ/Ibis performs better for applications with short message communication patterns, whereas MPJ Express shows better performance for medium and long message communication patterns.

Although Sparse, Crypt and SOR experimental results have also been analyzed, they have been omitted for conciseness and only the main conclusions are presented. Thus, on the one hand, Sparse results are slightly lower than LUFact speedups. On the other hand, Crypt and SOR results are similar to Series results, but showing lower speedups, especially for SOR. These results are consistent with the computation/communication ratio of the kernels, which is the main explaining factor of their performance behavior.

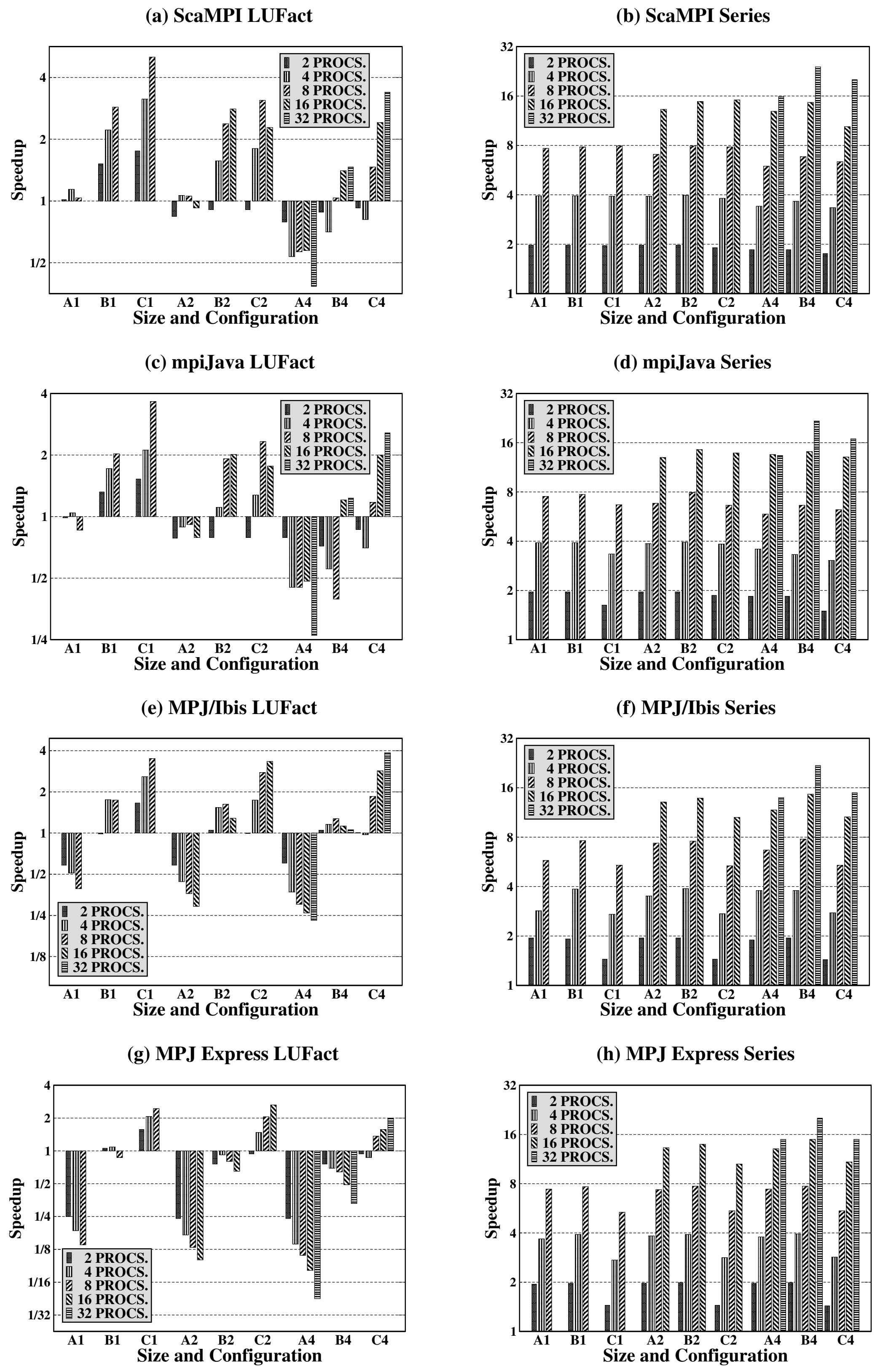


Figure 2.5: Speedups of selected Java Grande kernels

From this benchmarking process, it has been observed that the poorest speedups are obtained with communication-intensive kernels, especially with small problem sizes and using MPJ/Ibis and MPJ Express. However, MPJ/Ibis and MPJ Express show speedups comparable with native libraries performance in the remaining situations.

2.6. Analysis of the Kernel Benchmarking Results on Dual Nodes

The kernel benchmarking has also served to analyze the influence of message-passing overhead on dual nodes with and without hyper-threading enabled. This analysis has been carried out on the SCI cluster using single, dual and dual SMT configurations. ScaMPI and MPJ/Ibis have been selected as representative libraries of native and Java message-passing libraries, respectively.

2.6.1. Performance Analysis on Dual-Processor Nodes

According to the graphs of Figure 2.5, LUFact speedups are higher using one process per node than using two processes (i.e., A1 speedups > A2 speedups, B1 > B2, C1 > C2), whereas Series speedups remain similar. In order to quantify the influence of using two processes per node instead of 1 a new metric has been derived. This metric is the ratio $T_{SCI-single}(p)/T_{SCI-dual}(p)$ for p processes, where p nodes are used on SCI-single and $p/2$ nodes on SCI-dual. A ratio higher than 1 means that the kernel benefits from running p processes on SCI-dual, instead of running on SCI-single. From Table 2.6 it can be observed that LUFact, Sparse and SOR, communication-intensive kernels, do not benefit from using two processes per node, whereas Series and Crypt, computation-intensive kernels, can slightly benefit from this. The reason is that using two processes per node each process has available approximately half of the resources of the node, instead of the resources of the whole node (as it happens with one process per node). As communication-intensive kernels need more resources for communications than computation-intensive kernels (the message-passing libraries use additional buffers and threads when communicating), the performance benefits of intra-node communication do not make up for the reduction of available resources for inter-node communication.

After assessing that running p processes on $p/2$ nodes instead of on p nodes only improves performance on computation-intensive kernels, another interesting evaluation is the comparison of the kernel results on nd nodes assigning one process per node against the kernel results on nd nodes with two processes per node. The associated metric is the ratio $T_{SCI-single}(nd)/T_{SCI-dual}(2 \times nd)$. A ratio higher than 1 means that the kernel benefits from running two processes per node instead of running only one for a fixed number of nodes nd. From the discussion in Subsection 2.3.2, both t_0 and t_b are higher on SCI-dual than on SCI-single. Moreover, the communication overhead is higher for $2 \times nd$ processes instead of for nd processes. Thus, clearly the communication cost is higher for $T_{SCI-dual}(2 \times nd)$ than for $T_{SCI-single}(nd)$. Nevertheless, the workload for each of the $2 \times nd$ processes on SCI-dual is approximately half of the workload for each of the nd processes on SCI-single. Therefore, ratios slightly below 2 can be predicted for computation-intensive kernels (Series and Crypt), whereas more modest ratios, even significant slowdowns, can be predicted for communication-intensive kernels (LUFact, Sparse and SOR). Table 2.7 presents the obtained ratios, that are in tune with these predictions.

2.6.2. Performance Analysis on SMT Dual-Processor Nodes

The influence of enabling the hyper-threading has not been taken into account in the previous analyses. This influence can be characterized by the ratio of the runtime on SCI-dual to the runtime on SCI-dual SMT, in both cases using nd nodes. Thus, the metric is $T_{SCI-dual}(2\times nd)/T_{SCI-dual\ SMT}(4\times nd)$, where the number of processes is $2\times nd$ on SCI-dual and $4\times nd$ on SCI-dual SMT. A ratio higher than 1 means that the kernel benefits from enabling hyper-threading, for a fixed number of nodes nd. From Subsection 2.3.2, it can be predicted that both t_0 and t_b are higher on SCI-dual SMT than on SCI-dual. From the first paragraph of Subsection 2.3.1 it can be obtained that the computational performance should be slightly higher (around 30% higher). Thus, it is expected that communication-intensive codes present poorer performance on SCI-dual SMT than on SCI-dual, whereas computation-intensive kernels increase their performance around 30%. Moreover, it is possible to achieve higher performance improvements with benchmarks that especially benefit from

Table 2.6: Ratio $T_{SCI-single}(p)/T_{SCI-dual}(p)$

		Small Size (A)		**Medium Size (B)**		**Large Size (C)**	
		$p = 4$	$p = 8$	$p = 4$	$p = 8$	$p = 4$	$p = 8$
ScaMPI	LUFact	0.93	**1.03**	0.69	0.82	0.56	0.64
	Series	0.98	0.88	0.99	**1.01**	**1.00**	0.99
	SOR	0.70	0.77	0.63	0.70	0.59	0.68
	Sparse	0.56	0.51	0.47	0.53	0.46	0.50
	Crypt	**1.02**	**1.01**	**1.01**	**1.00**	**1.00**	0.98
MPJ/Ibis	LUFact	0.86	0.91	0.88	0.92	0.68	0.79
	Series	**1.22**	**1.27**	**1.00**	0.99	**1.00**	**1.00**
	SOR	0.90	**1.03**	0.85	**1.04**	0.76	0.91
	Sparse	0.89	0.81	0.94	0.83	0.86	0.79
	Crypt	0.98	**1.00**	**1.02**	**1.02**	**1.00**	**1.04**

Table 2.7: Ratio $T_{SCI-single}(nd)/T_{SCI-dual}(2 \times nd)$

		Small Size (A)		**Medium Size (B)**		**Large Size (C)**	
		$4Nodes$	$8Nodes$	$4Nodes$	$8Nodes$	$4Nodes$	$8Nodes$
ScaMPI	LUFact	0.92	0.88	**1.03**	0.99	0.97	0.97
	Series	**1.83**	**1.77**	**1.99**	**1.90**	**1.99**	**1.92**
	SOR	**1.02**	**1.02**	0.99	0.99	0.98	0.98
	Sparse	0.62	0.65	0.71	0.68	0.72	0.69
	Crypt	**1.66**	**1.34**	**1.78**	**1.62**	**1.78**	**1.72**
MPJ/Ibis	LUFact	0.70	0.75	0.93	0.74	**1.08**	0.95
	Series	**2.18**	**2.03**	**1.95**	**1.82**	**1.93**	**1.88**
	SOR	**1.01**	0.82	**1.10**	0.94	**1.08**	0.96
	Sparse	0.58	0.61	0.63	0.64	0.67	0.64
	Crypt	**1.65**	**1.44**	**1.87**	**1.76**	**1.89**	**1.84**

Table 2.8: Ratio $T_{SCI-dual}(2 \times nd)/T_{SCI-dual\ SMT}(4 \times nd)$

		Small Size (A)		**Medium Size (B)**		**Large Size (C)**	
		$2Nodes$	$8Nodes$	$2Nodes$	$8Nodes$	$2Nodes$	$8Nodes$
ScaMPI	LUFact	0.61	0.39	0.78	0.55	0.88	0.72
	Series	**1.41**	**1.23**	**1.72**	**1.54**	**1.73**	**1.21**
	SOR	0.80	0.46	0.87	0.64	0.89	0.52
	Sparse	0.54	0.53	0.74	0.63	0.88	0.65
	Crypt	0.94	0.72	**1.22**	0.90	**1.24**	0.95
MPJ/Ibis	LUFact	0.46	0.49	0.60	0.48	0.70	0.69
	Series	**1.63**	**1.10**	**1.97**	**1.41**	**1.91**	**1.33**
	SOR	0.70	0.49	0.70	0.61	0.75	0.62
	Sparse	0.44	0.39	0.44	0.39	0.49	0.43
	Crypt	**1.24**	**1.07**	**1.31**	**1.18**	**1.41**	**1.22**

parallelization, i.e. codes that show higher parallel efficiencies as the number of processors increases. Table 2.8 shows the obtained ratios, that are in tune with these predictions. Thus, computation-intensive kernels (Series and Crypt) benefit from enabling hyper-threading (up to 41% performance increase for Crypt and 97% for Series, which especially benefits from parallelization), whereas communication-intensive kernels (LUFact, Sparse and SOR) reduce their performance, especially on 8 nodes.

It can be concluded that representative message-passing implementations do not benefit from systems with multiple processor nodes. A solution could be the use of multithreading instead of interprocess communication for handling intra-node communications. The development of shared memory communication protocols for intra-node communications and its combination with current inter-node protocols would achieve higher performance. Nevertheless, the message-passing library must implement thread-safe communication mechanisms, which are a highly interesting feature for multi-core systems. Several related projects, e.g., USFMPI [18] and pCoR [2], propose to integrate multithreading and message-passing communications.

2.7. Chapter 2 Conclusions

The characterization of the message-passing communication overhead on high-speed clusters is extremely important. Message-passing performance is critical for the overall system scalability and performance. Representative native MPI (MPICH-GM, ScaMPI and SCI-MPICH) and Java message-passing libraries (mpiJava, MPJ/Ibis and MPJ Express) have been selected for performance modeling and evaluation. For this purpose, an accurate message-passing communication model, together with a message-passing micro-benchmark suite to derive these models, have been proposed. The predictions obtained by this model have been validated against experimental results obtaining better estimates than preceding models. The estimates have shown only a 7% average absolute relative error. Moreover, performance metrics derived from the models have been used to evaluate message-passing primitives implementations and their performance on high-speed clusters. These models have also served to identify inefficient communication primitives. To solve these inefficiencies, some primitives can be replaced by a more efficient equivalent combination of primitives. This process has obtained important latency reductions and can be easily automatized.

From the analysis of message-passing performance, it can be concluded that native libraries and mpiJava benefit from the low start-up and high bandwidth of the high-speed interconnects. Nevertheless, these libraries are not portable. MPJ/Ibis and MPJ Express overcome this issue, but this involves an important additional overhead.

Besides the message-passing performance analysis on high-speed interconnects, it has been carried out a kernel benchmarking. This process has been performed in order to analyze the influence of message-passing overhead and the use of multiple processor nodes on the overall application performance. The main conclusion is that message-passing implementations, especially "pure" Java libraries, do not take advantage of these systems.

Finally, this work intends to provide parallel programmers and library developers with guidelines for efficiently exploiting high-speed cluster interconnects and multiple processor nodes. The design of low-level communication middleware that increases Java performance on high-speed clusters, where far less research has been done, is the next goal of the Thesis (Chapter 3).

Chapter 3

JFS: High Performance Java Fast Sockets

The next objective of the Thesis is to provide parallel and distributed Java applications with an efficient socket implementation, named Java Fast Sockets (JFS), for high performance computing on multi-core clusters with high-speed networks. By optimizing the widely used socket API, parallel and distributed Java applications based on it improve performance transparently. Several projects have previously attempted to increase Java communication performance, especially on high-speed cluster networks, but they lack desirable features like those discussed in Section 1.1.

JFS optimizes the JVM socket protocol reducing communication overhead, especially for shared memory transfers. Among its main features, JFS: (1) provides efficient high-speed cluster interconnects support (SCI, Myrinet and InfiniBand); (2) optimizes Java IO sockets, more popular and extended than NIO sockets; (3) avoids the need for primitive data type array serialization; (4) significantly reduces buffering and unnecessary copies; (5) implements an optimized shared memory protocol; and (6) it is user and application transparent, no source code modification is necessary to use JFS.

The chapter is organized as follows: Section 3.1 presents the main design features of JFS together with its efficient protocol implementation. Section 3.2 illustrates the results of the micro-benchmarking of JFS conducted on SCI, Myrinet, InfiniBand and Gigabit Ethernet networks, as well as on shared memory. The results indicate that JFS obtains significant performance improvements over Sun JVM sockets. Moreover, JFS has also an important impact on the performance of final applications. Section 3.3 analyzes this impact on representative parallel kernels and applications. Finally, Section 3.4 concludes the chapter with a summary of its main contributions.

3.1. Efficient Java Socket Implementation

The development of an optimized Java socket library poses several challenges such as serialization overhead reduction and protocol performance increase, especially through a more efficient data transfer implementation. JFS has contributed to these goals by: (1) avoiding primitive data type array serialization (see Subsection 3.1.1); (2) reducing buffering and unnecessary copies in the protocol (Subsection 3.1.2); and (3) providing shared memory communication with an optimized transport protocol as will be shown in Subsection 3.1.3.

3.1.1. Serialization Overhead Reduction

Serialization imposes severe performance penalties as this process involves the extraction of the byte values from the data to be sent. An example of this is shown in Listing 3.1, where `java.io.Bits.putInt()` writes an integer `val` to the stream `b` at the position `off`. As Java socket restriction of sending only byte arrays does not hold for native sockets, JFS defines native methods (see Listing 3.2) to transfer primitive data type arrays directly without serialization.

Listing 3.1: Example of a costly serialization operation of an integer value

```
static void putInt(byte[] b, int off, int val) {
    b[off + 3] = (byte) (val >>> 0);
    b[off + 2] = (byte) (val >>> 8);
    b[off + 1] = (byte) (val >>> 16);
    b[off + 0] = (byte) (val >>> 24);
}
```

Listing 3.2: JFS extended API for direct transfer of primitive data type arrays

```
jfs.net.SocketOutputStream.write(int buf[], int offset, int length);
jfs.net.SocketOutputStream.write(double buf[], int offset, int length);
jfs.net.SocketOutputStream.write(float buf[], int offset, int length);
  ...
jfs.net.SocketInputStream.read(int buf[], int offset, int length);
  ...
```

3.1.2. Socket Protocol Optimization

The operation of the Sun JVM sockets has been analyzed in order to avoid its main performance bottlenecks in the implementation of JFS. Figure 3.1 displays the diagram of the operation of Sun JVM sockets representing the data structures used and the path followed by socket messages. A primitive data type array transfer for representativeness and illustrative purposes has been selected. First, `ObjectOutputStream`, the class used to serialize objects, writes `sdata` to a block data buffer (`blockdata`). As recommended, serialized data is buffered in order to reduce the number of accesses to native sockets. Then, the socket library uses the JNI function `GetByteArrayRegion(byte[] buf)` to copy the buffered data to `jvmsock_buf`, a native buffer that is dynamically allocated for messages longer than 2 KB (configurable size). The native socket library and its buffer `nativesock_buf` are involved in the next copy. Then, data is transferred through the network thanks to the network driver. The receiving side operates in reverse order, and thus the whole process involves nine steps: a serialization, three copies, a network transfer, another three copies and a deserialization. Potential optimizations detected in this analysis in order to improve performance are the reduction in the number of copies and the decrease of the serialization overhead.

These optimizations have been included in JFS as shown in Figure 3.2. The function `GetPrimitiveArrayCritical(<primitive data type> {s|r}data[])` allows native code to obtain through JNI a direct pointer to the Java array in order to avoid serialization. Thus, a one-copy protocol can be implemented as only one copy is needed to transfer `sdata` to the native socket library. However, data can be transferred with a zero-copy protocol without involving the CPU on RDMA-capable high-speed cluster interconnects (such as SCI, Myrinet and InfiniBand). This zero-copy protocol obtains higher bandwidths and lower CPU loads than the one-copy protocol, although RDMA imposes a higher start-up latency. Therefore, one-copy is used only for short messages (size below a configurable threshold). A related issue is the receiving strategy, where polling (a busy-loop waiting for message reception) obtains lower start-up latency but higher CPU load than blocking. Thus, polling is preferred only for short messages. These protocols and strategies are handled by JFS. The whole optimized process involves up to two copies and a network communication in the worst case. Furthermore, a potential optimization has been detected for shared memory communication, presented in the following subsection.

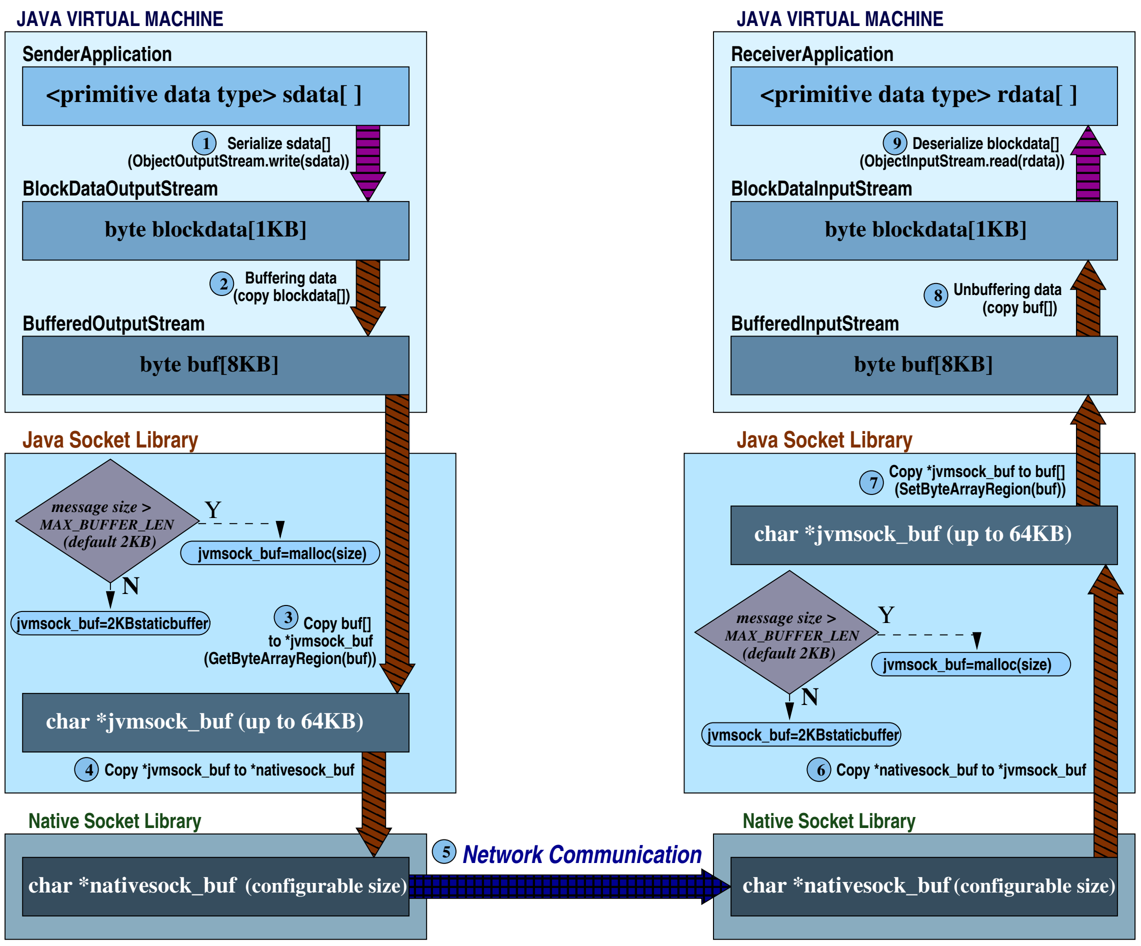

JAVA VIRTUAL MACHINE
SenderApplication
<primitive data type> sdata[ ]
1 Serialize sdata[] (ObjectOutputStream.write(sdata))
BlockDataOutputStream
byte blockdata[1KB]
2 Buffering data (copy blockdata[])
BufferedOutputStream
byte buf[8KB]
Java Socket Library
message size > MAX_BUFFER_LEN (default 2KB)
Y
N
jvmsock_buf=malloc(size)
jvmsock_buf=2KBstaticbuffer
3 Copy buf[] to *jvmsock_buf (GetByteArrayRegion(buf))
char *jvmsock_buf (up to 64KB)
4 Copy *jvmsock_buf to *nativesock_buf
Native Socket Library
char *nativesock_buf (configurable size)
5 Network Communication
JAVA VIRTUAL MACHINE
ReceiverApplication
<primitive data type> rdata[ ]
9 Deserialize blockdata[] (ObjectInputStream.read(rdata))
BlockDataInputStream
byte blockdata[1KB]
8 Unbuffering data (copy buf[])
BufferedInputStream
byte buf[8KB]
Java Socket Library
7 Copy *jvmsock_buf to buf[] (SetByteArrayRegion(buf))
char *jvmsock_buf (up to 64KB)
message size > MAX_BUFFER_LEN (default 2KB)
Y
N
jvmsock_buf=malloc(size)
jvmsock_buf=2KBstaticbuffer
6 Copy *nativesock_buf to *jvmsock_buf
Native Socket Library
char *nativesock_buf (configurable size)

Figure 3.1: Sun JVM socket operation

3.1.3. Efficient Shared Memory Socket Communication

The emergence of multi-core architectures has increased the use of shared memory socket communication, the most efficient way to exchange messages between two Java applications running on the same machine. However, JVM sockets handle intra-node transfers as TCP/IP transmissions. Some optimizations exist, like using a larger Maximum Transmission Unit (MTU) size, usually an order of magnitude higher, in order to reduce IP packet fragmentation, but TCP/IP overhead is still the throughput bottleneck. In order to reduce this performance penalty JFS has implemented shared memory transfers resorting to UNIX sockets (or similar lightweight non-TCP/IP sockets when available) and direct memory transfers, and

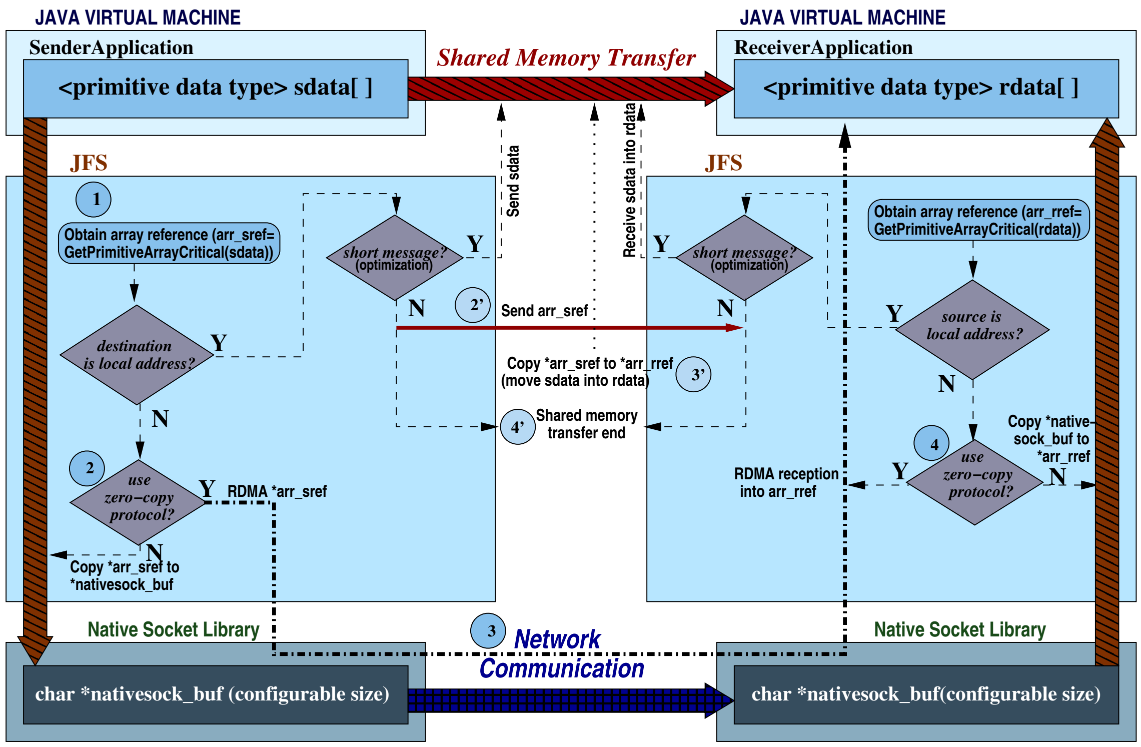

JAVA VIRTUAL MACHINE
SenderApplication
<primitive data type> sdata[ ]
Shared Memory Transfer
JAVA VIRTUAL MACHINE
ReceiverApplication
<primitive data type> rdata[ ]
JFS
1
Obtain array reference (arr_sref= GetPrimitiveArrayCritical(sdata))
destination is local address?
Y
N
2
use zero−copy protocol?
Y
RDMA *arr_sref
N
Copy *arr_sref to *nativesock_buf
short message? (optimization)
Y
N
Send sdata
2'
Send arr_sref
Receive sdata into rdata
Copy *arr_sref to *arr_rref (move sdata into rdata)
3'
4'
Shared memory transfer end
JFS
short message? (optimization)
Y
N
Obtain array reference (arr_rref= GetPrimitiveArrayCritical(rdata))
source is local address?
Y
N
4
use zero−copy protocol?
Y
N
Copy *native-sock_buf to *arr_rref
RDMA reception into arr_rref
Native Socket Library
3
Network Communication
Native Socket Library
char *nativesock_buf (configurable size)
char *nativesock_buf(configurable size)

Figure 3.2: JFS optimized protocol

therefore avoiding TCP/IP (see Figure 3.2). Thus, JFS first sends the `sdata` direct pointer (`arr_sref`) to the receiver, which then moves `sdata` content into `rdata` array through a native copy (`memcpy` or analogous). Finally, the sender polls for the copy end notification, a control message or a flag setting by the receiver. JFS greatly benefits from this optimization achieving memory-to-memory bandwidth, although for short messages the start-up latency of this three-step protocol can be enhanced by sending the data in only one transaction. This efficient shared memory support, together with optimized inter-node transfers, allows socket-based parallel applications to achieve good performance on multi-core clusters. This is due to the combination of the scalability provided by the distributed memory paradigm and the high performance of the shared memory communication.

JFS provides efficient socket communication through an optimized protocol. However, the usefulness of these improvements depends on the range of potential target systems and applications. Thus, in order to extend this range, JFS adds efficient support for high-speed cluster interconnects (presented next in Subsection 3.1.4). JFS also provides application transparency, in order to be used by Java applications without source code modification, as will be shown in Subsection 3.1.5.

3.1.4. Efficient Java Sockets on High-speed Networks

JFS includes a high-speed cluster network support much more efficient than the use of IP emulations. Thus, JFS relies on native socket operation that does not experience problems with the JVM. An example is the avoidance of IPv6, preferred by JVM sockets and usually not implemented for high-speed networks. This high-speed interconnect support is implemented specifically for each network through JNI, which provides native socket throughput to Java. JNI is also used by JVM sockets, although their generic access to the network layer is inefficient for high-speed networks as they do not take advantage of the underlying native libraries.

Figure 3.3 represents a schema of the components involved in socket operation on high-speed networks. From bottom to top, the first layer is the Network Interface Card (NIC) for each high-speed network, on top of this layer is the NIC driver (or shared memory protocol), next the TCP/IP emulations and the sockets implementations, next the Java IO libraries (both JFS and JVM IO sockets), on top of the sockets the Java communication middleware, and finally the parallel and distributed applications. Java applications access Java sockets usually through Java communication middleware, such as MPJ libraries, typically based either on RMI or directly on sockets.

With respect to shared memory and Gigabit Ethernet communications, the Java support is directly on the native sockets libraries. The SCI low-level drivers are IRM (Interconnect Resource Manager) and SISCI (Software Infrastructure for SCI), whereas SCILib is a communication protocol on top of SISCI that offers unidirectional message queues. On SCI JFS resorts to SCI Sockets and SCILib, higher level solutions than IRM and SISCI but still efficient libraries. On Myrinet and InfiniBand JFS relies on Sockets-MX and Socket Direct Protocol (SDP), respectively, for providing Java applications with efficient communication. Moreover, JFS also provides JVM sockets with high-speed network support in order to avoid IP emulations.

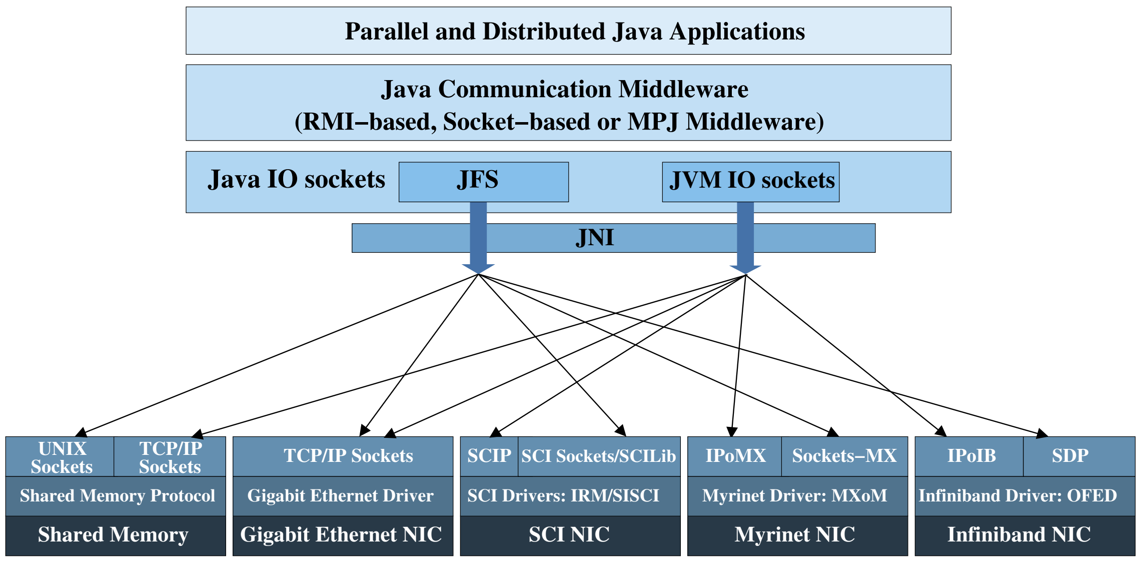


Figure 3.3: Java communication middleware on high-speed multi-core clusters

Additionally, JFS aims to transparently obtain the highest performance on systems with several communication channels through a failover approach. Thus, JFS first attempts to use the option with the highest performance. If this fails, it follows, in descending order of performance, with the remaining communication channels that are available.

3.1.5. JFS Application Transparency

By implementing the socket API, a wide range of parallel and distributed target applications can take advantage transparently of the efficient JFS communication protocol. As Java has a built-in procedure (setting factories) to swap the default socket library, it is easy to replace the JVM sockets by JFS. However, the JVM socket design has to be followed in order to implement a swappable socket library. Figure 3.4 presents JFS core classes: `PlainSocketImpl` is the Sun JVM socket implementation, `FastSocketImplFactory` creates custom JFS sockets, and the I/O stream classes, whose package is `java.net` for Sun JVM sockets and `jfs.net` for JFS. The stream classes are in charge of managing the transport protocol. The JFS setting as the default socket library is shown in Listing 3.3. From then on the appli-

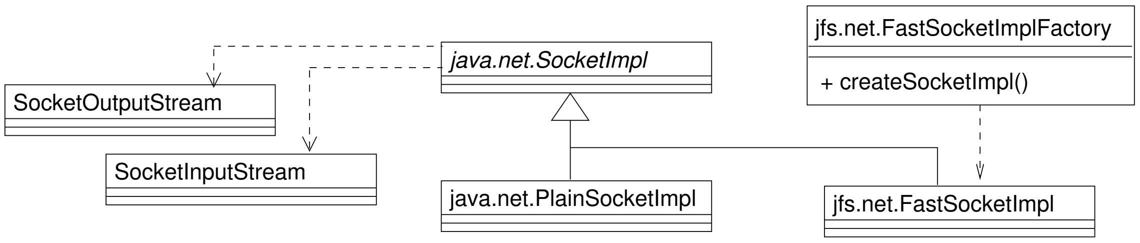


Figure 3.4: JFS core class diagram

Listing 3.3: Swapping Java socket implementation

```
SocketImplFactory factory = new jfs.net.FastSocketImplFactory();
Socket.setSocketImplFactory(factory);
ServerSocket.setSocketFactory(factory);
```

Listing 3.4: JFS *launcher* application code

```
[Swap Java socket implementation]

Class cl = Class.forName(className);
Method method = cl.getMethod("main", parameterTypes);
method.invoke(null, parameters);
```

cation will use this implementation. As this procedure requires source code modification, Java's reflection has been used in order to obtain a transparent solution. Thus, a small application *launcher* swaps its default socket factory and then invokes the `main` method of the target class (see Listing 3.4). The target application will use JFS transparently even without source code availability.

JFS extends the socket API by adding methods that avoid serialization and eliminate unnecessary copies when sending portions of primitive data type arrays. Listing 3.5 presents an example of this feature. As JVM sockets can not send array portions (except for parts of byte arrays) a new array must be created to store the data to be serialized and then sent. This costly process is repeated at the receiver side. Listing 3.5 shows the handling of this communication scenario in a portable way in order to use the efficient JFS methods when they are available. This feature is of special interest in communication middleware such as Java message-passing libraries and RMI, yielding significant benefits to end applications without modifying their source code.

Listing 3.5: JFS direct send of part of an integer array

```
if (os instanceof jfs.net.SocketOutputStream) {
        jfsExtendedAPI = true;
        jfsos = (jfs.net.SocketOutputStream) os;
}
oos = new ObjectOutputStream(os);
int int_array[] = new int[20];

[...]

// Writing the first ten elements of int_array
if (jfsExtendedAPI)
        jfsos.write(int_array,0,10);
else {
        int[] ints = (int[])Array.newInstance(int.class, 10);
        System.arraycopy(int_array, 0, ints, 0, 10);
        oos.writeUnshared(ints);
}
```

Parallel and distributed Java applications and, especially, communication middleware can benefit transparently from the higher performance of JFS on high-speed networks and shared memory communication. This can be achieved without losing portability, using particular JFS features such as serialization avoidance only when this socket library is available. Moreover, this solution is interoperable as it can communicate with JVM sockets, although relying only on features shared by both implementations. Thus, the buffering and copying reduction could be used, but not the high-speed network support nor the optimized shared memory transfers. The subsequent sections evaluate JFS performance (Section 3.2) and its impact on representative kernels and applications (Section 3.3).

3.2. JFS Performance Evaluation

3.2.1. Experimental Configuration

The testbed used for the performance evaluation of JFS is different from the two high-speed clusters used in Section 2.3. Thus, a cluster of eight dual-processor nodes (Pentium IV Xeon 5060 dual-core at 3.2 GHz, 4 GB of memory) interconnected via SCI, Myrinet and Gigabit Ethernet has been used. The SCI NIC is a D334 card and the Myrinet NIC is an "F" Myrinet 2000 card (M3F-PCIXF-2 Myrinet-Fiber/PCI-X NIC). Both are plugged into 64-bit 66 MHz PCI slots. The Gigabit Ethernet

NIC is an Intel PRO/1000 using a 3Com 2816-SFP Plus switch. A DGS-1216T Dlink switch for evaluating Gigabit Ethernet Jumbo Frames performance has also been used. The main differences between this cluster and the SCI cluster used in Section 2.3 are the processor, a more recent dual-core Pentium IV Xeon at 3.2 GHz instead of a Pentium IV Xeon at 1.8 GHz, and the amount of memory, 4 GB instead of 1 GB. This cluster has been used for the SCI, Myrinet, Gigabit Ethernet, and shared memory performance evaluation. Additionally, for InfiniBand benchmarking a cluster that consists of 8 DELL SC430 nodes (Intel Pentium IV Prescott at 2.8 GHz, 1 GB of memory) interconnected via Mellanox MT25204 InfiniBand HCAs (Single Port, 20 Gbps) through a HUAWEI Quidway S1224 switch has been used.

The OS of both clusters is Linux RedHat 4 (CentOS 4.4 and AS for the first and the second cluster, respectively) with Linux kernel 2.6.9-42.ELsmp and C compiler gcc 3.4.6. The JVM used is Sun JDK 1.5.0_07 as it obtains slightly better performance than IBM JDK 1.5 for the benchmarks used in Sections 3.2 and 3.3. The SCI libraries are SCI Sockets 3.1.4, DIS 3.1.11 (it includes IRM, SISCI and SCILib) and SCIP 1.2.0; the Myrinet libraries are MX 1.1.1 and Sockets-MX 1.1.0, whereas the InfiniBand libraries are Sockets Direct Protocol (SDP), IPoIB (the IP emulation over InfiniBand) and the Open Fabrics Enterprise Distribution (OFED) drivers 1.2 (see Figure 3.3).

In order to micro-benchmark JFS performance, a Java socket version of NetPIPE [102] has been developed. The results considered in this section are half of the round trip time of a ping-pong test running JIT compiled bytecode. In order to obtain optimized JIT results, 10,000 warm-up iterations were executed before the actual measurements. The performance of byte, integer and double arrays have been benchmarked, as they are data structures frequently used in parallel and distributed applications. For purposes of clarity the JNI array notation has been used. Thus, `B]` denotes a byte array, `I]` an integer array and `D]` a double array. When using serialization, it has been pointed out the procedure through the use of the keys `OOS` and `OBOS`. `OOS` indicates a `java.net.ObjectOutputStream` object wrapping a `SocketOutputStream` object, whereas `OBOS` is a `java.net.ObjectOutputStream` object wrapping a `BufferedOutputStream` around the supplied `SocketOutputStream`. `OOS` writes the serialized data directly to the stream in order to reduce the start-up latency, whereas `OBOS` buffers the serialized data in a byte array (by default an 8 KB buffer) in order to minimize the stream accesses and thus increase bandwidth. In fact, `OBOS` usually outperforms `OOS` results, especially for long messages, as shown for Gigabit Ethernet in Figures 3.11– 3.12. `OBOS` also obtains, in general, better performance than `OOS` on the remaining interconnects and shared memory, and thus for clarity purposes only the best results (`OBOS`) are shown (see Figures 3.5–3.10 and 3.13–3.14 for SCI, Myrinet, InfiniBand and shared memory performance).

3.2.2. JFS Micro-benchmarking on High-speed Networks

Figures 3.5-3.12 show the latencies and bandwidths of native and Java socket libraries as a function of the message size, for byte, integer and double arrays on SCI, Myrinet, InfiniBand and Gigabit Ethernet. The native libraries considered are SCI Sockets, Sockets-MX, SDP and the native TCP/IP sockets, whereas the Java sockets libraries are Sun JVM sockets and JFS. The latency graphs (at the top) serve to compare short message performance, whereas the bandwidth graphs (bottom) are useful to compare long message performance.

Figures 3.5 and 3.6 present latency and bandwidth results on SCI. The two available transport layers with Java support, the IP emulation SCIP and JFS, obtain significantly different results. Thus, JFS start-up latency is 6μs compared to 36-48μs for SCIP, showing an overhead reduction of up to 88%. Regarding bandwidth, JFS achieves up to 2366 Mbps whereas SCIP results are below 450 Mbps, up to 1305% performance increase for JFS (`B]JFS` vs. `OBOS(D])` for a 2 MB message). `B]`, `I]` and `D]` JFS results are quite similar among them as they use the same protocol, a direct send avoiding serialization. Thus, for clarity purposes, only `B]` values are presented as the representative results under the label `B],I],D] JFS`. As JFS is implemented on top of SCI Sockets (see Figure 3.3), its processing overhead (the difference between JFS and SCI Sockets performance) can be estimated in around 1-2μs for short messages, and approximately 5% bandwidth penalty for long messages. Therefore, JFS obtains quite similar results to SCI Sockets, the high performance native socket library on SCI. `OBOS` serialization imposes overheads on start-up latencies around 5-6μs and 10-12μs using JFS and SCIP, respectively (JFS start-up is 6μs, `OBOS` start-up over JFS is 11-12μs, Sun JVM sockets for `B]` on SCIP is 36μs, whereas `OBOS` Sun JVM sockets start-up is 46-48μs). `OBOS` over SCIP bandwidths are quite poor, under 400 Mbps. With respect to `OBOS` over JFS results,

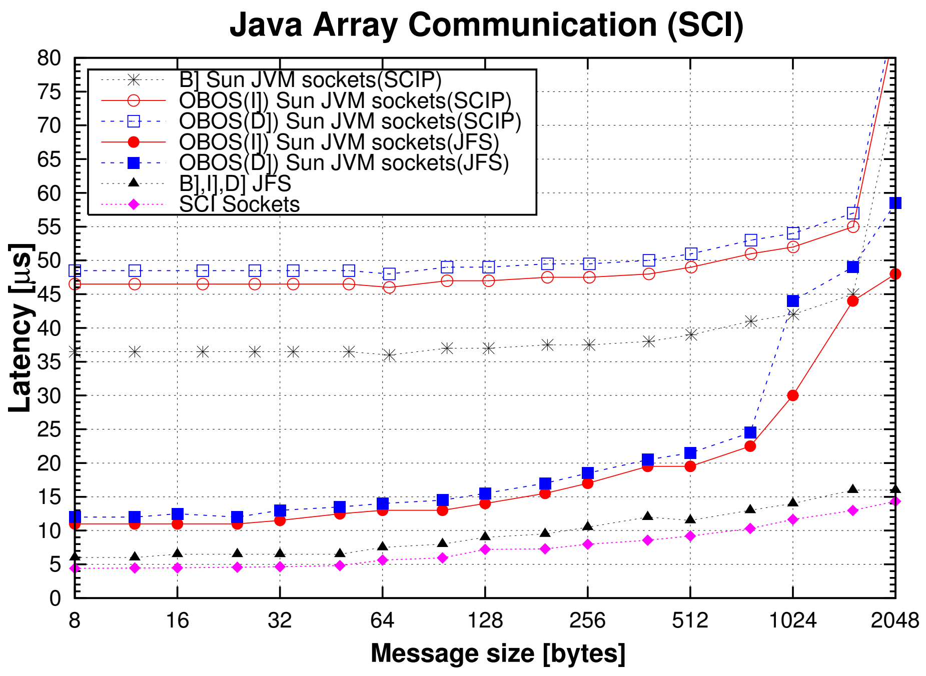


Figure 3.5: Java array communication latency on SCI

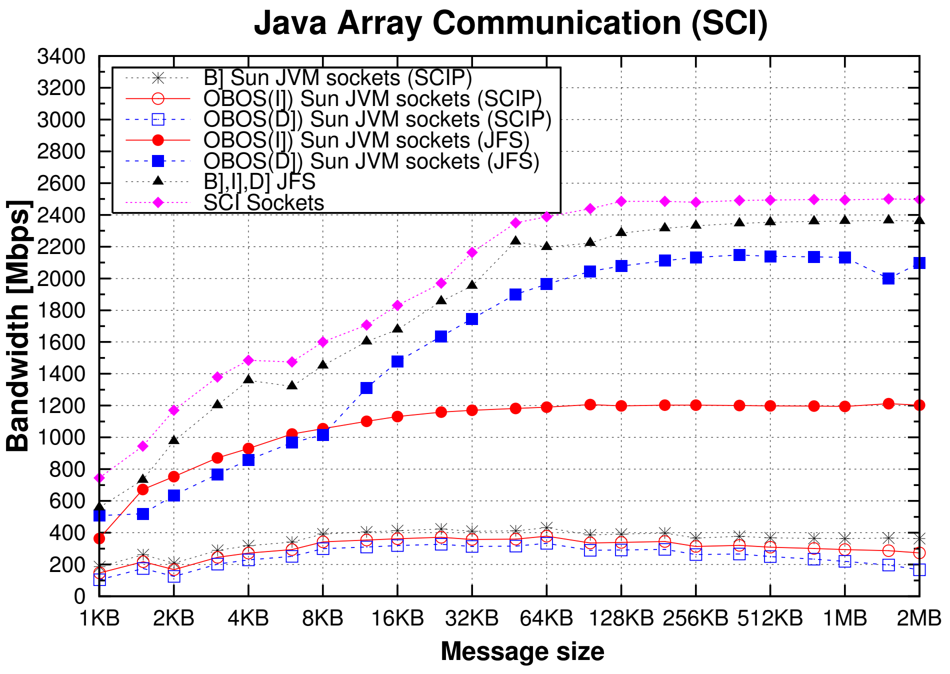


Figure 3.6: Java array communication bandwidth on SCI

OBOS(D]) bandwidth is close to JFS (around 90%) thanks to its optimized native implementation. Sun JVM provides optimized native methods for float and double array serialization and a pure Java method for integer array serialization. Thus, OBOS(I]) over JFS only obtains a 50% (around 1200 Mbps) of JFS performance for long messages. However, for short messages, OBOS(I]) obtains better performance than OBOS(D]).

Figures 3.7 and 3.8 present latency and bandwidth results on Myrinet. The best Java sockets results have been obtained using JFS as transport layer, although using the IP emulation IPoMX the differences narrow as the message size increases, showing similar long message bandwidth for byte arrays. The reason for this behavior is the higher start-up latency of IPoMX (22/32μs for byte arrays and serialized data, respectively) compared to JFS (7μs, up to 78% less than IPoMX), and that the Myrinet NIC is the communication bottleneck limiting the maximum transfer rate to 2 Gbps. In fact, the experimentally measured JFS and IPoMX bandwidths can only increase up to 85% of this value (1700 Mbps). JFS Myrinet support is based on Sockets-MX rather than Sockets-GM for its better performance. This has been experimentally assessed on our testbed, where JFS resorting to Sockets-GM obtained higher start-up latency (23μs) and lower bandwidths than using Sockets-MX. The presented Sockets-MX results show that JFS overhead on Myrinet is quite reduced, obtaining almost native performance. OBOS serialization imposes an overhead on start-up latency of around 7-10μs. Regarding bandwidth, OBOS over JFS performs better than using IPoMX. The native serialization method in OBOS(D]) (serialization implemented in native code) improves the performance of the pure Java serialization method used in OBOS(I]) (100% Java implementation) only over JFS, but not over IPoMX. However, B],I],D] JFS clearly outperforms OBOS results with a performance increase of up to 412%.

Figures 3.9 and 3.10 present latency and bandwidth results on InfiniBand, where both SDP and IPoIB have been used as underlying protocols. JFS has obtained the best results, especially for long messages and when serialization is needed (i.e. sending integer and double arrays). In this latter case, JFS throughput is up to 860% higher than JVM sockets over IPoIB. This peak result (860%) has been obtained with a 256 KB message, for which JFS obtains 6.7 Gbps and JVM sockets over IPoIB 0.78 Gbps. Regarding start-up latency, JFS reduces significantly, up to 65%,

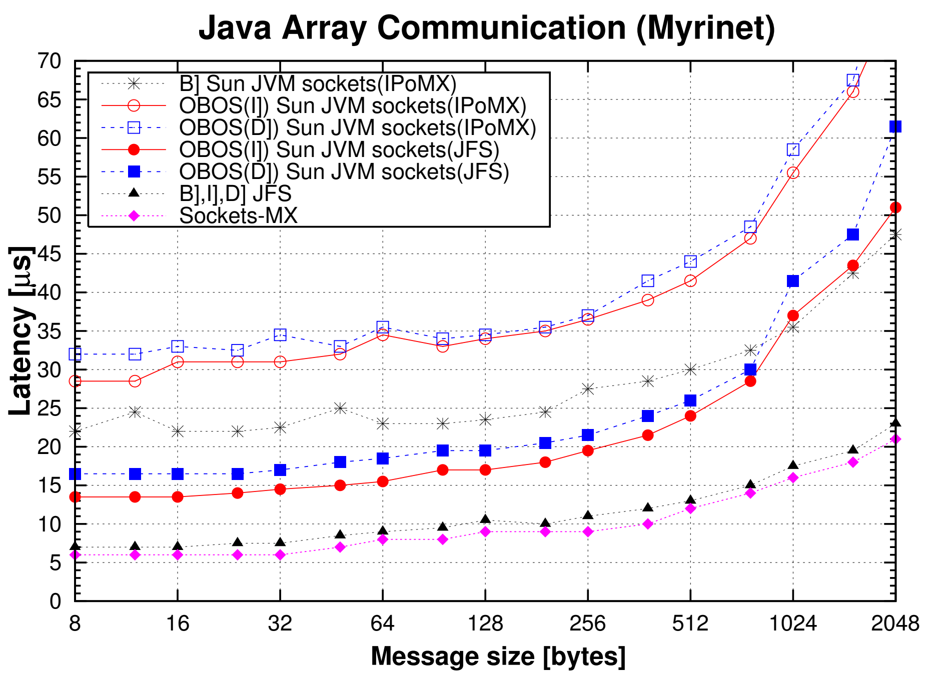


Figure 3.7: Java array communication latency on Myrinet

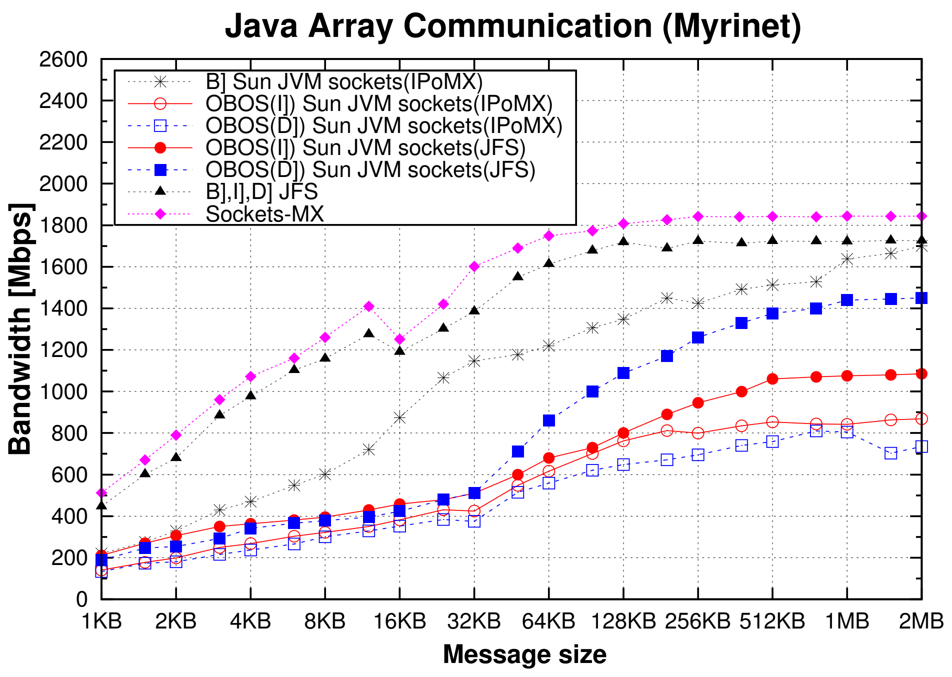


Figure 3.8: Java array communication bandwidth on Myrinet

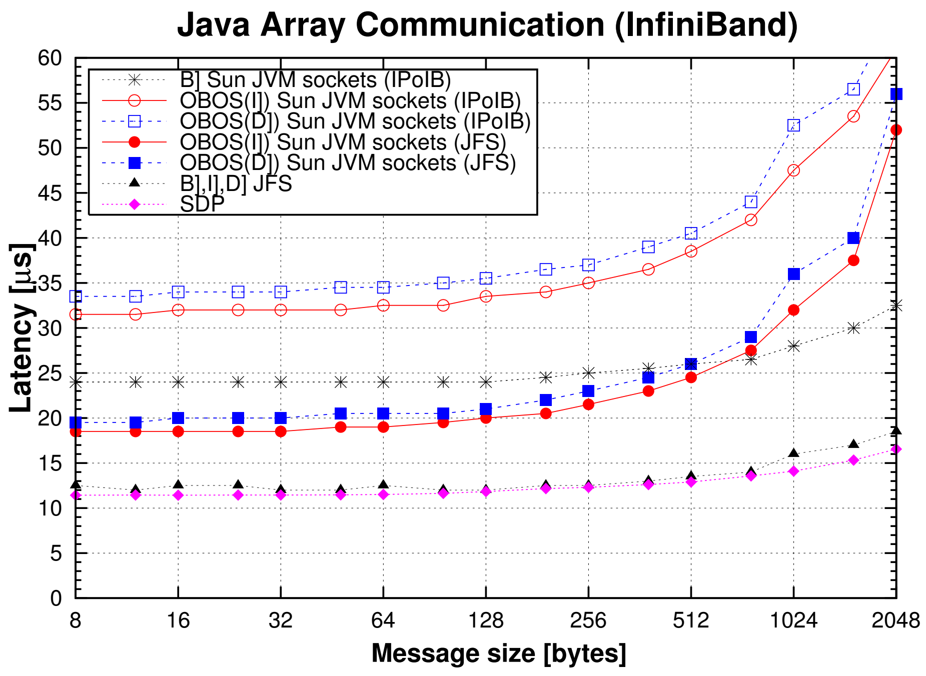


Figure 3.9: Java array communication latency on InfiniBand

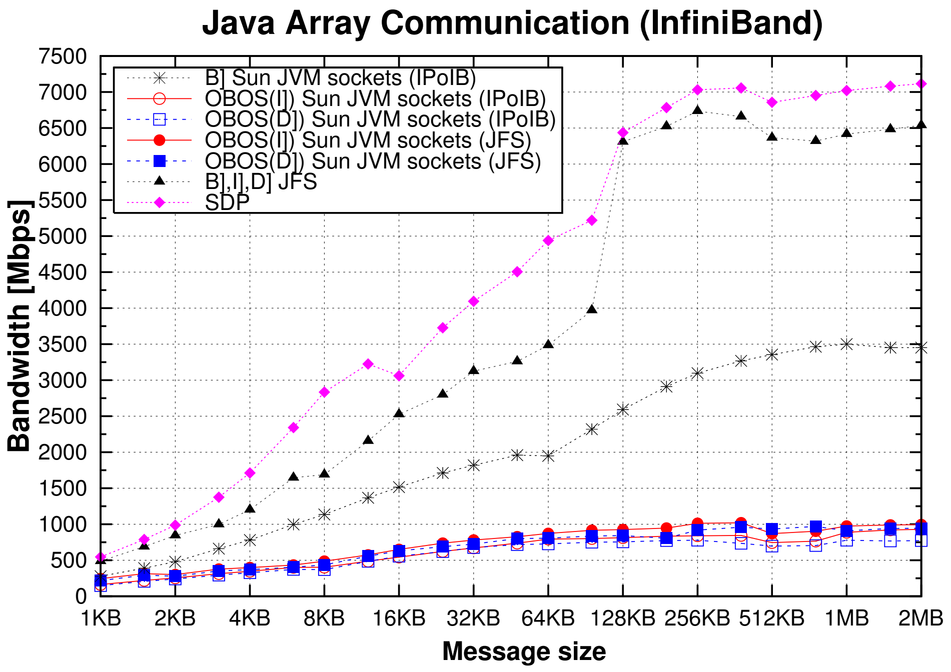


Figure 3.10: Java array communication bandwidth on InfiniBand

the overhead of sending serialized data with JVM sockets over IPoIB, from $34\mu s$ down to $12\mu s$. JVM sockets over JFS show relatively low start-up latencies (around $19\mu s$) but poor long message performance, with less than 1 Gbps bandwidth sending serialized data. However, JVM sockets over IPoIB present the worst performance sending serialized data, with results below 0.8 Gbps. This small gap in serialized data performance between the use of JFS and IPoIB as underlying layer for JVM sockets (around 0.2 Gbps) is due to the fact that the serialization is the main performance bottleneck, and not the network communication. Whereas on SCI and Myrinet the main performance bottleneck is the NIC, with 5.33 and 2 Gbps of theoretical bandwidth, respectively, on the InfiniBand testbed, whose theoretical bandwidth is 16 Gbps, the network is not the main performance bottleneck. Thus, as shown, the serialization imposes an important overhead on JVM sockets sending serialized data, whereas JFS and SDP are limited by the PCI Express x4 bus (8 Gbps of theoretical unidirectional bandwidth limit) in which the InfiniBand card is plugged. In this scenario, with a higher theoretical bandwidth limit than on the SCI and Myrinet testbed, the IP emulation layer over InfiniBand (IPoIB) obtains better performance for `B]` than IPoMX on Myrinet and SCIP on SCI.

Figures 3.11 and 3.12 present latency and bandwidth results on Gigabit Ethernet. There is not a significant difference in byte array performance between socket implementations, although JFS slightly outperforms Sun JVM sockets for medium-size messages. However, JFS performance improvement of sending integer and double arrays (`I]` and `D]`) is up to 119%, result obtained for a 2 MB message, thanks to avoiding serialization. It can be seen that the serialization imposes an overhead of around $4\mu s$ in start-up latency for `OOS(I])/OOS(D])`, and of around $7\mu s$ for `OBOS(I])/OBOS(D])` due to the additional buffering overhead. The native serialization method used in `OBOS(D])` only outperforms the pure Java method `OBOS(I])` for long messages. Without buffering, i.e. using `OOS(I])/OOS(D])`, Java does not take advantage of the use of the native method in `OOS(D])`. However, the Ethernet protocol is the main performance bottleneck as it imposes high start-up latencies, around $50\mu s$, and low bandwidths, below the 1 Gbps maximum network transfer rate, severely limiting throughput improvement. This analysis is confirmed by the presented native TCP/IP sockets results, which show similar performance to Java sockets implementations, due to the low processing overhead of the sockets layer.

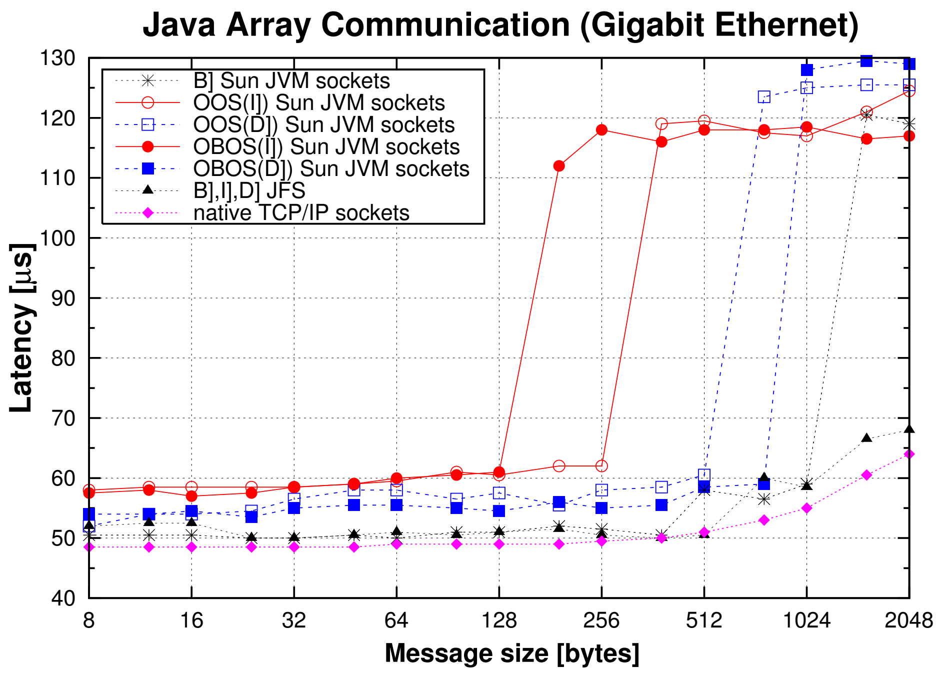


Figure 3.11: Java array communication latency on Gigabit Ethernet

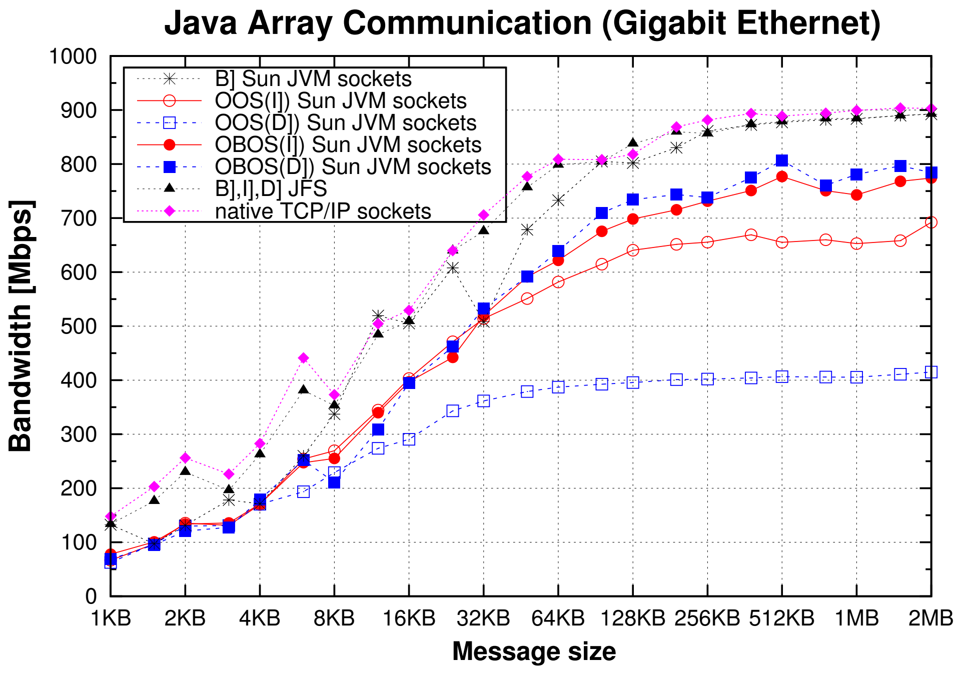


Figure 3.12: Java array communication bandwidth on Gigabit Ethernet

Additionally, socket latencies are clustered around 50 and 120μs (see Figure 3.11) caused by the operation of the Gigabit Ethernet driver on our testbed, which only notifies the data reception to the sockets library with some latency intervals (e.g., around [$45 - 50\mu s$] and [$110 - 120\mu s$]). The effect of these clustered latencies can also be observed for JFS in Figure 3.12 where the bandwidth for [1 KB − 16 KB] messages presents a saw-tooth shape. Looking for potential improvements in order to partly overcome these limitations the use of Gigabit Ethernet Jumbo Frames has been evaluated.

3.2.3. JFS on Gigabit Ethernet Jumbo Frames

The Ethernet default Maximum Transmission Unit (MTU) of 1500 bytes has been maintained for backward compatibility in order to handle any communication between 10/100/1000 Mbps devices without any Ethernet frame fragmentation or reassembly. Nevertheless, this is a rather small size that increases CPU load due to handling numerous frames when sending long messages. A larger MTU reduces CPU overhead and therefore increases long message bandwidth, although for medium-size messages waiting for filling larger Ethernet frames increases latency. Jumbo Frames is the technology that extends MTU size up to 9000 bytes.

The use of JFS with Jumbo Frames for MTU sizes of 3000, 4500, 6000 and 9000 bytes has been evaluated. Jumbo Frames increase slightly JFS long message performance. For a 2 MB message the bandwidth rises from 892 Mbps, with the default MTU, up to 932 Mbps using an MTU of 9000 bytes. This improved result is 93% of the maximum theoretical bandwidth, 4% more than using the default MTU. Regarding medium-size messages, the use of Jumbo Frames increases JFS latency in the range [1.5 KB − 256 KB] up to 90% (this peak latency increase was obtained for a 6 KB message with an MTU of 9000 bytes). This latency increase is especially high for [1.5 KB − 16 KB] messages, while for larger messages the negative impact of Jumbo Frames is reduced as the message size increases.

An additional characteristic of the use of Jumbo Frames is the CPU communication processing offloading. Table 3.1 presents the CPU overhead of two Java socket implementations in terms of percentage of CPU load (using a Xeon 3.2 GHz) devoted to socket communication processing. The NetPIPE benchmark sending from 9 KB up to 2 MB messages (range with Ethernet frame fragmentation) has been used for measuring these values. It can be seen that Jumbo Frames reduce significantly CPU overhead. Nevertheless, as Jumbo Frames trade off medium-size message performance for CPU offloading, this is not an especially useful feature.

A general conclusion can be made that the use of Jumbo Frames is recommended for applications sending only long messages. Regarding the CPU offloading, Jumbo Frames contribution is not especially important as JFS already reduces CPU load avoiding unnecessary buffering and extra copies. Thus, the JFS CPU load without using Jumbo Frames (i.e., MTU=1500 bytes) is 60% lower than using Sun JVM sockets (from 30% to 12%), as can be seen in Table 3.1, without trading off performance for CPU offloading.

Table 3.1: CPU load percentage of sockets processing using Gigabit Ethernet Jumbo Frames

		MTU (bytes)				
		1500	3000	4500	6000	9000
Socket Implementation	Sun JVM sockets	30%	25%	15%	16%	5%
	JFS	12%	10%	4%	4%	3%

3.2.4. Java Shared Memory Communication

Figure 3.13 presents the performance of the JFS shared memory protocol (see Subsection 3.1.3) for short messages. Although the default underlying library for this protocol is UNIX sockets, JFS performance using TCP sockets is shown for comparison purposes. JFS start-up latency is $8\mu s$, half of Sun JVM sockets start-up. However, this value is larger than the 6 and $7\mu s$ JFS start-up latencies on SCI and Myrinet, respectively (see Subsection 3.2.2), as the underlying native library, UNIX sockets, imposes higher start-up overhead than the native sockets on these high-speed networks. In a multi-core scenario it is key to reduce the high start-up latency of shared memory native communication. The optimization of UNIX sockets implementation or the development of a high performance sockets library on shared memory could reduce shared memory start-up overhead.

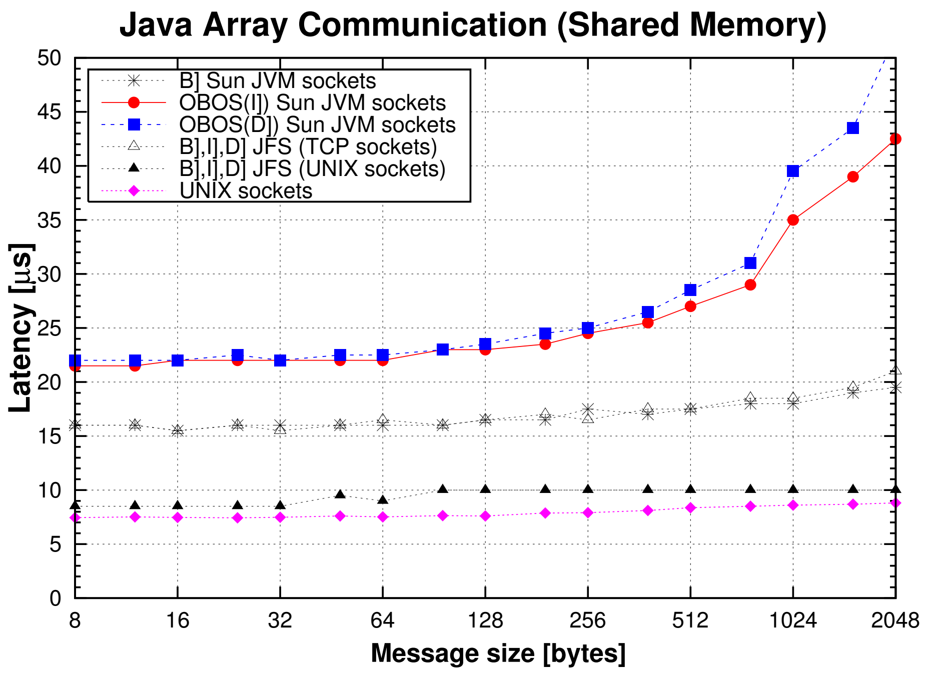


Figure 3.13: Java array communication latency on shared memory

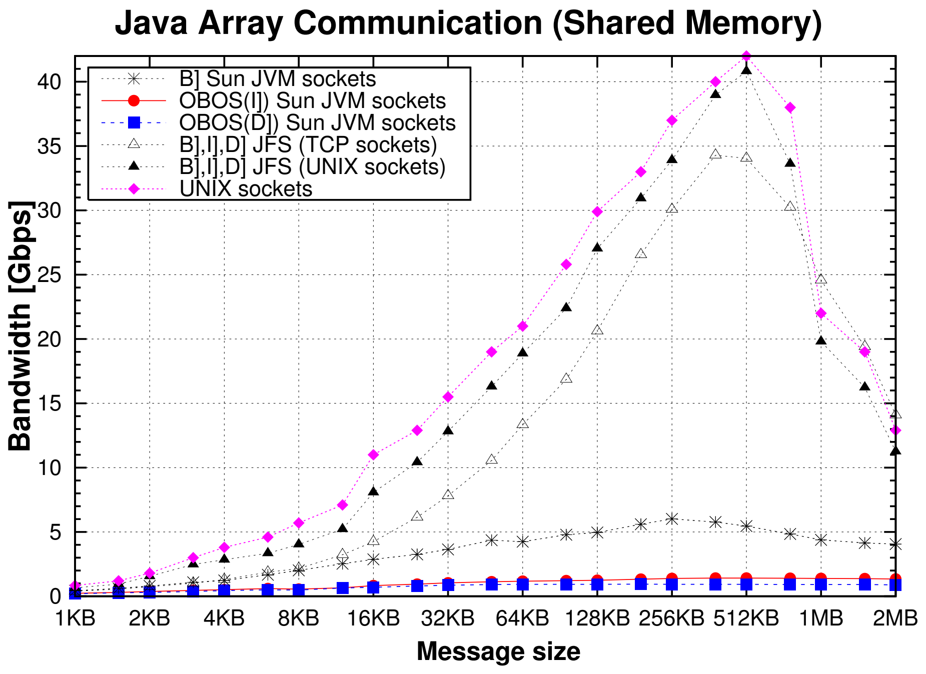


Figure 3.14: Java array communication bandwidth on shared memory

Figure 3.14 shows the significant bandwidth increase of JFS communication due to the use of the optimized shared memory protocol for messages longer than 16 KB. This protocol increases the peak bandwidth from 9 Gbps, using JFS without this optimized protocol, up to 34 and 41 Gbps for JFS using TCP and UNIX sockets, respectively. These peak bandwidths are obtained for [256 KB − 768 KB] message sizes, as memory-to-memory transfers also obtain their peak bandwidths for this range. The messages within this range have the whole data in cache (the L2 cache in the testbed is 2MB and the sockets libraries use around 1 MB), thus significantly increasing sockets performance. The cache invalidation technique has not been used as it does not reflect the usual situation in a production environment. The performance of the optimized shared memory protocol has also been measured for the native UNIX sockets library, showing that JFS also obtains almost native performance on shared memory. Sun JVM sockets performance is very poor, under 1.5 Gbps for integer and double arrays and under 6 Gbps for byte arrays. The observed bandwidth increase is up to 4411%, peak value obtained by comparing a 512 KB `D]` message sent with `JFS (UNIX sockets)` vs. sent with `OBOS(D])` Sun JVM sockets.

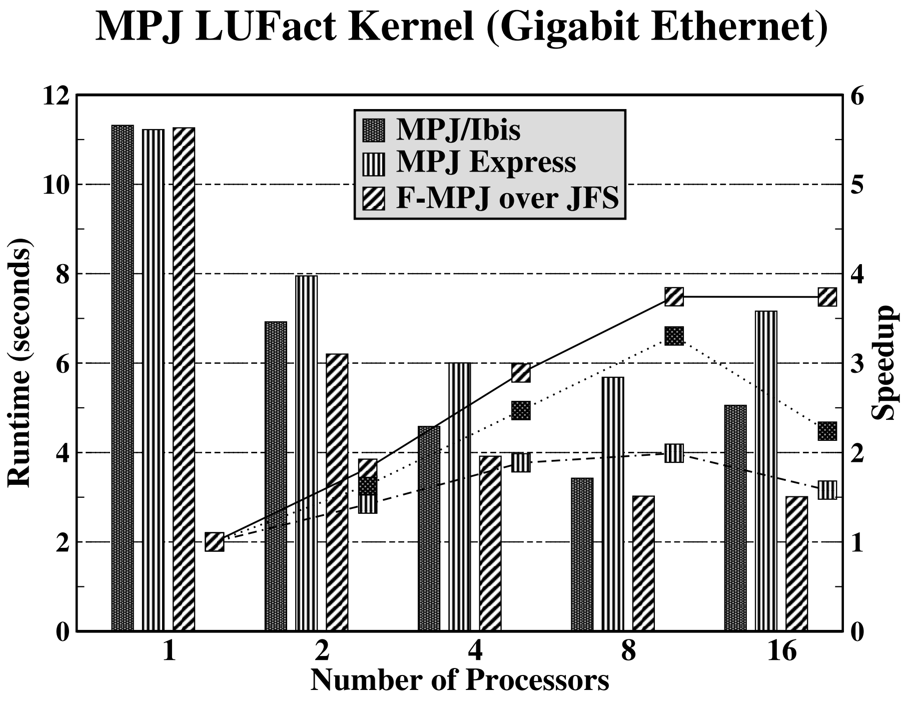
3.3. Performance Impact on Parallel Applications

JFS micro-benchmarking has shown significant performance improvement, but its usefulness depends on the impact on the overall application performance. The range of JFS applicability covers socket-based MPJ applications and MPJ libraries such as MPJ Express [81] and MPJ/Ibis [16], RMI applications and RMI-based middleware (see two upper layers in Figure 3.3). In short, any socket-based parallel or distributed Java application running on a cluster can use JFS. These applications can benefit immediately from JFS thanks to its user and application transparency. The impact of JFS on the overall message-passing application performance has been analyzed in the current section where two MPJ benchmarks, LUFact and Moldyn, from the Java Grande Forum (JGF) Benchmark Suite [17] have been selected for evaluation. Additionally, the impact of a JFS-based RMI protocol optimization on RMI applications will be presented in Section 4.5.

The testbed used for JFS-based message-passing evaluation has been the Xeon 5060 cluster with two interconnects: Gigabit Ethernet due to its wide deployment, and SCI as the JFS micro-benchmarking has achieved the best performance on this cluster using this network (see Subsections 3.2.1 and 3.2.2). In order to isolate the impact of these networks on performance, only one processor per node has been used for running the benchmarks on up to 8 processors. Furthermore, two processors per node have been used for obtaining 16-processor results in order to analyze the behavior of hybrid high-speed network/shared memory (inter-node/intra-node) communication. As the trend is to move to multi-core clusters with high-speed networks, the performance of this hybrid approach is of special interest.

Two message-passing benchmarks, LUFact, a matrix LU factorization kernel, and Moldyn, a molecular dynamics N-body parallel simulation, have been selected (specifically using their size C workloads) in order to analyze the performance impact of the use of JFS-based Message-Passing in Java (MPJ) middleware. These benchmarks have been run using three MPJ libraries: MPJ/Ibis, MPJ Express and Fast MPJ (F-MPJ), our JFS-based MPJ implementation (further explained in Chapter 5). On SCI, JFS has been used instead of JVM sockets over SCIP as underlying layer for MPJ/Ibis and MPJ Express (see Figure 3.3) in order to avoid the IP emulation and thus ensure a fair comparison. Therefore, the three MPJ libraries use the same underlying socket library on SCI and the performance differences are exclusively due to their implementation. Thus, the benefits of the JFS-based F-MPJ implementation can be easily noticed. In fact, only F-MPJ is labeled "over JFS" in the key of Figures 3.15 and 3.16, as although MPJ/Ibis and MPJ Express use JFS on SCI, they do not fully support JFS features, especially the serialization avoidance (see Subsection 3.1.5).

Figure 3.15 shows MPJ LUFact runtimes and speedups. The performance differences on two, four and eight processors are explained exclusively by the performance of these MPJ libraries on high-speed networks. However, results on sixteen processors combine network communication (inter-node) with shared memory communication (intra-node). F-MPJ over JFS significantly outperforms MPJ/Ibis and MPJ Express, especially using sixteen processors and SCI, obtaining a speedup increase of up to 179%. Both MPJ/Ibis and MPJ Express scale performance on only up to eight processors, decreasing their speedups for sixteen processors. Nevertheless, F-MPJ over JFS obtains higher speedups on sixteen processors than on eight processors, although very slightly on Gigabit Ethernet, thanks to combining efficiently its inter-node and intra-node communication.



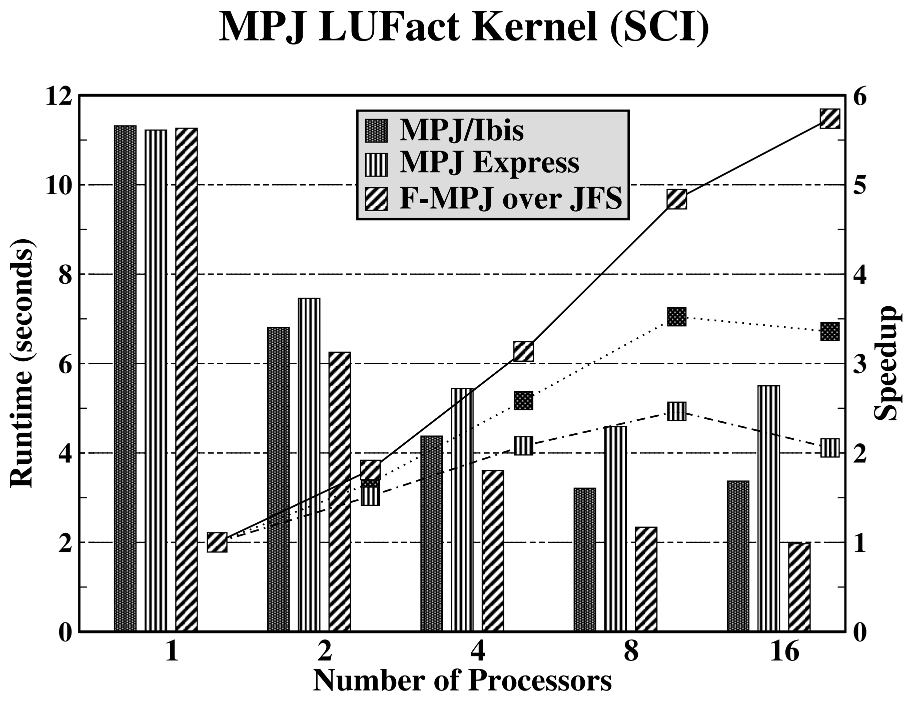


Figure 3.15: MPJ LUFact kernel performance on Gigabit Ethernet and SCI

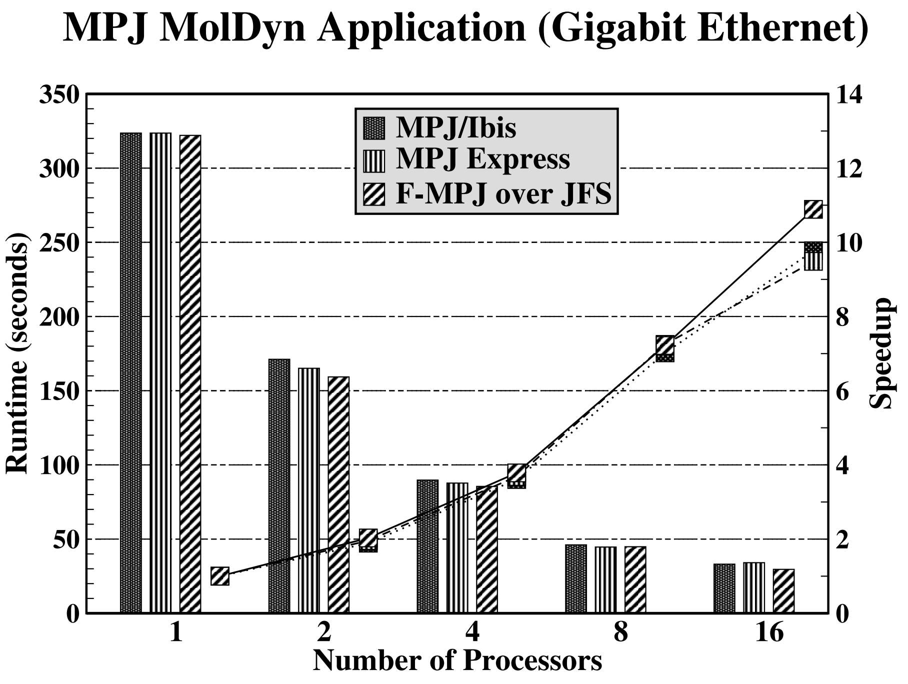
Figure 3.16 shows MolDyn runtimes and speedups. MolDyn is a more computation-intensive code than LUFact, obtaining almost linear speedups on up to eight processors. Nevertheless, for sixteen processors MPJ Express and MPJ/Ibis show significantly worse performance than F-MPJ over JFS, which outperforms these libraries up to 14% and 42% on Gigabit Ethernet and SCI, respectively. MPJ Express performs slightly better than MPJ/Ibis for this benchmark, except for sixteen processors. However, these differences are small due to the limited influence of communication overhead on the overall performance.

This analysis of MPJ libraries performance has also been useful for evaluating two additional Java sockets libraries: Java NIO and Ibis sockets (see Section 1.1). Thus, the differences observed among MPJ/Ibis, MPJ Express and the JFS-based MPJ are predominantly explained by the socket libraries used in their implementation, Ibis sockets, Java NIO and JFS, respectively. Although JFS has been used as underlying runtime layer for MPJ/Ibis and MPJ Express on SCI, it has not replaced the socket libraries used in their implementations, Ibis sockets and Java NIO, respectively, so this analysis is valid. Thus, Java NIO sockets obtain the lowest performance, as this implementation is more focused on providing scalability in distributed systems rather than efficient message-passing communication. Ibis sockets outperform Java NIO sockets, and are a good estimate for JVM sockets performance, according to previous evaluations [91]. Finally, JFS clearly achieves the highest performance, showing a significant impact on the overall performance of MPJ applications when using a JFS-based MPJ implementation (F-MPJ).

3.4. Chapter 3 Conclusions

This chapter has presented Java Fast Sockets (JFS), an efficient Java communication middleware for high-speed clusters. JFS implements the widely used socket API for a broad range of target applications. Furthermore, the use of standard Java compilers and JVMs, and its interoperability and transparency allow for immediate performance increase. Among its main contributions, JFS:

- Enables efficient communication on clusters interconnected via high-speed networks (SCI, Myrinet, InfiniBand) through a general and easily portable solution.



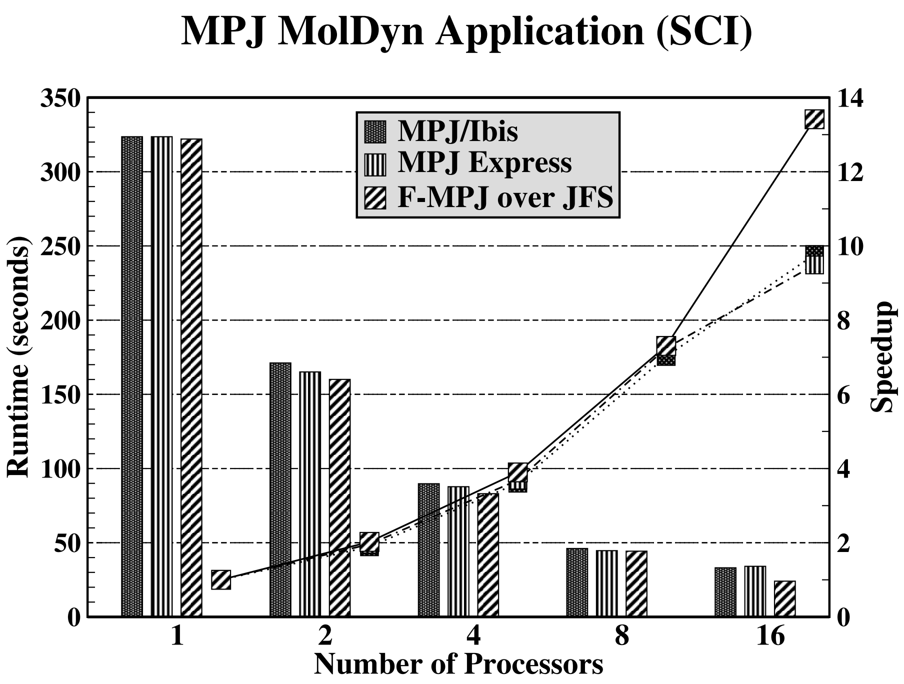


Figure 3.16: MPJ MolDyn application performance on Gigabit Ethernet and SCI

- Avoids the need of primitive data type array serialization.
- Reduces buffering and unnecessary copies.
- Optimizes shared memory (intra-node) communication.

A detailed performance evaluation of JFS has been conducted on SCI, Myrinet, InfiniBand and Gigabit Ethernet. Moreover, JFS performance on shared memory has been evaluated on a dual-core node. Table 3.2 summarizes the performance improvement obtained. JFS has also enhanced the performance of communication-intensive parallel applications obtaining speedup increases of up to 179% (LUFact benchmark on sixteen processors) compared to the analyzed socket-based Java message-passing libraries. However, the observed improvements significantly depend on the amount of communication involved in the applications. Additionally, JFS reduces the CPU load of socket processing on Gigabit Ethernet up to 60% compared to Sun JVM sockets (SCI, Myrinet and InfiniBand rely on the high-speed NIC interconnect rather than on the CPU for the communication protocol processing).

Although JFS has significantly improved parallel and distributed Java applications performance, this library is also intended for middleware developers in order to implement JFS-based higher level communication libraries like Java message-passing and RMI implementations, as presented in subsequent chapters.

Table 3.2: JFS performance improvement compared to Sun JVM sockets

	SCI	Myrinet	InfiniBand	Gig. Eth.	Shared mem.
Start-up reduction	up to 88%	up to 78%	up to 65%	up to 10%	up to 50%
Bandwidth increase	up to 1305%	up to 412%	up to 860%	up to 119%	up to 4411%

Chapter 4

Efficient iodev Low-level Message-Passing and RMI Middleware

This chapter presents two Java communication middleware implementations: (1) iodev, a low-level communication device especially designed to be the base of Java parallel applications and higher level communication middleware such as message-passing libraries; and (2) an optimization of the Java RMI protocol for high-speed clusters. Both solutions are implemented on top of the Java Fast Sockets (JFS) middleware developed in the previous chapter. Although Java NIO sockets already provide features such as non-blocking communication methods, and thus some MPJ libraries have implemented their communication support on these sockets (e.g., MPJ Express [81] and P2P-MPI [36]), their use in message-passing middleware usually results in lower performance than using Java IO sockets, which do not provide non-blocking methods. Thus, the implementation of an efficient low-level message-passing library should use Java IO sockets, and therefore implement the non-blocking support on these sockets. The iodev message-passing device provides efficient non-blocking communication on top of any Java IO sockets library, but it is on JFS that iodev maximizes its throughput, especially on shared memory and high-speed clusters and when avoiding the serialization overhead. The comparative performance evaluation of iodev and other low-level communication devices, both Java and native implementations, shows that iodev obtains almost native performance when using JFS.

The optimization of the RMI protocol developed in this chapter is focused on: (1) the overhead reduction of the RMI transport layer through the use of JFS and the minimization of the amount of data to be transferred; (2) avoiding the serialization overhead for primitive data type arrays thanks to JFS; and (3) maximizing the object manipulation performance through a new serialization method for array processing, and reducing the versioning information and the class annotations.

The structure of this chapter is as follows: Section 4.1 describes the design of iodev. The novel issues in its implementation, together with its communication algorithm operation, are shown in Section 4.2. The implementation details on different underlying communication libraries are also covered in this section. The comparative performance evaluation of iodev and other low-level message-passing communication devices on Gigabit Ethernet, SCI, Myrinet, InfiniBand and shared memory is shown in Section 4.3. This evaluation consists of a micro-benchmarking of point-to-point communications. Section 4.4 presents the RMI protocol optimization. The performance results of the micro-benchmarking of three RMI implementations, Sun JVM RMI, KaRMI and the optimized RMI, on Gigabit Ethernet and SCI are discussed in Section 4.5. Finally, Section 4.6 summarizes the main contributions presented in this chapter.

4.1. Low-level Message-Passing Devices Overview

The use of pluggable low-level communication devices is widely extended in message-passing libraries. Thus, MPICH/MPICH2 [40] include several devices that implement the Abstract Device Interface (ADI/ADI3), the MPICH low-level messaging API, on several communication layers (e.g., GM/MX for Myrinet, IBV/VAPI for InfiniBand, and shared memory). Moreover, OpenMPI [74] also contains several Byte-Transfer-Layer (BTL) communication devices (modules) in its implementation (e.g., also on GM/MX for Myrinet, IBV/VAPI for InfiniBand, and shared memory). Regarding MPJ libraries, in MPJ Express the low-level xdev layer [8] provides communication devices for different interconnection technologies. The two implementations of the xdev API currently available are niodev over Java NIO sockets, a "pure" Java communication device, and mxdev over MX, a wrapper to a native library device for the Myrinet. Moreover, there is another device, not publicly available yet, on shared memory, based on Java threads [83].

In order to follow this approach, the xxdev (eXtended xdev) layer which presents an API similar to xdev, but with several improvements, has been defined. The motivation behind trying to stick to the xdev API is to favor the standardization of the xdev/xxdev API as low-level Java communication layer in Java applications and message-passing libraries. The improvements of xxdev compared to xdev are the incorporation of additional functionalities (e.g., allowing the communication of any serializable object without data buffering) and the use of a more encapsulated design, as xdev references classes outside the standard Java library (from the mpjdev –higher level messaging– and mpjbuf –buffering layer– packages of MPJ Express), whereas xxdev does not. Thus, xxdev is more flexible (communicates any serializable object), portable, modular and hence, more reusable than xdev.

The low-level xxdev layer provides a simple (only 13 methods, see Subsection 4.2.1) but powerful message-passing API, as it has served as base in the development of our message-passing library Fast MPJ (F-MPJ, detailed in Chapter 5). Moreover, the reduced set of methods in xxdev eases the implementation of xxdev communication devices for specific interconnection technologies. However, iodev, the implementation of the xxdev API using Java IO sockets, can run on top of JFS, thus obtaining high performance on SCI, Myrinet, InfiniBand and Gigabit Ethernet communications and on shared memory.

Figure 4.1 presents an overview of the layered design of the communication middleware developed in this Thesis on representative HPC hardware: high-speed interconnects and shared memory systems. From top to bottom, first the MPJ libraries, such as F-MPJ, and the xxdev-based applications, which are implemented directly on top of the low-level xxdev API for performance reasons, can be seen. The optimization of the communications in this Thesis has been performed up to this upper layer. Therefore, their underlying middleware, the iodev communication device, i.e., the implementation of the xxdev API on Java IO sockets, should provide efficient communication. For purposes of clarity, we denote the IO sockets API as “Java IO sockets”. Two implementations of Java IO sockets are considered in this work: the default JVM IO sockets and JFS. In order to provide Java with high performance communications the iodev device accesses HPC hardware through JNI using either JFS or the standard JVM IO sockets (TCP). However, as already shown in Figure 3.3, the JVM IO sockets resort to IP emulations (SCIP, IPoMX and IPoIB on SCI, Myrinet and InfiniBand, respectively), whereas JFS relies on high performance native libraries (SCI Sockets, Sockets-MX and Sockets Direct Protocol –SDP– on SCI, Myrinet and InfiniBand, respectively). These IP emulations and native libraries accessed by JFS and the JVM IO sockets are presented below the JNI layer in Figure 4.1. IP emulations usually provide wider support but a higher communication overhead than high performance native sockets. In fact, JVM IO sockets are usually only supported by IP emulations. Thus, iodev provides efficient communication over high performance native libraries through the use of JFS, if available. If JFS is not available, iodev resorts to HPC hardware through the standard JVM sockets and IP emulations, maintaining the portability of the solution. The design and implementation details of the iodev operation are presented in the next section.

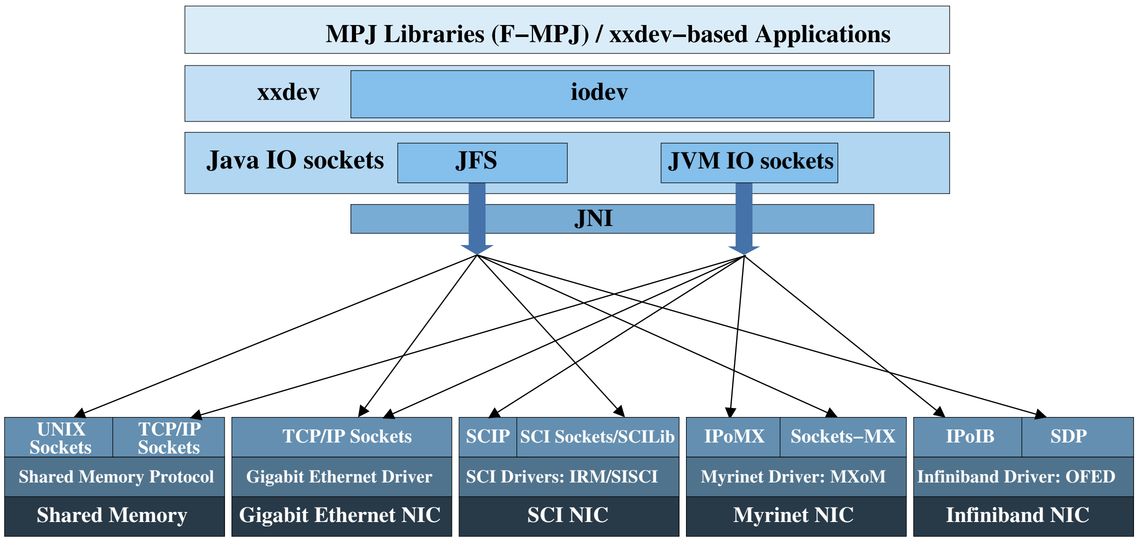


Figure 4.1: xxdev low-level communications stack on high-speed multi-core clusters

4.2. iodev: Low-level Message-Passing Library

The iodev low-level message-passing communication device implements the xxdev API on Java IO sockets. Although Java NIO sockets provide additional features not present in Java IO sockets, such as the direct non-blocking communication support, which are quite useful for the implementation of communication middleware, the Java NIO sockets-based MPJ libraries usually show lower performance than the ones implemented on top of Java IO sockets (see Sections 2.3, 2.5 and 3.3, where performance results are presented for MPJ Express, based on Java NIO, and MPJ/Ibis, on top of Java IO sockets). This lower performance could be due to a poor implementation of the Java NIO sockets-based MPJ libraries, but this hypothesis was discarded as it has been checked that the underlying socket implementation has an important impact on the message-passing communication performance. In fact, Java NIO sockets present higher start-up latencies than Java IO sockets, reducing the performance of MPJ applications, especially for communication-intensive codes (see Section 2.5). Therefore, iodev uses Java IO sockets, and thus it has to implement efficiently the non-blocking communication support on this socket library. Moreover, Java IO sockets have also been selected as underlying layer for iodev in order to take advantage of the JFS middleware, the Java IO sockets developed in the previous chapter, which provides shared memory and high-speed networks support. Thus, iodev can rely either on the standard JVM sockets or on JFS, if it is available. This combination of a portable JVM-based implementation with a custom solution for HPC native libraries provides both portability and high performance.

Other options considered for the development of communication devices that implement the xxdev API are RMI and asynchronous Java sockets [43], but they have been discarded due to its high communication overhead and the lack of portability, respectively. Furthermore, both solutions do not provide high-speed networks support. The following subsections present the design of the xxdev low-level communication layer, the iodev implementation together with its communication protocols operation, and finally the efficient JFS support in iodev, which provides high performance low-level message-passing communications for Java on shared memory and high-speed networks.

4.2.1. Design of the xxdev Low-level Communication Layer

The xxdev API has been designed with the goal of being simple, providing only basic communication methods in order to ease the development of xxdev devices. A communication device is similar to an MPI communicator, but with reduced functionality. Thus, the xxdev API, presented in Listing 4.1, is composed of 13 methods. Moreover, its API extends the MPJ Express xdev API, allowing the communication of any serializable object instead of being limited to transfer only the custom MPJ Express buffer objects. The `newInstance` method instantiates the pluggable xxdev device implementations. The `init` method first reads machine names, ports and ranks from a configuration file (passed as a parameter in `args`), creates the connections, disables Nagle's algorithm and sets socket buffers, both for sending and receiving, to 512 KB. Then, the identification of the initialized device is broadcast through all the open connections. Finally, the identifiers of the communication peers are gathered in order to complete the initialization. The `id` method returns the identifier (`id`) of the device. The `finish` method is the last method to be called and completes the device operation.

The xxdev communication primitives only include point-to-point communication, both blocking (`send` and `recv`, like MPI_Send and MPI_Recv) and non-blocking (`isend` and `irecv`, like MPI_Isend and MPI_Irecv). Synchronous communications are also embraced (`ssend` and `issend`). These methods use as `dst` (destination) and `src` (source) parameters the identifiers read from the configuration file. The `probe` method waits until a message matching `src`, `tag` and `context` arrives. Its non-blocking version, `iprobe`, checks if the message has been received. The `peek` method (blocking) returns the most recently completed `Request` object, useful for the `Request.iwaitany` implementation. Listing 4.2 presents the API of the `Request` class, whose wait methods are used to complete the non-blocking communications. Thus, each non-blocking primitive, isend/issend or irecv, returns a Request object which handles the communication. A call to its `iwait` method blocks the code execution until the communication has been completed. The static method `iwaitany` also blocks the calling thread until any of the requests from the array of Requests `reqs` has been completed. The `itest` method checks the completion status of the communication associated to the `Request` object. The `cancel` method cancels the non-blocking operation.

All methods in the `Device` and `Request` classes handle runtime errors throwing exceptions of type `XxdevException`. Thus, the interface of these methods should include `throws XxdevException` after the parameter list, but it is not shown for clarity purposes. Despite the simplicity of the xxdev API, a Java message-passing library can implement its communications exclusively on top of it, as will be demonstrated in Chapter 5, making an intensive use of non-blocking methods for communications overlapping.

Listing 4.1: Public interface of the *xxdev.Device* class

```
public abstract class Device {
 static public Device newInstance(String deviceImpl);
 public int[] init(String[] args);
 public int id();
 public void finish();

 public Request isend(Object buf, int dst, int tag);
 public Request irecv(Object buf, int src, int tag, Status stts);

 public void send(Object buf,  int dst, int tag);
 public Status recv(Object buf,  int src, int tag);

 public Request issend(Object buf,  int dst, int tag);
 public void ssend(Object buf,  int dst, int tag);

 public Status iprobe(int src, int tag, int context);
 public Status probe(int src, int tag, int context);
 public Request peek();
}
```

Listing 4.2: Public interface of the *xxdev.Request* class

```
public class Request {
 public Status iwait();
 public static Status iwaitany(Request[] reqs)
 public Status itest();
 public boolean cancel();
}
```

4.2.2. Implementation of the iodev Communication Device

The iodev device implements the low-level multiplexed, non-blocking communication primitives on top of Java IO sockets. In iodev each process is connected to every other process through two TCP sockets, one for sending and another for receiving. This is a design decision in order to reduce synchronization overheads when sending/receiving data to/from the same peer process. The access to these sockets, both for reading and writing, is controlled by locks, as several threads have read/write access to these sockets.

In iodev all communication methods are based on the non-blocking primitives `isend/irecv`. Thus, blocking communication methods are implemented as a non-blocking primitive followed by an `iwait` call. In order to handle the non-blocking communications their `Request` objects are internally stored in two sets named `pending_sendRequestSet` and `pending_recvRequestSet`.

An iodev message consists of a header plus data. The message header includes the `datatype` sent, the source identification `src`, the message `size`, the `tag`, the `context` and `control` information. In order to reduce the overhead of multiple accesses to the network the iodev message header is buffered. Once the message header buffer has been filled in, it is written to the network. The message data is next sent to the network. Thus, only two accesses are required for each message, although for very short messages (<4 KB) the header and data are merged in order to perform a single socket write call. When source and destination of a message are the same the socket communication is replaced by an array copy.

Regarding message identification, in iodev a message is unequivocally identified by the triplet `src`, `tag` and `context`, although the wildcard values `xxdev.Device.ANY_SRC` and `xxdev.Device.ANY_TAG` skip `src` and `tag` matching, respectively. The message reception is carried out by both the `input handler`, a thread in charge of receiving data (also known in the literature as the progress engine), and the `Request.iwait` method. Usually, in message-passing libraries, both native and Java implementations, only the `input handler` receives messages. This presents a high reception overhead that consists of: (1) the reception of the message by the `input handler`; (2) the notification of the reception to the `Request` object, which is in a `wait` state; (3) waking up the `Request` object; and (4) context switching between the `input handler` and the `Request`, in order to continue the process execution. However, in iodev both the `input handler` thread and the `Request.iwait` method receive messages. Thus, if `Request.iwait` receives the message the overhead of the `input handler` reception is avoided.

Figure 4.2 shows the `Request.iwait` pseudocode in order to illustrate its reception operation. It can be seen that iodev implements a polling strategy together with periodically issued yield calls, which decrease `iwait` thread priority in order to not monopolize system CPU. This strategy allows to significantly reduce message latency in exchange for a moderate CPU overhead increase, compared with the approach where only the `input handler` receives data. This iodev approach yields significant benefits, especially in communication-intensive codes, as message latency reduction provides higher scalability than the availability of more CPU power.

```
Method Request.iwait():Status

if alreadyCompleted then
    return status;
init_timer();
while completed = false do
    receive_data();
    if received_the_expected_data then
        completed ← true;
    if $timer_{elapsed} > max_{polling-time}$ then
        current_thread_yield();
        reset_timer();
status ← new Status(statusDetails);
alreadyCompleted ← true;
return status;
```

Figure 4.2: *Request.iwait* method pseudocode

4.2.3. iodev Communication Protocols

The iodev device implements the eager and rendezvous protocols, targeted to short and long messages, respectively. The threshold between these protocols is configurable and usually ranges from 128 to 512 KB.

iodev Eager Protocol

The eager protocol is targeted to short messages, typically below 128 KB. It is based on the assumption that the receiver has available storage space (otherwise an out of memory exception is thrown), so there is no exchange of control messages before the actual data transfer. This strategy minimizes the overhead of control messages, that can be significant for short messages, although it adds the extra copy overhead when the receiver is not waiting for the message.

Figure 4.3 shows eager protocol pseudocode. Regarding eager `isend` operation, the sender writes the data under the assumption that the receiver will handle it. At the receiver side there are two possible scenarios for the `input handler` (see pseudocode in Figure 4.3), depending on whether a matching receive has been already posted or not. Thus, if a matching `recvRequest` exists the message is copied into the destination buffer; otherwise, it will be stored in a temporary buffer, waiting for

```
Method isend(buffer,dst,tag,context):Request (Eager)

  sendRequest ← new SendRequest(buffer,dst,tag,context);
  send(dst,buffer);
  sendRequest.completed ← true;
  return sendRequest;
```

```
Method input handler thread (Eager)

  while running do
    receive_header(messageHeader);
    rRequest ← new RecvRequest(messageHeader);
    if rRequest in pending_recvRequestSet then
      recvRequest ← pending_recvRequestSet.remove(rRequest);
      recvRequest.buffer ← receive_data();
    else
      rRequest.temp_buffer ← receive_data();
      pending_recvRequestSet.add(rRequest);
```

```
Method irecv(buffer,src,tag,context,status):Request (Eager)

  rRequest ← new RecvRequest(buffer,src,tag,context,status);
  if rRequest in pending_recvRequestSet then
    recvRequest ← pending_recvRequestSet.remove(rRequest);
    buffer ← recvRequest.temp_buffer;
    return recvRequest;
  else
    pending_recvRequestSet.add(rRequest);
    return rRequest;
```

Figure 4.3: *iodev eager* protocol pseudocode

the corresponding `irecv` post. The `input handler` is constantly running during iodev operation, from the `init` up to the `finish` call. This behavior is controlled by a flag (`running`). The `irecv` operation (see Figure 4.3) also presents two scenarios, depending on whether the input handler has already received the message or not. This iodev eager protocol implementation significantly reduces the short message transfer overhead, allowing short message communication-intensive MPJ applications to significantly increase their scalability.

iodev Rendezvous Protocol

The rendezvous protocol is targeted to long messages, typically above 128 KB. It is based on the use of control messages in order to avoid buffering. Thus, the steps of the protocol are: (1) the source sends a ready-to-send message; (2) the destination replies with a ready-to-receive message; and (3) data is actually transferred. This strategy avoids buffering although it increases protocol overhead. However, the impact of the control messages overhead is usually reduced for long messages.

Figure 4.4 shows rendezvous protocol pseudocode. The `isend` operation consists of writing a ready-to-send control message. At the receiver side there are three possible scenarios for the `input handler` (see pseudocode in Figure 4.4), depending on the incoming message: (1) a ready-to-send message; (2) a ready-to-receive message; or (3) a data message. In scenario (1) a ready-to-receive message reply is written if a matching receive has been posted; otherwise, the ready-to-send message is stored until such matching receive is posted. In (2) the actual transfer of the data is performed through a forked thread in order to avoid `input handler` blockage while writing data. In this case the `input handler` is run by the sender process and therefore can access the source buffer. Finally, in (3) the `input handler` receives the data. The `irecv` operation (see Figure 4.4) presents two scenarios, depending on whether the input handler has already received the ready-to-send message or not. Thus, it either replies back with a ready-to-receive message or stores the receive post, respectively. This iodev rendezvous protocol implementation contributes significantly to Java message-passing scalability as it reduces the overhead of message buffering and network contention. Therefore, scalable Java communication performance can be achieved.

Method `isend(`*buffer,dst,tag,context*`)`:Request (Rendezvous)

```
sendRequest ← new SendRequest(buffer,dst,tag,context);
pending_sendRequestSet.add(sendRequest);
send(dst,ready-to-send_Message);
sendRequest.completed ← false;
return sendRequest;
```

Method `input handler` (Rendezvous)

```
while running do
    messageHeader ← receive_header();
    request ← new Request(messageHeader);
    if messageHeader from a ready-to-send_Message then
        if request in pending_recvRequestSet then
            pending_recvRequestSet.remove(request);
            send(src,ready-to-recv_Message);
        else
            pending_recvRequestSet.add(request);
    else if messageHeader from a ready-to-receive_Message then
        Fork: rendez_Write_Thread:
        begin
            sendRequest ← pending_sendRequestSet.remove(request);
            send(dst,sendRequest.buffer);
            sendRequest.completed ← true;
        end
    else if messageHeader from a dataMessage then
        recvRequest ← pending_recvRequestSet.remove(request);
        recvRequest.buffer ← receive_data();
```

Method `irecv(`*buffer,src,tag,context,status*`)`:Request (Rendezvous)

```
rRequest ← new RecvRequest(buffer,src,tag,context,status);
if rRequest in pending_recvRequestSet then
    send(src,ready-to-recv_Message);
else
    pending_recvRequestSet.add(rRequest);
return rRequest;
```

Figure 4.4: *iodev rendezvous* protocol pseudocode

4.2.4. Java Fast Sockets Support in iodev

The default sockets library used by iodev, JVM IO sockets, presents several disadvantages for communication middleware: (1) this library has to resort to serialization; (2) as Java can not serialize/deserialize array portions (except for parts of byte arrays) a new array must be created to store the portion to be serialized/deserialized; (3) JVM IO sockets perform an extra copy between the data in the JVM heap and native memory in order to transfer the data; and finally, (4) this socket library is usually not supported by high performance native communication libraries, so it has to rely on IP emulations, a solution which presents a poorer performance.

However, in order to avoid these drawbacks, iodev has integrated the high performance Java sockets library JFS (see Chapter 3) in a portable and efficient way. Thus, JFS boosts iodev communication efficiency by: (1) avoiding primitive data type array serialization through an extended API that allows direct communication of primitive data type arrays (see Listing 4.3); (2) making unnecessary the data buffering when sending/receiving portions of primitive data type arrays using `offset` and `length` parameters (see JFS API in Listing 4.3 and its application in Listing 4.4); (3) avoiding the copies between the JVM data and native memory thanks to JFS's zero-copy protocol; and (4) providing efficient support on shared memory, and Gigabit Ethernet, SCI, Myrinet and InfiniBand networks through the use of the underlying high performance native libraries specified in Figure 4.1.

Listing 4.3: JFS extended API for communicating primitive data type arrays directly

```
jfs.net.SocketOutputStream.write(byte buf[], int offset, int length);
jfs.net.SocketOutputStream.write(int buf[], int offset, int length);
jfs.net.SocketOutputStream.write(double buf[], int offset, int length);
  ...
jfs.net.SocketInputStream.write(byte buf[], int offset, int length);
jfs.net.SocketInputStream.read(int buf[], int offset, int length);
jfs.net.SocketInputStream.write(double buf[], int offset, int length);
  ...
```

Listing 4.4 presents an example of iodev code that takes advantage of the efficient JFS methods when they are available, without compromising the portability of the solution. The communication of part of an integer array (`num_elements` entries) is straightforward with JFS: a single call to a method from the extended API of JFS is enough (see Listing 4.3). Nevertheless, the same task using JVM IO sockets requires: (1) the creation of a new array of the size of the slice to be sent; (2) copy the `num_elements` entries to be transferred to the new array; and (3) the sending of the new array through an `ObjectOutputStream`, which can involve up to nine steps: a serialization, three copies, a network transfer, other three copies and a deserialization, whereas JFS performs the same operation in only up to three steps (see Subsection 3.1.2). This handling of JFS communications is of special interest in message-passing libraries and, in general, in any communication middleware, as Java applications can benefit from the use of JFS without modifying their source code.

The integration of JFS in iodev has been done following this approach and thus preserving its portability while taking full advantage of the underlying communication middleware. In fact, JFS, in the presence of two or more supported libraries, prioritizes them depending on their performance: usually shared memory communication first, then high performance native socket libraries, and finally the default "pure" Java implementation, which relies on TCP/IP sockets and IP emulations.

Listing 4.4: JFS-based support in iodev for sending parts of arrays

```
if (os instanceof jfs.net.SocketOutputStream) {
  jfsAvailable = true;
  jfsos = (jfs.net.SocketOutputStream) os;
}
oos = new ObjectOutputStream(os);

[...]

// Writing int_array[offset] ... int_array[offset+num_elements-1]
if (jfsAvailable)
  jfsos.write(int_array, offset, num_elements);
else {
  int[] intBuf = (int[])Array.newInstance(int.class, num_elements);
  System.arraycopy(int_array, offset, intBuf, 0, num_elements);
  oos.writeUnshared(intBuf);
}
```

JFS significantly outperforms JVM sockets, especially in shared memory, high-speed networks and hence in hybrid shared/distributed memory architectures (e.g., high-speed multi-core clusters). Moreover, JFS is targeted to primitive data type array communications, frequently used in HPC applications. Thus, the iodev low-level message-passing communication device greatly benefits from the use of JFS. Therefore, a communication middleware that takes advantage of iodev would improve significantly its performance without losing portability, as will be experimentally assessed in Chapter 5 for our Fast MPJ (F-MPJ) library.

4.3. Performance Evaluation of iodev

The evaluation presented in this section consists of a micro-benchmarking of point-to-point primitives on two multi-core clusters. The iodev performance has been compared with the MPJ Express library communication devices (niodev on Java NIO sockets and mxdev on Myrinet), with mpiJava and with native MPI libraries, using Gigabit Ethernet, SCI, Myrinet and InfiniBand NICs and on a shared memory scenario.

4.3.1. Experimental Configuration

The testbed used for the performance evaluation of iodev on Gigabit Ethernet, SCI, Myrinet and shared memory is the same as the one used in the JFS evaluation (see Section 3.2). The only changes are the JVM and the C compiler, Sun JDK 1.6.0_05 and PGI pgcc 7.2 with the flags -O3 and -fast, respectively. However, the InfiniBand results have been obtained on the Finis Terrae supercomputer [34], ranked #427 in November 2008 TOP500 list [99] (14 TFlops), an Itanium2 (IA64) Linux multi-core cluster (2400 cores). This supercomputer will be also used for the performance evaluation of our MPJ library, Fast MPJ (F-MPJ), in the next Chapter (see Section 5.4) due to its higher number of cores compared to the other high-speed cluster used in this section (with only 32 cores).

The Finis Terrae consists of 142 HP Integrity rx7640 nodes, each of them with 16 Montvale Itanium2 (IA64) cores at 1.6 GHz and 128 GB of memory. The InfiniBand NIC is a dual 4X IB port Mellanox Technologies MT25208 InfiniHost III Ex (16 Gbps of theoretical effective bandwidth). The OS is SUSE Linux Enterprise Server 10 with C compiler Intel icc 9.1 used with the flags -O3 and -fast. The InfiniBand driver is OFED 1.2 (see Figure 4.1). The JVM is BEA JRockit 5.0 (R27.5). This JVM has been selected as it shows the highest performance on this system, significantly outperforming Sun JVM, the other JVM available for IA64 systems.

The Java low-level communication devices evaluated are niodev and mxdev from MPJ Express 0.27 and iodev. Additionally, for comparison purposes the native MPI libraries ScaMPI 1.13.8 on SCI, MPICH-MX 1.2.6 on Myrinet, MVAPICH2 1.0.2 on InfiniBand, and MPICH2 1.0.6 on Gigabit Ethernet and shared memory have been evaluated. This latter MPI library has been used, both for Gigabit Ethernet and shared memory communication, with the ch3 device and its specific channel ssm (sockets and shared memory), specially designed for its use on multi-core clusters (sockets are used for inter-node communication, whereas shared memory is used for intra-node transfers). Moreover, mpiJava 1.2.5x, an MPJ wrapper library on top of these native MPI implementations (see Section 1.3), has also been benchmarked. The benchmark results have been obtained at the communication device level for niodev, mxdev and iodev, whereas the native MPI libraries have been benchmarked at the MPI level due to the different APIs of their native low-level communication devices. However, their performance results are comparable to those of the Java communication devices as the high level layers of native MPI libraries usually add quite low overhead (almost negligible) to their underlying low-level message-passing devices. The mpiJava library has also been benchmarked at the MPI level as it relies on a native MPI library for communication, not on a communication device.

4.3.2. Point-to-point iodev Micro-benchmarking

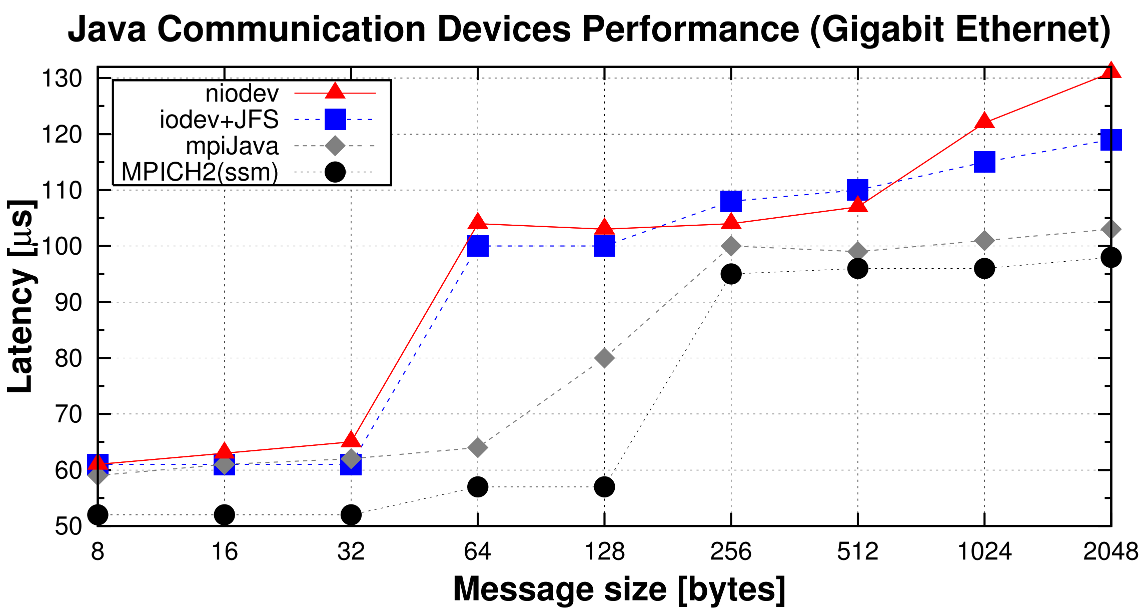
Figures 4.5–4.9 show experimentally measured point-to-point latencies and bandwidths for byte arrays using niodev (mxdev for Myrinet) and iodev over JFS. Additionally, mpiJava and native MPI results are shown for comparison purposes.

As general observations shared among the different configurations evaluated in this micro-benchmarking, it can be stated that:

- MPI implementations always outperform Java libraries, except for iodev+JFS on shared memory.

- The combination of iodev+JFS generally obtains the best performance among the Java message-passing communication devices.
- mpiJava shows the lowest long message bandwidth, due to the overhead of the JNI copy of the message data between the Java heap and the MPI library. This fact might seem to contradict the results of Section 2.3, where mpiJava obtained almost native MPI performance. However, its low overhead on the scenario ScaMPI/MPICH-GM + Sun JVM 1.4.2 was due to the avoidance of the JNI copy of the message data in mpiJava, which depends on the system, the MPI library and the JVM. Nevertheless, for the experimental configurations used in the current section (the JVMs Sun 1.6 or JRockit 5.0 over the native MPI libraries MPICH2, ScaMPI, MPICH-MX and MVAPICH2), it is not possible to avoid this JNI data copy. In fact, modern JVMs do not implement the pinning in memory of arrays from the Java heap, a feature that allows mpiJava to wrap native MPI libraries with low overhead.
- The start-up overhead added by mpiJava wrapping to native MPI is around $9\mu s$, independently of the MPI library and high-speed network.
- niodev presents the highest short message latencies, due to the high start-up overhead of Java NIO sockets.

The network is the main performance bottleneck for the Gigabit Ethernet results (shown in Figure 4.5). Actually, measured bandwidths are below 1000 Mbps and start-up latencies (the 0-byte message latency) are quite high (around $50\mu s$), imposing low performance for short messages. Moreover, the communication driver delays communications, causing latencies to be around multiples of $50\mu s$. Thus, the communication library implementation has a minor impact on performance results. In fact, the obtained results are quite similar among them. However, it is worth mentioning that mpiJava slightly outperforms niodev/iodev for short messages, whereas its long message performance is around 30% lower than the Java communication devices. The thresholds between eager and rendezvous send protocols can be observed in the bandwidth graph at 128 KB for all libraries except iodev, which shows an efficient implementation of the rendezvous protocol, without losing performance for medium-size messages.



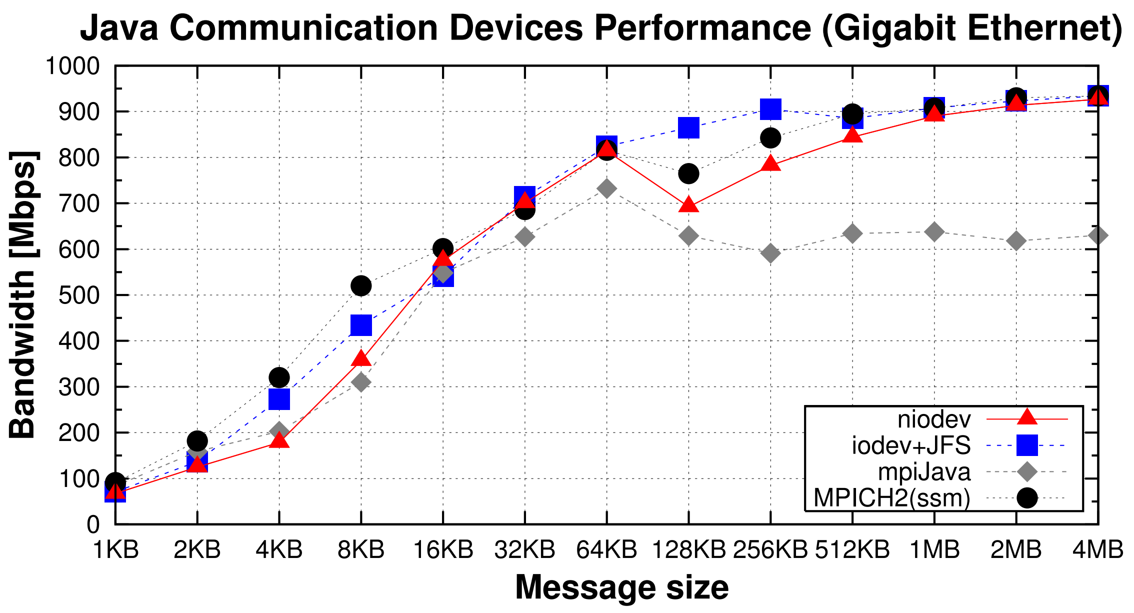
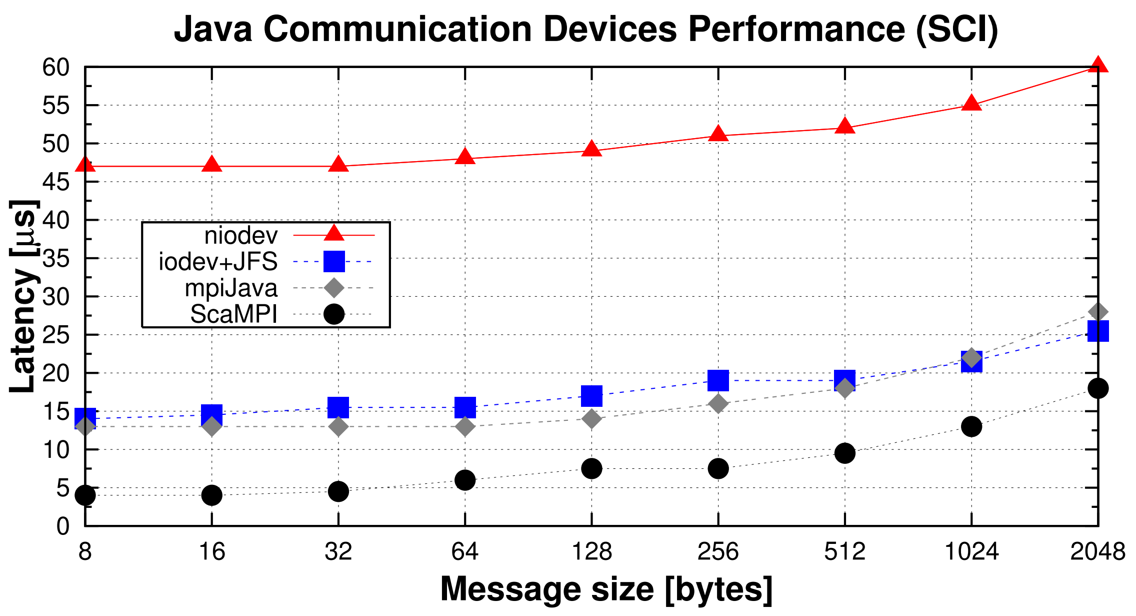


Figure 4.5: Point-to-point message-passing devices performance on Gigabit Ethernet

Figure 4.6 presents the performance of the evaluated results on SCI. Regarding short message performance, the native MPI implementation obtains the lowest start-up latency, $4\mu s$. The mpiJava wrapper adds an overhead of $9\mu s$ over the native layer (the mpiJava start-up latency is $13\mu s$). Moreover, the results of iodev+JFS are quite similar to mpiJava performance for short messages (its start-up latency is $14\mu s$). Nevertheless, niodev shows poor performance for short messages, with a start-up latency of $47\mu s$. With respect to large message bandwidths, ScaMPI, the native MPI, gets the best results, whereas iodev+JFS obtains the best performance among Java libraries. In fact, the results of iodev+JFS are quite similar to the native performance (around $90-95\%$ of ScaMPI results). Although mpiJava presents similar bandwidths to iodev+JFS for messages up to 64 KB, for larger messages the performance is around 60% of the ScaMPI results. The high start-up latency of niodev has a great impact on its performance, achieving $85-90\%$ of the iodev+JFS performance only for 64 KB messages (the largest message size fo the eager protocol) and for data transfers longer than 1 MB (rendezvous protocol), for which the impact on performance of the high overhead of the rendezvous protocol is reduced. In fact, for a 128 KB niodev message (see Figure 4.6) the rendezvous protocol (which involves two extra control messages and the actual data transfer) obtains around half of the performance that a 128 KB message would achieve using the eager protocol (only one data transfer).

Figure 4.7 shows the performance results on Myrinet, among which mxdev and iodev+JFS present quite similar overhead. Their start-up latencies are 17 and $16\mu s$, respectively, slightly higher values than the performance of mpiJava ($13\mu s$), which represents again an overhead of $9\mu s$ over the start-up latency of the native layer, MPICH-MX ($4\mu s$). Moreover, mxdev and iodev+JFS get similar large message bandwidths to the native MPI. In fact, the Myrinet NIC is the performance bottleneck for large messages, as its theoretical maximum bandwidth is 2000 Mbps. Thus, MPICH-MX, mxdev and iodev+JFS obtain around 1800 Mbps, 90% of the theoretical network bandwidth limit. The performance of mpiJava decreases as the message size grows, lowering down to 800 Mbps for large messages, due to the extra JNI data copy that occurs in this testbed between the Java heap and MPICH-MX.



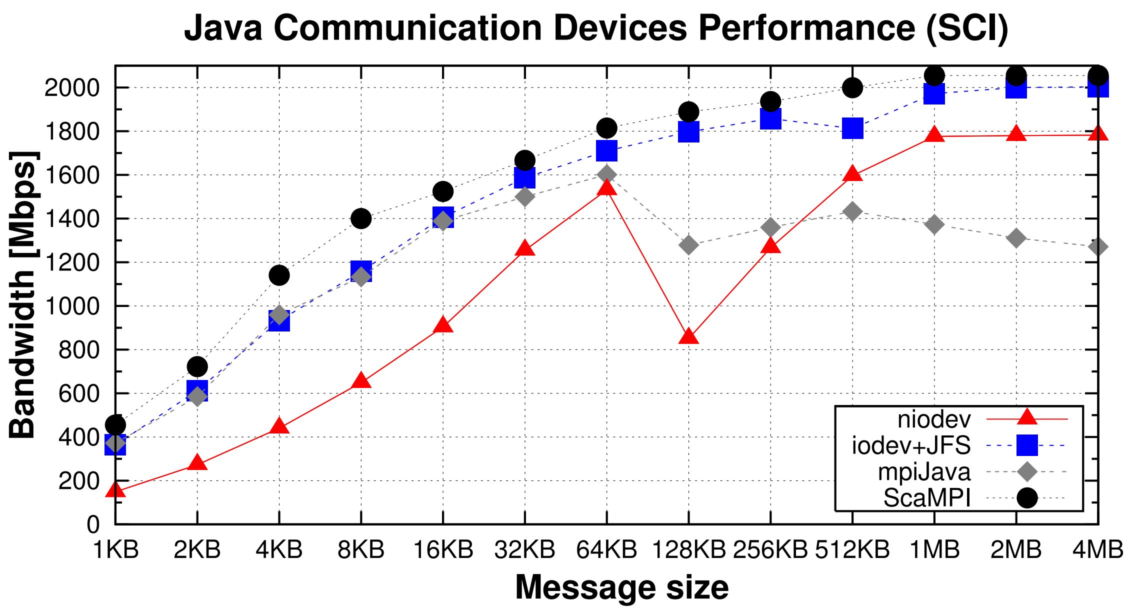
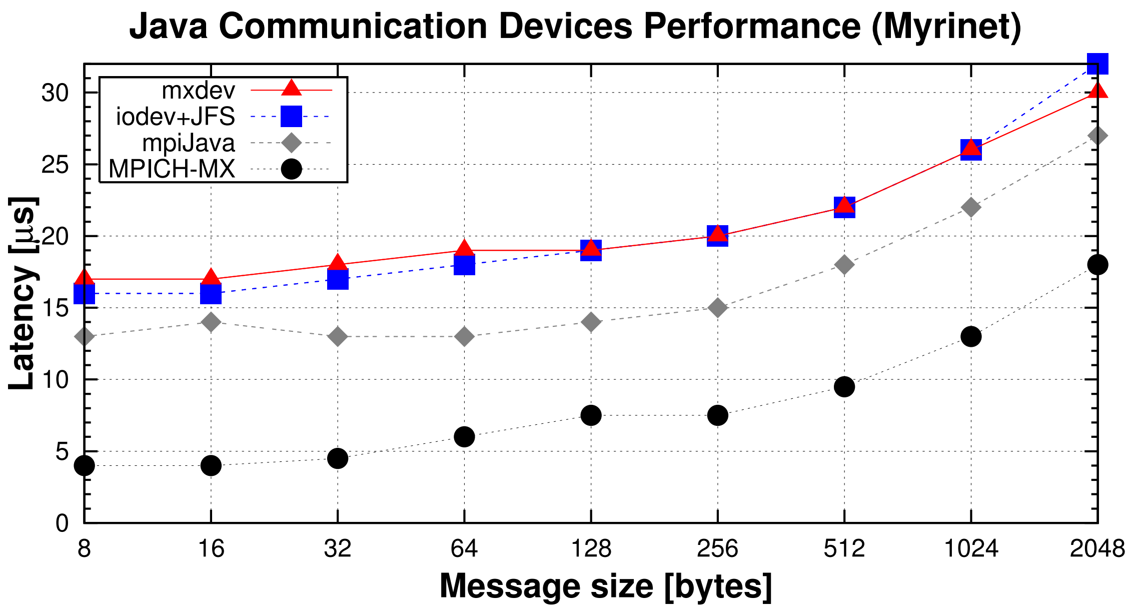


Figure 4.6: Point-to-point message-passing devices performance on SCI



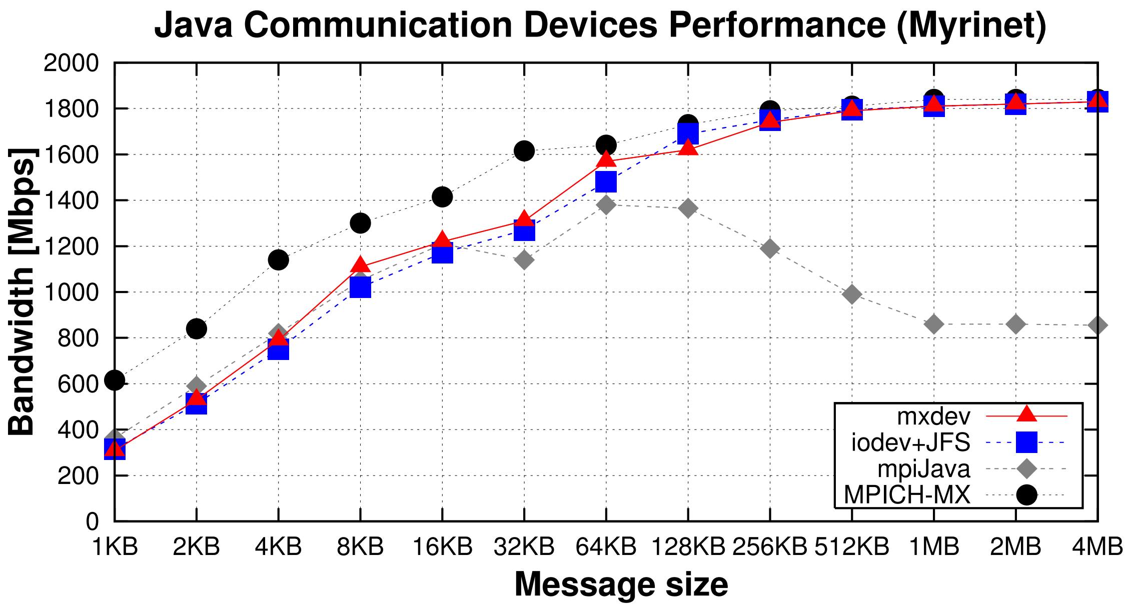
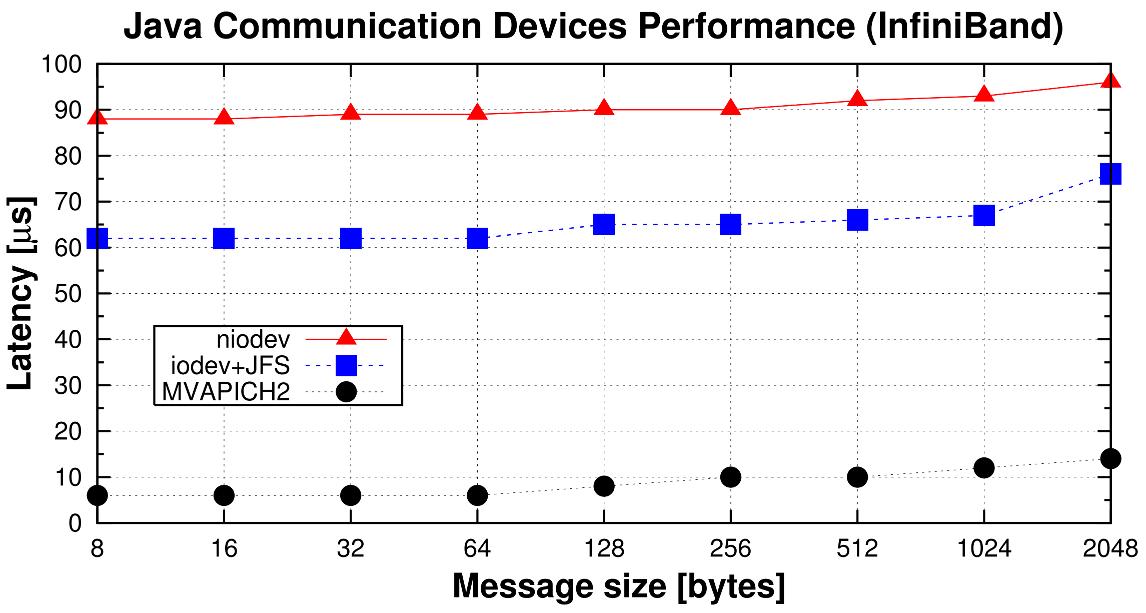


Figure 4.7: Point-to-point message-passing devices performance on Myrinet

Figure 4.8 presents the results on the Linux IA64 InfiniBand testbed. The installation of mpiJava in this system was not possible as the combination of JRockit JVM, MVAPICH2 and Linux IA64 system libraries is not supported by this wrapper library. This is an example of the portability issues of mpiJava. As MPJ Express competes in performance with mpiJava without large message penalties nor portability problems, it is clear that MPJ Express can supersede mpiJava as the reference Java message-passing library, as stated in Section 1.3.

Regarding the performance results, there is a significant performance gap between Java (niodev and iodev+JFS) and native code due to the low performance of the JVM implementations on Linux IA64 systems. Nevertheless, this Linux IA64 system allows the use of a higher number of cores (128) than the available processors (8) on the InfiniBand Linux Intel Pentium IV cluster used in the previous chapter. Thus, from now on, the InfiniBand performance results are obtained on this IA64 system. The micro-benchmarking of JFS over InfiniBand on this Pentium IV cluster (see Subsection 3.2.2) has shown that its performance is quite similar to the native SDP library. Thus, the latencies of short messages are only 1 μs higher than the native layer and the large message bandwidth (from 128 KB) achieves 90% of the native results. However, the start-up latency of the Java communication devices on the InfiniBand IA64 system is quite high (62 and 88μs for iodev+JFS and niodev, respectively), showing results more typical of Gigabit Ethernet than of a high-speed network. As the message size grows the performance of Java communication devices increases, obtaining bandwidths of up to 7.5 and 9 Gbps, for niodev and iodev+JFS, respectively. However, MVAPICH2 shows bandwidths of up to 10.3 Gbps. Thus, Java achieves up to 71% and 87% of the results of the native MPI for niodev and iodev+JFS, respectively. The impact of the poor short message performance is so important that the performance of the rendezvous protocol suffers from sending two messages (the control message and the actual message data) instead of one (for the eager protocol). Thus, iodev+JFS obtains lower performance for a 512 KB message (rendezvous protocol) than for a 256 KB message (eager protocol) as the threshold between protocols has been set to this value.

The shared memory results (Figure 4.9) have been obtained on the Gigabit Ethernet/SCI/Myrinet cluster. The most noticeable result is that iodev+JFS outperforms MPICH2 for messages larger than 16 KB. The explanation of this fact is that UNIX sockets, used by JFS, outperform the shared memory protocol implemented



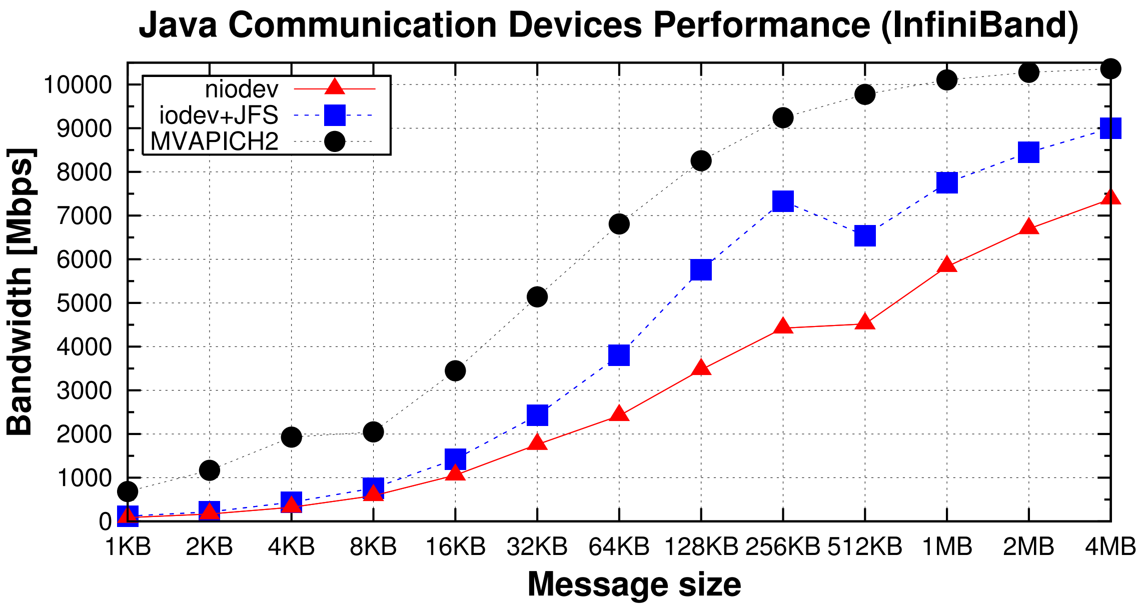
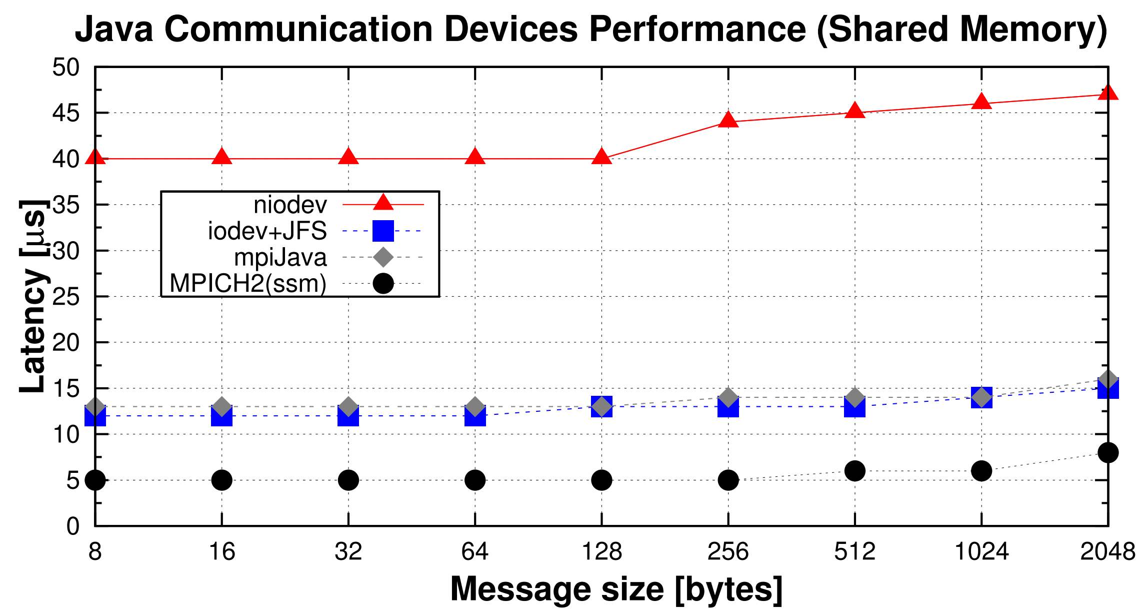


Figure 4.8: Point-to-point message-passing devices performance on InfiniBand



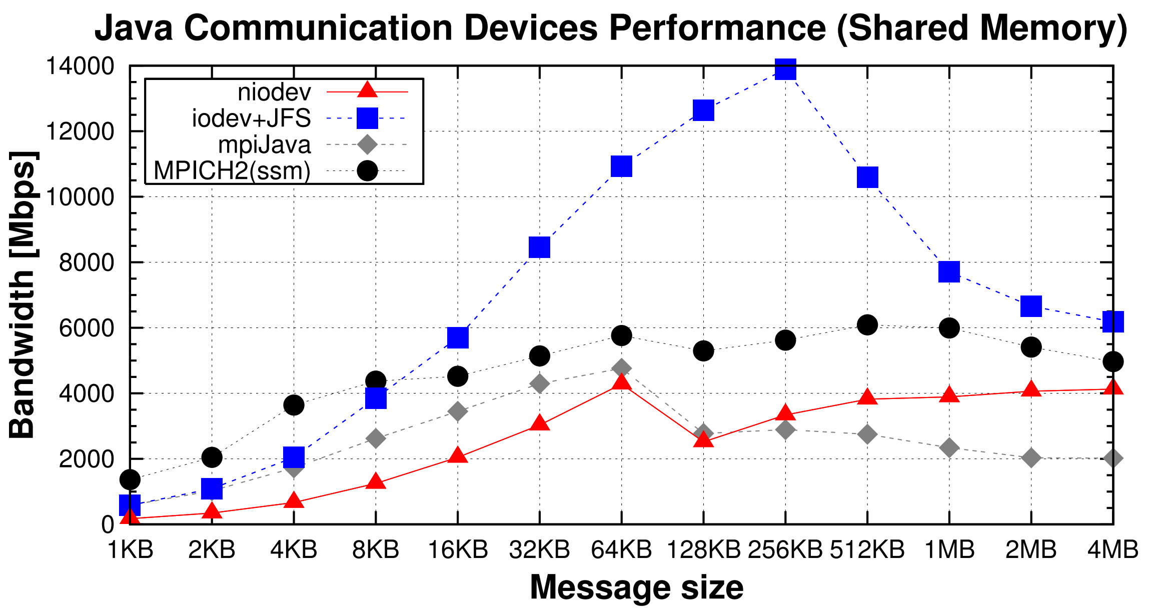


Figure 4.9: Point-to-point message-passing devices performance on Shared Memory

in the MPICH2 ssm channel, which obtains bandwidths only up to 6 Gbps. The iodev+JFS performance falls from 512 KB, the maximum buffer size of the sockets. The thresholds between eager and rendezvous send protocols can be observed in the bandwidth graph at 128 KB for niodev and mpiJava. Moreover, the results of mpiJava decrease as the message size grows, lowering down to half of the niodev performance for large messages (for a 4 MB message niodev achieves 4000 Mbps and mpiJava 2000 Mbps). The reason is the JNI overhead of the data copy between the Java heap and the buffers of MPICH2 that occurs in this testbed.

4.4. High Performance Java RMI

Java RMI is a widely extended communication protocol that usually presents poor performance on high-speed clusters, which has led to the development of several RMI optimization projects (see Section 1.2). This section presents an RMI protocol optimization for high-speed clusters named "Opt RMI" which takes advantage of JFS in a similar way to iodev (see Subsection 4.2.4). The design and implementation of Opt RMI has been done bearing in mind: (1) the advantages/disadvantages of previous Java RMI optimization projects analyzed in Section 1.2; (2) the transparency of the solution: it uses the standard API, increases the communication efficiency transparently to the user, no source code modification is required, and it is interoperable with other systems; and (3) several basic assumptions about the target architecture, high-speed clusters, such as a homogeneous cluster architecture and the use of a single JVM; out of these basic assumptions about high-speed clusters the Opt RMI library resorts to the JVM RMI implementation.

Java RMI has been designed following a layered architecture approach. Figure 4.10 presents, from bottom to top, the transport layer, responsible for managing all communications, the remote reference layer, responsible for handling all references to objects, the stub/skeleton layer, in charge of the invocation and execution, respectively, of the methods exported by the objects; and the client and server layer, also known as service layer. The activation, registry and Distributed Garbage Collection (DGC) services are also part of this service layer.

In order to optimize the Java RMI protocol, an analysis of the overhead of

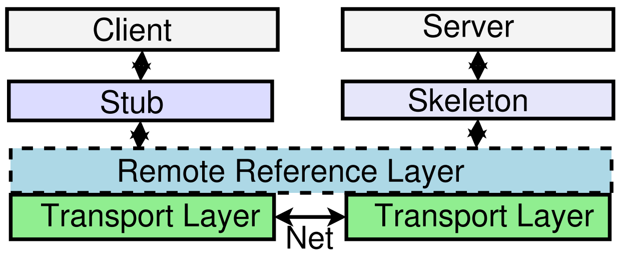


Figure 4.10: Java RMI layered architecture

an RMI call has been accomplished. This overhead can be decomposed into four categories: (1) `Network` or transport handling; (2) RMI `Protocol` processing, mainly stub and skeleton operation; (3) `Serialization`; and finally (4) `DGC`. Figure 4.11 shows a typical Sun JVM RMI call runtime's profile. It presents a 3 KB Object RMI send on an SCI network using SCIP as transport layer. Around a 86% of the overhead belongs to `Network`, 10% to the serialization process, 3.7% to `Protocol`, and a meager 0.1% to `DGC`.

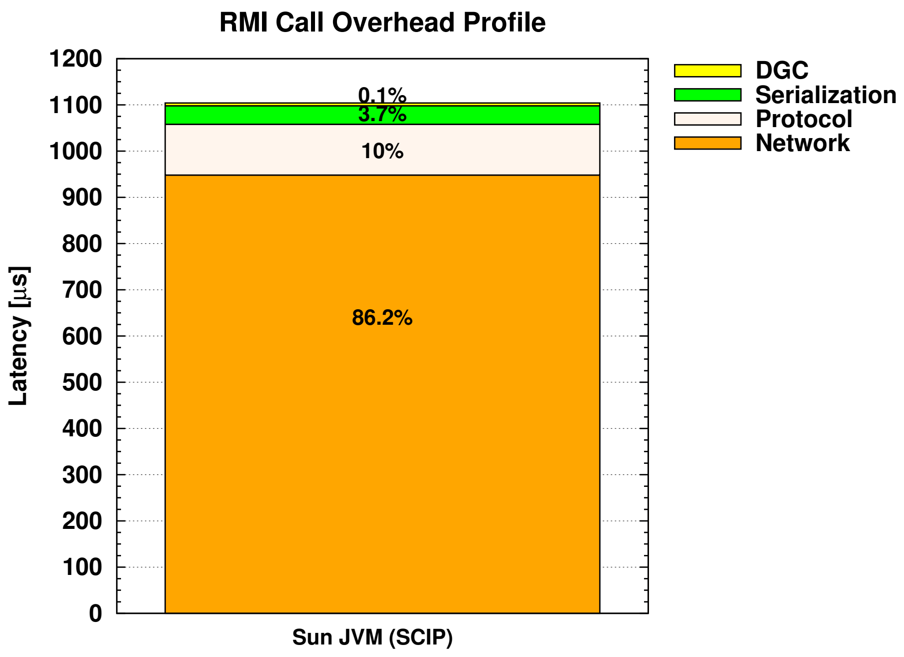


Figure 4.11: RMI send overhead of a 3 KB Object on SCI

The overhead incurred by the different phases of an RMI call has been considered in relative importance order to proceed with the optimization process. Thus, the proposed improvements are: (1) network overhead reduction through the use of the JFS support for high-speed networks; moreover, it is possible to optimize the RMI protocol to reduce the amount of data to be transferred and the buffering in the RMI network layer; (2) serialization avoidance for primitive data type arrays provided by JFS (see Subsection 3.1.1); and (3) object manipulation optimization, changing the protocol to reduce the information about objects that the RMI protocol includes in each communication, selecting the minimum amount of data required to successfully reconstruct a serialized object.

4.4.1. Transport Protocol Optimization

High Performance Sockets Support. The transport overhead can be reduced through the use of JFS, our high performance Java sockets implementation. JFS provides shared memory and high-speed networks support (Gigabit Ethernet, SCI, Myrinet and InfiniBand) on Java, and increases communication performance avoiding unnecessary copies and buffering as well as the serialization of primitive data type arrays (see Section 3.1).

Figure 4.12 shows an overview of the seven-layered communication stack for RMI-based Java parallel and distributed applications on shared memory, Gigabit Ethernet, SCI, Myrinet and InfiniBand. From bottom to top it can be seen the NIC layer, the NIC drivers, the native sockets and IP emulation libraries, JNI (which allows the Java access to native libraries), Java IO sockets implementations, Java RMI implementations, and RMI-based Java applications. The contributions of the Thesis depicted in Figure 4.12 are JFS and Opt RMI.

Reduction of Data Block Information. By default, all primitive data serialized for communication are inserted in data blocks. In order to separate data from different objects the data blocks are delimited by marks. To manage these blocks the Java RMI protocol uses a special write buffer, with some locks to protect the data integrity. The major goal of using this strategy for primitive data in serialization is to deal correctly with the versioning issue.

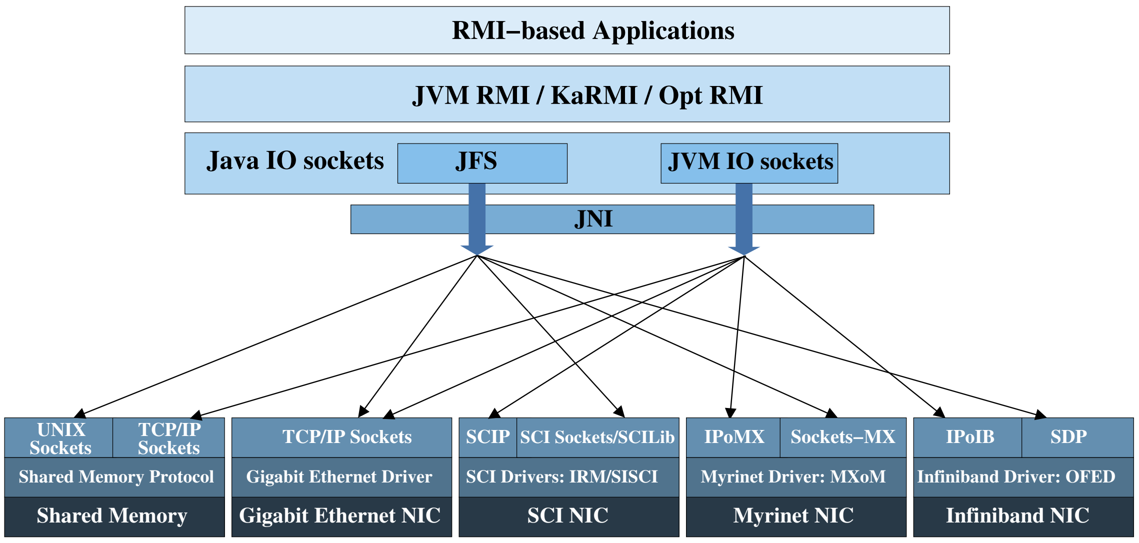


Figure 4.12: Java RMI communication stack on high-speed multi-core clusters

In the Opt RMI protocol the removal of some versioning information (improvement that will be described in Subsection 4.4.3) makes this data block strategy useless. Thus, the management of the data to be transferred has been simplified, without compromising the data integrity and correctness of the serialization and deserialization procedures.

4.4.2. Serialization Overhead Reduction

Native Array Serialization. In earlier versions of Java RMI, primitive data type arrays had to be serialized in an element-by-element and byte-by-byte approach. For example, in a double array each element had to be processed obtaining the long value that represents the bit layout of the double and then using a pair of operations per byte (of the long): a boolean `AND` and a right shift to process the next byte (except for the least significant one). However, in more recent JVM versions this issue has been partially solved, implementing the serialization of double and float arrays in native code, reducing significantly the serialization overhead.

In Opt RMI it has been implemented a generic method for array serialization which can process directly arrays of any primitive data type as they were byte arrays. This method relies on JFS and its serialization avoidance (see Subsection 3.1.1).

4.4.3. Object Manipulation Improvements

Versioning Information Reduction. For each object that is serialized, the Java RMI protocol serializes its description, including its type, version number and a whole recursive description of its attributes; i.e., if an attribute is an object, all its attributes have to be described through versioning. This is a costly process, because the version number of an object has to be calculated using reflection to obtain information about the class.

This versioning information is important to deserialize the object and reconstruct it on the receiving side, because sender and receiver can be running different versions of the JVM. Under the assumption of a single JVM and a shared class path (e.g., through a shared file system) the proposed solution is to send only the name of the class to which the object belongs, and reconstruct the object with the class description at the receiving JVM. As both sides use the same JVM and share the class path the interoperability is not compromised.

Class Annotation Reduction. Class annotations are used to indicate the locations (as Java Strings) from which the remote class loaders have to get the serialized object classes. This involves the use of specific URL class loaders. In a high-speed cluster environment with a single JVM, it is useful to avoid annotating classes from the `java.*` packages, as they can be loaded by the default class loader that guarantees that serialized and loaded classes are the same.

This change has been restricted to `java.*` packages to preserve interoperability, but it could also be applied to user classes. In fact, Opt RMI is interoperable, as it uses the optimized protocol only for intra-cluster communication whereas it relies on the JVM RMI for communicating with a machine outside the cluster. Thus, the ability to use multiple class loaders is not compromised.

Array Processing Improvements. The Java RMI protocol processes arrays as objects, with the consequent useless type checks and reflection operations. The proposed solution is to create a specific method to deal with array serialization (see Subsection 4.4.2), which provides several benefits in addition to the serialization avoidance. Thus, this method avoids the type checking and reflection overheads. Opt RMI implements an early array detection check, obtaining the array type using the `instanceof` operator against a list of the primitive data types. This list has been empirically obtained from the frequency of primitive data type appearance in high performance Java applications. Thus, this list (in the order: *double, integer, float, long, byte, Object, char, boolean*) optimizes the type casting compared to the default list (*Object, integer, byte, long, float, double, char, boolean*). If an object encapsulates a primitive data type array the proposed serialization method will handle this array when serializing recursively the members of the object.

4.5. Performance Evaluation of Java RMI for HPC

4.5.1. Experimental Configuration

The testbed consists of two dual-processor nodes (Pentium IV Xeon EMT64 at 3.2 GHz –a one-core processor– with hyper-threading disabled and 2 GB of memory) interconnected via SCI and Gigabit Ethernet. The SCI NIC is the same as described in Subsection 2.3.1, whereas the remaining configuration is specific to this evaluation. Thus, the Gigabit Ethernet NIC is a Marvell 88E8050 with an MTU of 1500 bytes. The OS is Linux CentOS 4.2 with compilers gcc 3.4.4 and Sun JDK 1.5.0_05. The SCI libraries are SCI Sockets 3.0.3, DIS 3.0.3 (it includes IRM, SISCI and SCILib), and SCIP 1.2.0. JFS has also been used. Finally, the three RMI implementations evaluated are the Sun JVM RMI of Sun JDK 1.5.0_05, KaRMI [76] from JavaParty 1.9.5, and our Opt RMI protocol.

In order to evaluate the RMI communication overhead, a Java RMI version of NetPIPE [102] has been developed due to the lack of Java RMI micro-benchmarks. Figure 4.13 shows the sequence diagram of the Java RMI ping and ping-pong tests, although the results considered in this section are only the half of the round trip time of the ping-pong test. The implementation issues of the Java RMI NetPIPE benchmark (design, implementation and JIT performance) are similar to the issues found in the development of the NetPIPE version for Java sockets presented in Subsection 3.2.1. The current section presents the performance of integer and Object arrays as they are common communication patterns in Java parallel and distributed applications. Moreover, the impact of the use of the Opt RMI protocol on RMI-based applications performance (see Subsection 4.5.3) has been analyzed.

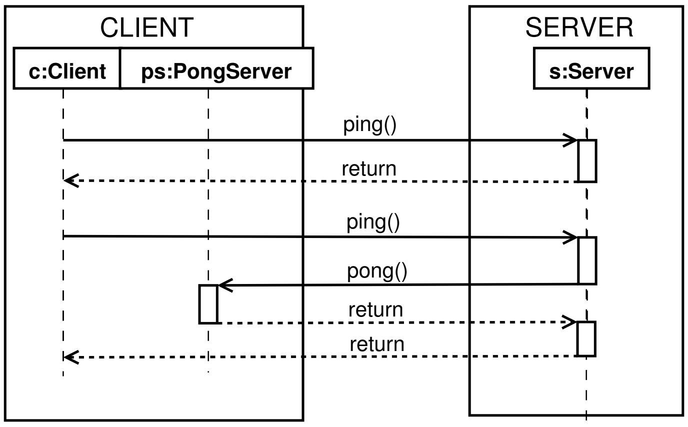


Figure 4.13: Ping and ping-pong RMI benchmark sequence diagram

4.5.2. Micro-benchmarking Results

Figure 4.14 compares the overhead of three RMI implementations, Sun JVM RMI, KaRMI (see Section 1.2) and Opt RMI sending a 3 KB Object on SCI, using both SCIP and JFS as underlying layers. The measures presented (see Section 4.4 and Figure 4.11) are the mean of ten calls, showing a small variance among them. The results show that Opt RMI reduces significantly the overhead of the other RMI libraries: Sun JVM RMI over SCIP (65%), KaRMI over SCIP (60%), Sun JVM RMI over JFS (53%) and KaRMI over JFS (45%). As can be seen, `Protocol` and especially `Network` latencies have decreased significantly. The explanation for these results is that sending an Object with several attributes (both Objects and primitive data types) can be more costly in Sun JVM RMI and KaRMI than in Opt RMI because of the overhead, in terms of data payload, imposed by the versioning information and the class annotation. In this case, sending this particular 3 KB Object involves a payload almost 3 times larger in Sun JVM RMI than in Opt RMI. Regarding KaRMI overhead, it is slightly lower than that of Sun JVM RMI. Although Opt RMI shows the lowest overheads, the Opt RMI `Serialization` presents lower performance than the Sun JVM one because the Opt RMI protocol has to obtain the information on how to deserialize the object.

Figures 4.15 and 4.16 present the results for RMI integer array communication. Regarding Gigabit Ethernet results, KaRMI shows the lowest latency for short messages (< 1 KB), but the highest communication overhead for larger messages. In

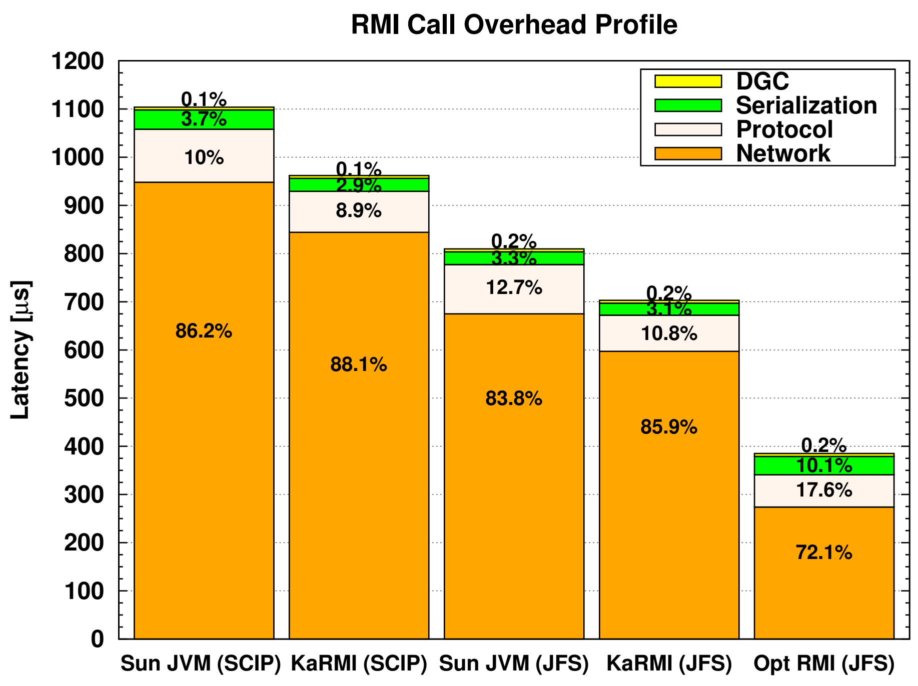


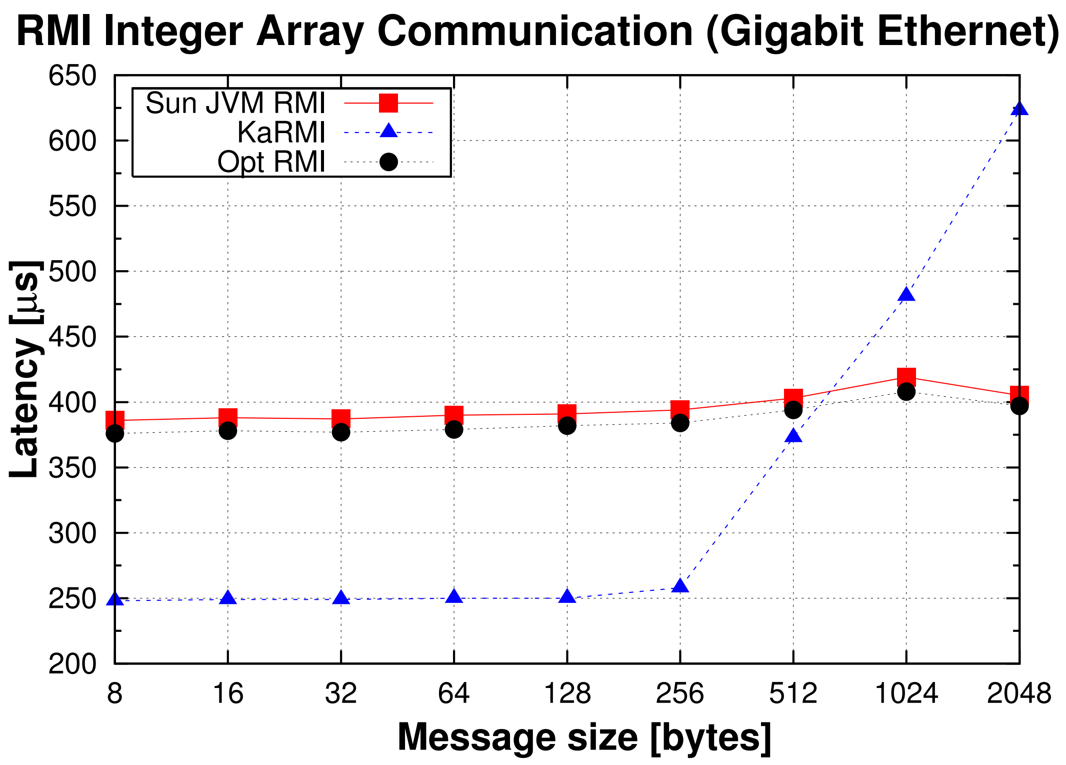
Figure 4.14: Performance results of RMI overhead of sending a 3 KB Object on SCI

this scenario the Opt RMI obtains slightly better results than Sun JVM RMI. With respect to SCI graphs, KaRMI and Sun JVM RMI on SCIP show the poorest results. However, substituting JVM sockets over SCIP by JFS as transport protocol improves the results significantly. In this case, KaRMI presents slightly better performance than Sun JVM RMI, for all message sizes. Regarding the RMI bandwidth on SCI, it can be seen that Sun JVM RMI and KaRMI performance drop for large messages ($>$ 256 KB) when using JFS due to a socket buffer size issue between the socket library and these RMI implementations. However, both Sun JVM RMI and KaRMI over JFS only obtain lower performance than over SCIP for [512 – 1536 KB] messages. Opt RMI does not present this performance degradation for large messages. In fact, it significantly outperforms Sun JVM RMI and KaRMI for large messages. Moreover, the Opt RMI presents significantly lower start-up latencies than Java RMI and KaRMI over SCIP and slightly lower latencies than Java RMI and KaRMI over JFS. Additionally, the Opt RMI obtains higher performance on SCI than on Gigabit Ethernet, where the interconnection network limits severely the performance (as presented in Sections 3.2.2 and 4.3.2). Finally, KaRMI also shows better performance on SCI than on Gigabit Ethernet, as it has been designed mainly for high-speed network communication.

Figure 4.17 presents results of communicating arrays of medium-size (3 KB) Objects. Figure 4.14 already presented the performance profile of the results obtained sending only one object. Regarding the benchmark results, the Opt RMI obtains the best performance for arrays of up to 8 Objects. Although its latency for a single Object is small, there is an important "extra" overhead for each Object processed, greater than for KaRMI and Sun JVM RMI overheads. Thus, for arrays from 8 Objects Sun JVM RMI and KaRMI obtain better performance than Opt RMI on Gigabit Ethernet and SCI, respectively. The Opt RMI "extra" overhead is caused by its inability to detect that the Object sent did not change since previous RMI calls. Nevertheless, the relatively poor results of Opt RMI sending Objects have little influence on the performance of Java HPC applications as object communication is not usual in these applications.

4.5.3. Performance Impact on RMI-based Applications

As Opt RMI reduces significantly the RMI overhead, the impact of its use in an RMI-based middleware has been evaluated. The RMI-based ProActive middleware [5, 77], as shown in [61], presents a poor scalability due to the significant penalty that RMI overhead imposes on the overall performance of ProActive applications. Thus, it has been selected for performance evaluation using RMI and Opt RMI, discarding KaRMI as it would require the reimplementation of ProActive using the KaRMI API. The benefits of using Opt RMI have been evaluated using two representative communication-intensive applications, MG and CG, from the ProActive NAS Parallel Benchmarks (NPB) [4]. MG is a 3D MultiGrid method with a Poisson solver algorithm, whereas CG solves a structured sparse linear system by the Conjugate Gradient method. Figure 4.18 shows MG and CG results using Sun JVM RMI and Opt RMI on SCI, relying on their default support on this network, SCIP and JFS, respectively. The high memory requirements of MG have prevented this benchmark from being run on a single node. The MG speedups have then been calculated using the runtime on two processors as reference. As can be seen in the figure, Opt RMI increases speedup up to 24% for MG and up to 157% for CG. Thus, Opt RMI allows to improve transparently the performance of RMI-based middleware and applications, especially on high-speed networks. Therefore, Opt RMI enables parallel and distributed RMI-based high-level programming with less overhead than Sun JVM RMI.



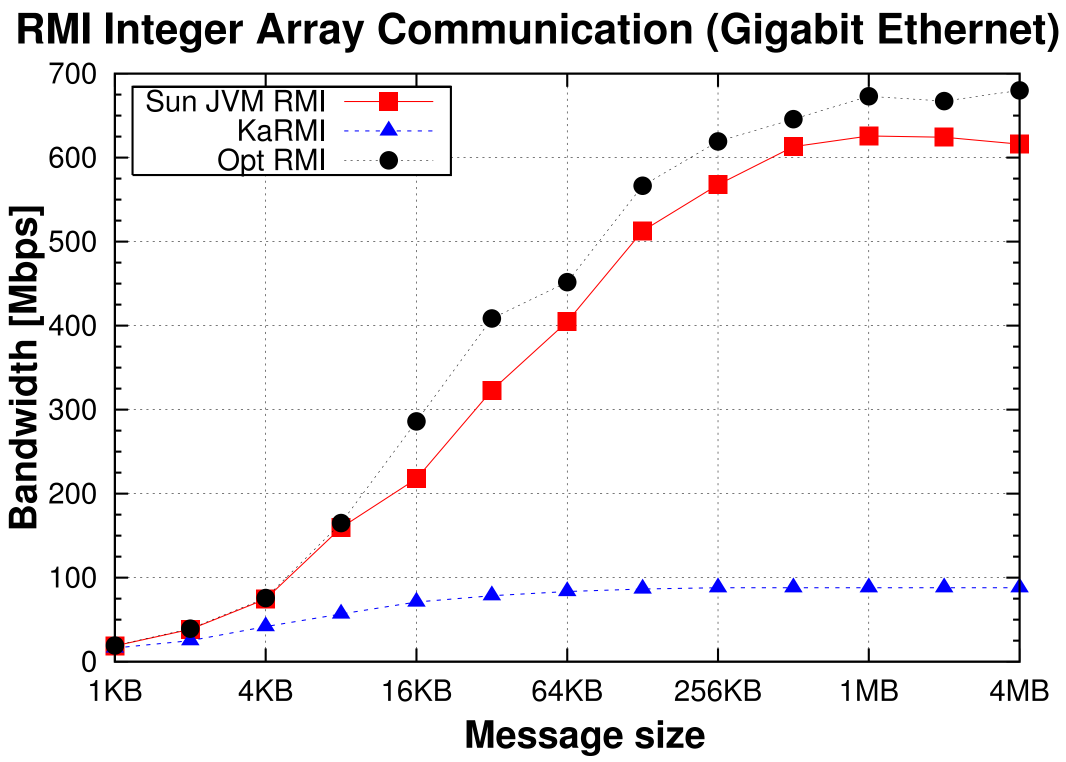
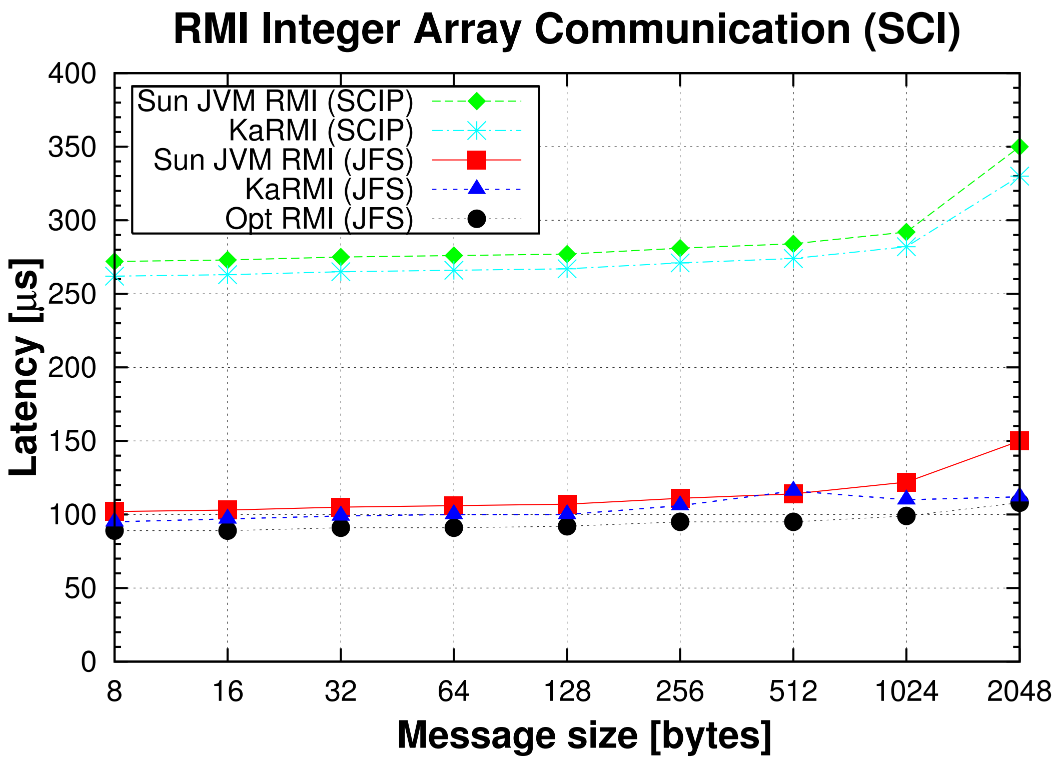


Figure 4.15: RMI integer array communication performance on Gigabit Ethernet



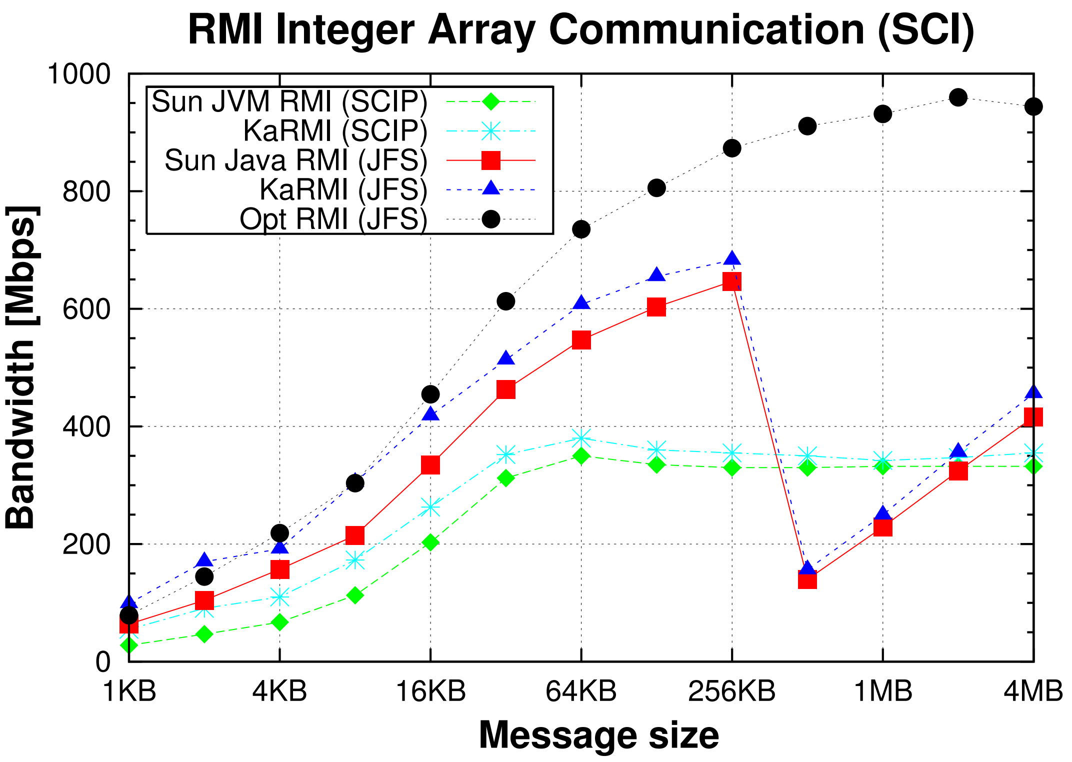
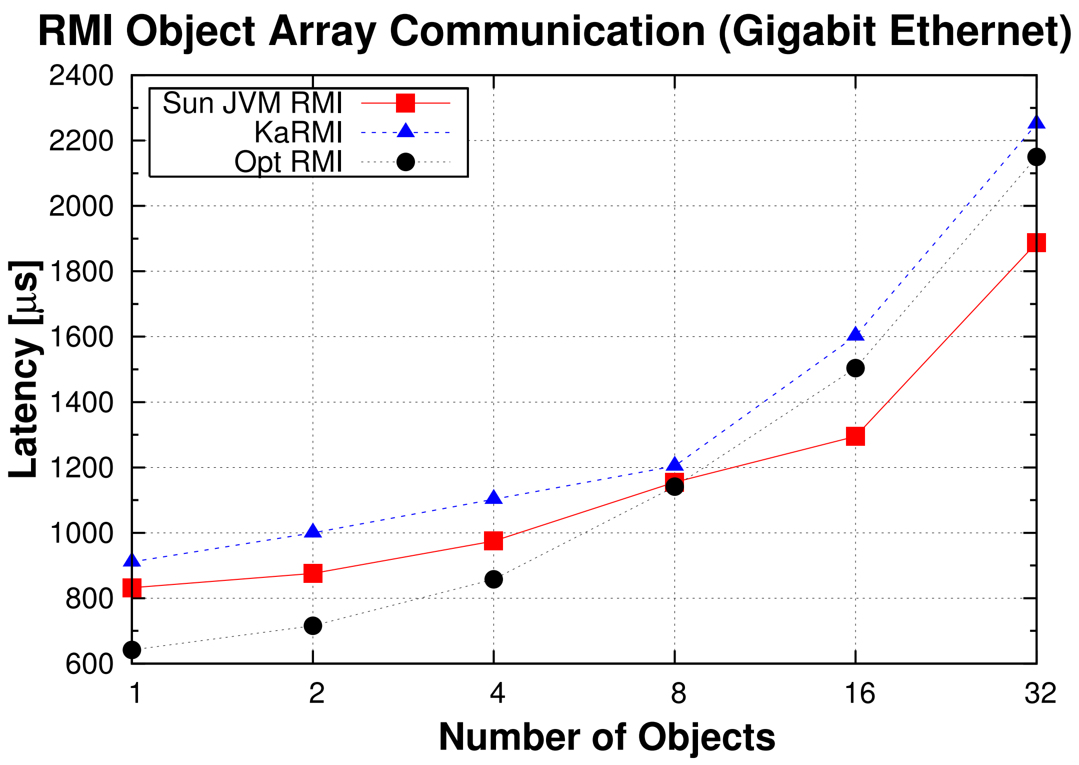


Figure 4.16: RMI integer array communication performance on SCI



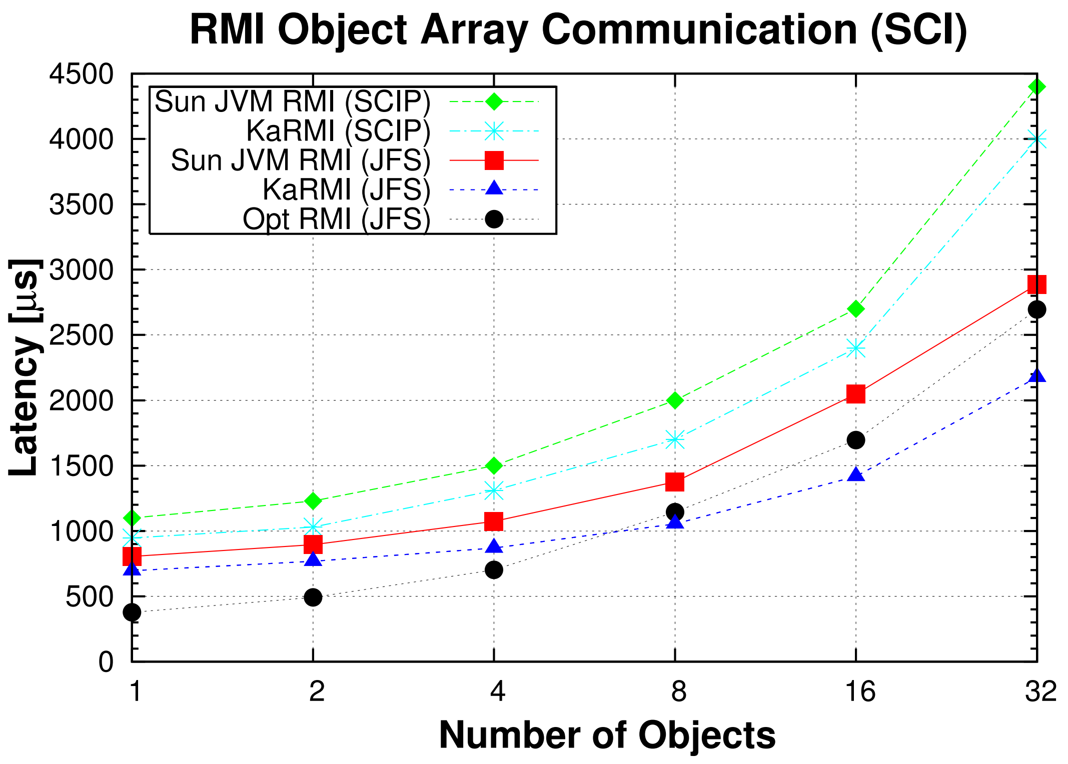


Figure 4.17: RMI object communication performance on Gigabit Ethernet and SCI

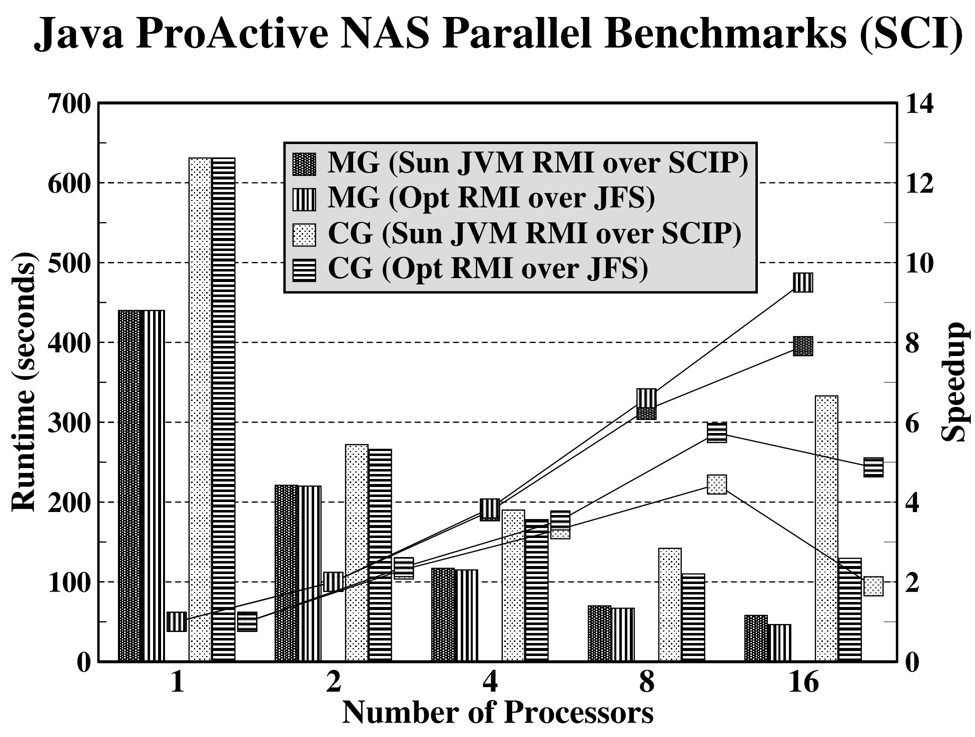


Figure 4.18: Performance of ProActive NPB MG and CG (Class C workload)

4.6. Chapter 4 Conclusions

This chapter has presented iodev, a low-level message-passing communication device. Among its main contributions iodev:

- Provides efficient non-blocking communication on Java IO sockets.
- Takes advantage of high-speed networks through the use of JFS, the high performance Java sockets implementation presented in Chapter 3.
- Avoids the use of buffers for the message data.
- Reduces the serialization overhead, especially for arrays of primitive data types.
- Implements a communication protocol that minimizes the start-up latency and maximizes the bandwidth.

The iodev communication device has been evaluated on shared memory, Gigabit Ethernet, SCI, Myrinet and InfiniBand scenarios, outperforming significantly the MPJ Express niodev device and mpiJava, obtaining similar performance to the MPJ Express mxdev device on Myrinet, and showing competitive results (and even higher on shared memory) compared to the native MPI libraries.

Additionally, this chapter has presented an efficient Java RMI implementation, a communication middleware also based on JFS. The solution proposed is transparent to the user (it does not need source code modification) and significantly improves performance. The RMI protocol optimizations have been focused on:

- Increasing the transport protocol performance through the use of the high performance Java sockets library JFS and reducing the information to be transferred.
- Providing a new method which deals with array communication, avoiding type checks and taking advantage of the reduction of the serialization overhead using JFS.
- Reducing the versioning and data block information as well as class annotations.

Experimental results have shown that the implemented RMI protocol optimizations reduce significantly the RMI call overhead, mainly on high-speed networks and for communication patterns frequently used in Java HPC applications, especially for arrays of primitive data types.

Chapter 5

Fast MPJ: Efficient Java Message-Passing Library

This chapter presents Fast MPJ (F-MPJ), a scalable and efficient Java message-passing library implemented on top of the low-level message-passing middleware iodev developed in the previous chapter. The efficient communication support provided by iodev and JFS (see Chapter 3) allows F-MPJ to take advantage of high-speed multi-core clusters (see Section 5.1). Moreover, the F-MPJ development has been focused on the use of scalable collective algorithms (see Section 5.2). Thus, F-MPJ has implemented several collective algorithms per primitive, allowing their selection at runtime, as will be shown in Section 5.3. F-MPJ has been evaluated on an InfiniBand multi-core cluster, outperforming significantly two representative MPJ libraries: MPJ Express and MPJ/Ibis (see Section 5.4). This evaluation consists of a micro-benchmarking of point-to-point and collective primitives, where F-MPJ increases performance up to 60 times, and an analysis of the impact of the use of F-MPJ+iodev+JFS on several kernel and application Java parallel benchmarks. The main conclusion is that F-MPJ improves significantly MPJ performance and scalability, thanks to the efficient point-to-point transfers and the collective primitives implementation. Section 5.5 presents a summary of the main contributions of this Chapter.

5.1. Efficient MPJ Communication

The current state of the art in Message-Passing in Java (MPJ) libraries has been presented in Section 1.3. Then, the kernel benchmarking on three representative MPJ libraries, mpiJava, MPJ/Ibis and MPJ Express, presented in Section 2.5 has shown that the performance of the pure Java implementations (MPJ/Ibis and MPJ Express) is significantly lower than native MPI, especially for communication-intensive codes, whereas the mpiJava wrapper library, although achieves higher performance than the pure Java libraries, presents associated drawbacks such as instability and lack of portability issues. The implementation efforts carried out in MPJ Express, using Java NIO sockets and direct byte buffers, and in MPJ/Ibis, using the high performance Ibis framework, have not bridged the gap with native MPI implementations.

In order to overcome this performance limitation in Java communication libraries, more efficient communication middleware in Java, such as JFS and iodev, have been implemented in this Thesis. This middleware serves as base to implement a high performance MPJ library, named Fast MPJ (F-MPJ). As the main motivation of this Thesis is the design of efficient Java communication libraries, the implementation of F-MPJ has been focused on increasing efficient point-to-point and collective primitives performance. Thus, advanced MPI functionalities, such as group, communicator and process topology management, have not been implemented (e.g., only the MPI.COMM_WORLD group has been defined). Moreover, no additional tools for development (e.g., logging generation and debugging facilities) and runtime environments are provided. F-MPJ implements the mpiJava 1.2 API [21], which has been selected as most of the MPJ codes publicly available use this API (instead of the JGF MPJ specification [22]). Furthermore, the two most active MPJ projects, mpiJava and MPJ Express, implement this API as well.

The implementation of the MPJ point-to-point primitives in F-MPJ is direct as iodev already provides the basic point-to-point primitives (see Section 4.2). Nevertheless, collective message-passing primitives require the development of algorithms that involve multiple point-to-point communications. MPJ application developers use collective primitives for performing standard data movements (e.g., Broadcast, Scatter and Gather) and basic computations among several processes (reductions). This greatly simplifies code development, enhancing programmers productivity together with MPJ programmability. Moreover, it relieves developers from communication optimization. Thus, collective algorithms must provide scalable performance, usually through overlapping communications in order to maximize the number of operations carried out in parallel. An unscalable algorithm can easily waste the performance provided by an efficient communication middleware.

Figure 5.1 presents an overview of the F-MPJ layered design on representative HPC hardware. From top to bottom, it can be seen that a message-passing application in Java (MPJ application) calls F-MPJ point-to-point and collective primitives. These primitives implement the MPJ communications API on top of the xxdev layer, whose implementation on Java IO sockets is iodev (see Section 4.2). The use of iodev, especially in combination with JFS, allows F-MPJ to take full advantage of shared memory and high-speed networks. The remaining layers in Figure 5.1 show the supported HPC hardware, already presented in Section 4.1 and more specifically in Figure 4.1.

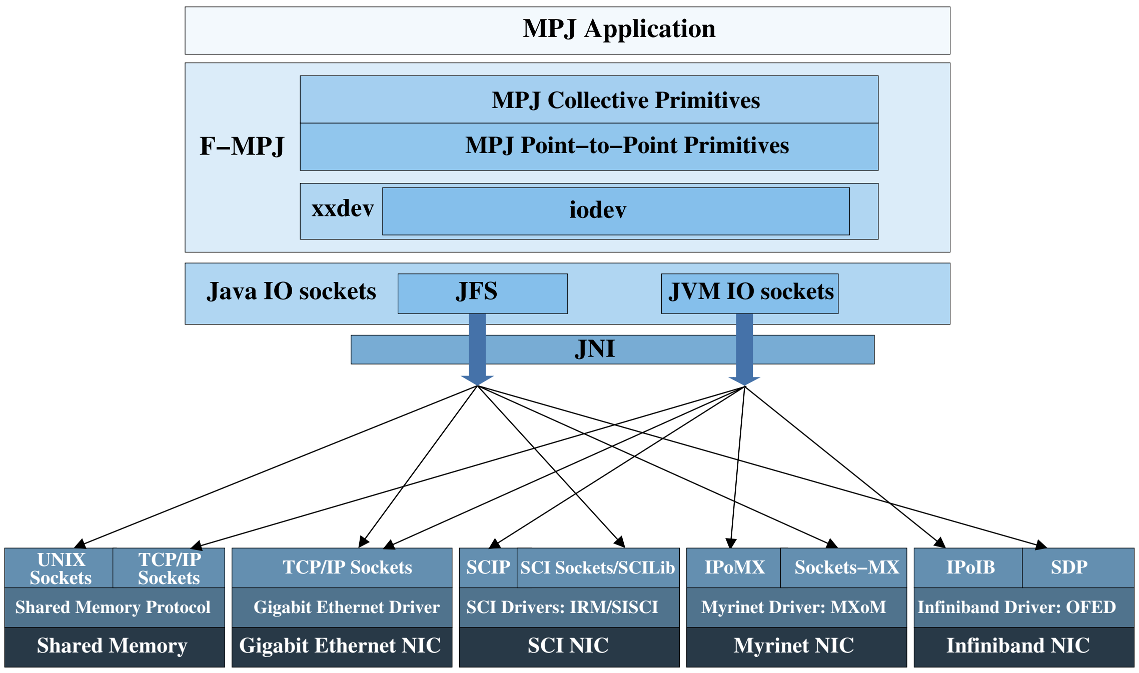


Figure 5.1: Overview of F-MPJ communication layers on HPC hardware

5.2. MPJ Collective Algorithms

The design, implementation and runtime selection of efficient collective communication operations have been extensively discussed in the context of native message-passing libraries [13, 23, 97, 103], but not in MPJ. Therefore, F-MPJ has tried to adapt the research in native libraries to MPJ. As far as we know, this is the first project in this sense, as up to now MPJ library developments have been focused on providing production-quality implementations of the full MPJ specification, rather than concentrating on developing scalable MPJ collective primitives.

The collective algorithms present in MPJ libraries can be classified in six types, namely Flat Tree (FT) or linear, Minimum-Spanning Tree (MST), Binomial Tree (BT), Four-ary Tree (Four-aryT), Bucket (BKT) or cyclic, and BiDirectional Exchange (BDE) or recursive doubling.

The simplest algorithm is FT, where all communications are performed sequentially. Figure 5.2 shows the pseudocode of the FT Broadcast using either blocking primitives (henceforth denoted as bFT) or exploiting non-blocking communications (henceforth nbFT) in order to overlap communications. As a general rule, valid for all collective algorithms, the use of non-blocking primitives avoids unnecessary waits and thus increases the scalability of the collective primitive. However, for the FT Broadcast only the send operation can be overlapped. The variables used in the pseudocode are also present in the following figures. Thus, `x` is the message, `root` is the root process, `me` is the rank of each parallel process, $\mathtt{p}_i$ the i-*th* process and `npes` is the number of processes used.

Figures 5.3 and 5.4 present MST pseudocode and operation for the Broadcast, which is initially invoked through `MSTBcast(x,root,0,npes-1)`. The parameters `left` and `right` indicate the indices of the left- and right-most processes in the current subtree. A variant of MST is BT, where at each step i (from 1 up to $\lceil log_2(npes)\rceil$) the process $\mathtt{p}_j$ communicates with the process $\mathtt{p}_{j+2^{i-1}}$. In a Four-ary Tree algorithm at each step i (from 1 up to $\lceil log_4(npes)\rceil$) the process $\mathtt{p}_j$ (node) communicates with its up to four associated processes (children).

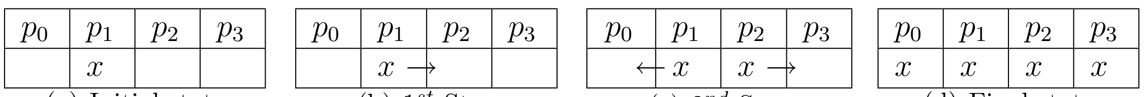
```
Method Bcast(x,root)(bFT)

if $me = root$ then
    for $i=0,...,npes-1$ do
        if $me != i$ then
            SEND (x,$p_i$);
else
    RECV (x,root);
```

```
Method Bcast(x,root)(nbFT)

if $me = root$ then
    for $i=0,...,npes-1$ do
        if $me != i$ then
            $sreq_i$ =ISEND (x,$p_i$);
else
    $rreq$ =IRECV (x,root);
    WAIT ($rreq$);
if $me = root$ then
    for $i=0,...,npes-1$ do
        if $me != i$ then
            WAIT ($sreq_i$);
```

Figure 5.2: FT Broadcast pseudocode

```
Method MSTBcast(x,root,left,right)

if $left = right$ then return;
mid = $\lfloor$(left+right)/2$\rfloor$;
if $root \leq mid$ then dest=right; else dest = left;
if $me = root$ then SEND (x,dest);
if $me = dest$ then RECV (x,root);
if $me \leq mid$ and $root \leq mid$ then MSTBCAST (x,root,left,mid);
else if $me \leq mid$ and $root > mid$ then MSTBCAST (x,dest,left,mid);
else if $me > mid$ and $root \leq mid$ then MSTBCAST (x,dest,mid+1,right);
else if $me > mid$ and $root > mid$ then MSTBCAST (x,root,mid+1,right);
```

Figure 5.3: MSTBcast pseudocode



(a) Initial state (b) 1^{st} Step (c) 2^{nd} Step (d) Final state

Figure 5.4: Minimum-spanning tree algorithm for Broadcast

Figures 5.5 (left) and 5.6 show BKTAllgather pseudocode and operation. In BKT all processes are organized like a ring and send at each step data to the process at their right. Thus, data eventually arrives to all nodes. F-MPJ implements an optimization by posting all `irecv` requests at BKT start-up. A subsequent synchronization (barrier) prevents early communication that incurs in buffering overhead when the `irecv` has not already been posted. The communications overlapping is achieved through `isend` calls. Finally, the algorithm waits for the completion of all requests. Figures 5.5 (right) and 5.7 present BDEAllgather pseudocode and operation, which requires that `npes` is a power of two. In BDE the message exchanged by each process pair is recursively doubled at each step until data eventually arrives to all nodes.

```
Method BKTAllgather($x$)
    prev = me - 1;
    if $prev < 0$ then prev= npes - 1;
    next = me + 1;
    if $next = npes$ then next=0;
    current_i = me;
    for $i=0,...,npes-2$ do
        current_i = current_i - 1;
        if $current\_i < 0$ then
        current_i=npes-1;
        $rreq_i$=IRECV ($x_{current\_i}$,prev);
    BARRIER ();
    current_i = me;
    for $i=0,...,npes-2$ do
        $sreq_i$=ISEND ($x_{current\_i}$,next);
        current_i = current_i - 1;
        if $current\_i < 0$ then
        current_i=npes-1;
    for $i=0,...,npes-2$ do
        WAIT ($sreq_i$);
        WAIT ($rreq_i$);
```

```
Method
BDEAllgather($x$,$left$,$right$)
    if $left = right$ then return;
    size = right − left + 1;
    mid = $\lfloor$(left+right)/2$\rfloor$;
    if $me \leq mid$ then
        partner = me + $\lfloor$size/2$\rfloor$;
    else
        partner = me − $\lfloor$size/2$\rfloor$;
    if $me \leq mid$ then
        BDEALLGATHER (x,left,mid);
    else
        BDEALLGATHER (x,mid+1,right);
    if $me \leq mid$ then
        SEND ($x_{left:mid}$,partner);
        RECV ($x_{mid+1:right}$,partner);
    else
        SEND ($x_{mid+1:right}$,partner);
        RECV ($x_{left:mid}$,partner);
```

Figure 5.5: BKTAllgather and BDEAllgather pseudocode

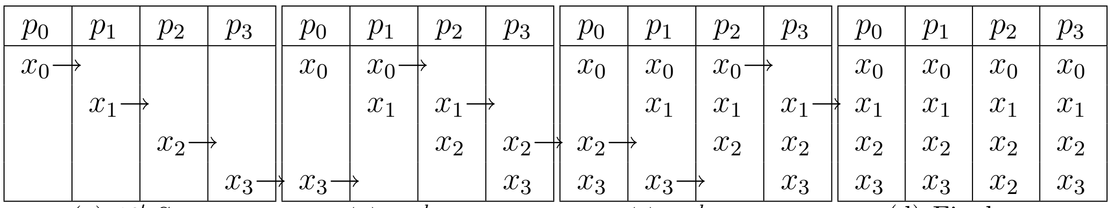

(a) 1st Step (b) 2nd Step (c) 3rd Step (d) Final state

Figure 5.6: Bucket algorithm for Allgather (BKTAllgather)

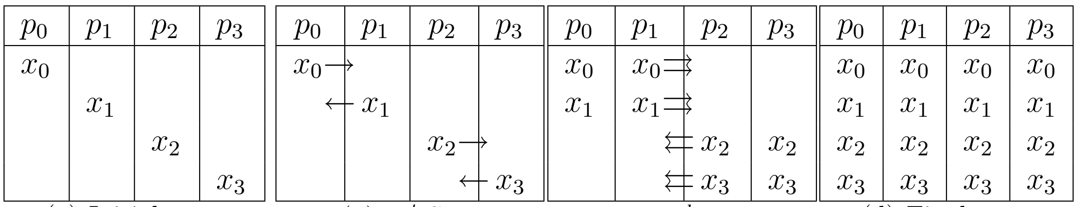

(a) Initial state (b) 1st Step (c) 2nd Step (d) Final state

Figure 5.7: Bidirectional exchange algorithm for Allgather (BDEAllgather). In the 2nd step, bidirectional exchanges occur between the two pairs of processes p_0 and p_2, and p_1 and p_3

5.3. F-MPJ Collective Primitives Algorithms

Although there is a wide variety of collective algorithms, as shown in the previous section, current MPJ libraries mainly resort to FT implementations, usually the only one primitive implementation provided. Nevertheless, F-MPJ defines up to three algorithms per primitive, selected at runtime. This section shows a comparative analysis of the collective primitives algorithms used in MPJ libraries.

Table 5.1 presents a complete list of the collective algorithms used in F-MPJ, MPJ Express and MPJ/Ibis. It can be seen that F-MPJ implements algorithms with usually higher scalability than MPJ Express and MPJ/Ibis collective primitives, taking advantage of communications overlapping. Thus, MPJ/Ibis only uses non-blocking communications in Alltoall and Alltoallv primitives, and MPJ Express resorts to bFT, an algorithm with poor scalability, for the Reduce. Moreover, MPJ Express uses a four-ary tree for Broadcast (Bcast) and Barrier, although with blocking communication. Nevertheless, in the remaining primitives MPJ Express takes advantage of non-blocking communications, except for Allreduce.

Table 5.1: Collective algorithms used in representative MPJ libraries (^{1}selected algorithm for short messages; ^{2}selected algorithm for long messages; ^{3}selectable algorithm for long messages and *npes* power of two)

Collective	**F-MPJ**	**MPJ Express**	**MPJ/Ibis**
Barrier	MST	nbFTGather+ bFour-aryTBcast	bFT
Bcast	MST^1 MSTScatter+BKTAllgather^2	bFour-aryT	BT
Scatter	MST^1 nbFT^2	nbFT	bFT
Scatterv	MST^1 nbFT^2	nbFT	bFT
Gather	MST^1 nbFT^2	nbFT	bFT
Gatherv	MST^1 nbFT^2	nbFT	bFT
Allgather	MSTGather+MSTBcast^1 BKT^2 BDE^3	nbFT	BKT(double ring)
Allgatherv	MSTGatherv+MSTBcast	nbFT	BKT
Alltoall	nbFT	nbFT	nbFT
Alltoallv	nbFT	nbFT	nbFT
Reduce	MST^1 BKTReduce_scatter+ MSTGather^2	bFT	BT(commutative) bFT(non commutative operation)
Allreduce	MSTReduce+MSTBcast^1 BKTReduce_scatter+ BKTAllgather^2 BDE^3	BT	BDE
Reduce_scatter	MSTReduce+MSTScatterv^1 BKT^2 BDE^3	bFTReduce+ nbFTScatterv	{BTReduce *or* bFTReduce}+ bFTScatterv
Scan	nbFT	nbFT	bFT

As F-MPJ implements up to three algorithms per primitive, the selection of the most suitable algorithm per collective primitive call is required. Thus, the selection depends on the message size, using the algorithms with the lowest latencies for short message communication and minimizing message buffering for long message communication. Table 5.1 indicates the selected algorithms using superscripts. The message size threshold used in this selection is configurable (32 KB by default) and independent for each primitive. The use of efficient communications and scalable algorithms in F-MPJ provides scalable MPJ performance, as will be assessed in the next section.

5.4. F-MPJ Performance Evaluation

5.4.1. Experimental Configuration

The testbed used for the performance evaluation of F-MPJ is the same as used for the iodev evaluation over InfiniBand (see Section 4.3), the Finis Terrae supercomputer [34]. In addition to the benchmarking of F-MPJ inter-node communication over InfiniBand, this system has also been used for the evaluation of F-MPJ inter-node communication (each node has 16 cores), and the hybrid inter-node/intra-node communications scenario. The evaluated MPJ libraries are F-MPJ with iodev and JFS 0.3.1, MPJ Express 0.27 and MPJ/Ibis 1.4. For comparison purposes, HP-MPI 2.2.5.1 as representative native MPI library has been selected as it demonstrates slightly better performance than MVAPICH2 1.0.2. HP-MPI uses two communication devices on the Finis Terrae: SHM, a special shared memory protocol for intra-node transfers, and IBV (OFED InfiniBand Verbs) for inter-node communication. The remaining configuration details are the same as those presented in Subsection 4.3.1.

The evaluation presented in this section consists of a micro-benchmarking of point-to-point primitives (Subsection 5.4.2) and collective communications (Subsection 5.4.3); and a benchmarking of two kernels and an application from the Java Grande Forum (JGF) Benchmark Suite [17] (Subsection 5.4.4).

5.4.2. Micro-benchmarking MPJ Point-to-point Primitives

In order to micro-benchmark F-MPJ primitives performance our own micro-benchmark suite [87] has been used as detailed in Section 2.2. Thus, the results shown are one half of the round trip time of a ping-pong test (point-to-point latency) or its corresponding bandwidth. Figure 5.8 shows point-to-point latencies and bandwidths for MPJ libraries communicating byte and double arrays, data structures frequently used in parallel applications, for intra-node (shared memory) and inter-node communication (InfiniBand). Moreover, the native MPI performance (i.e. HP-MPI results) is also shown for comparison purposes. The latency graphs serve to compare short message performance, whereas the bandwidth graphs are useful to compare long message performance. For purposes of clarity, the JNI array notation has been used in order to denote byte and double arrays in Java (B] and D], respectively).

The difference between this micro-benchmarking and that of Section 4.3 is the different level of the evaluation. Thus, Java low-level message-passing communication devices (niodev, mxdev and iodev) have been evaluated in the previous chapter, whereas the current section presents performance results from several MPJ implementations (F-MPJ, MPJ/Ibis and MPJ Express). The use of a higher level API presents an additional overhead on the underlying Java communication devices layer (i.e. MPJ-level performance is lower than that of the Java communication devices), which is characterized through the micro-benchmarking presented in this section.

F-MPJ, MPJ/Ibis and MPJ Express rely on different sockets implementations (JFS/Java IO sockets, Java IO sockets and Java NIO sockets, respectively), and thus it is not possible to compare directly the MPJ library processing overhead. However, as the sockets implementations share the same underlying layers, a fair comparison involves the analysis of the overhead of F-MPJ+JFS, MPJ/Ibis+Java IO sockets and MPJ Express+Java NIO sockets. The processing overhead of the MPJ libraries plus socket implementations can be estimated from Figure 5.8, where F-MPJ+JFS shows significantly lower overhead than MPJ Express+Java NIO and MPJ/Ibis+Java IO sockets, especially for short messages and double arrays (D]) communication. Regarding native performance, HP-MPI significantly outperforms MPJ libraries due to the low performance of the JVM on Linux IA64 (see Subsection 4.3.1) and the fact that the MPJ libraries are implemented using Java sockets instead of low-level communication protocols such as IBV or SHM.

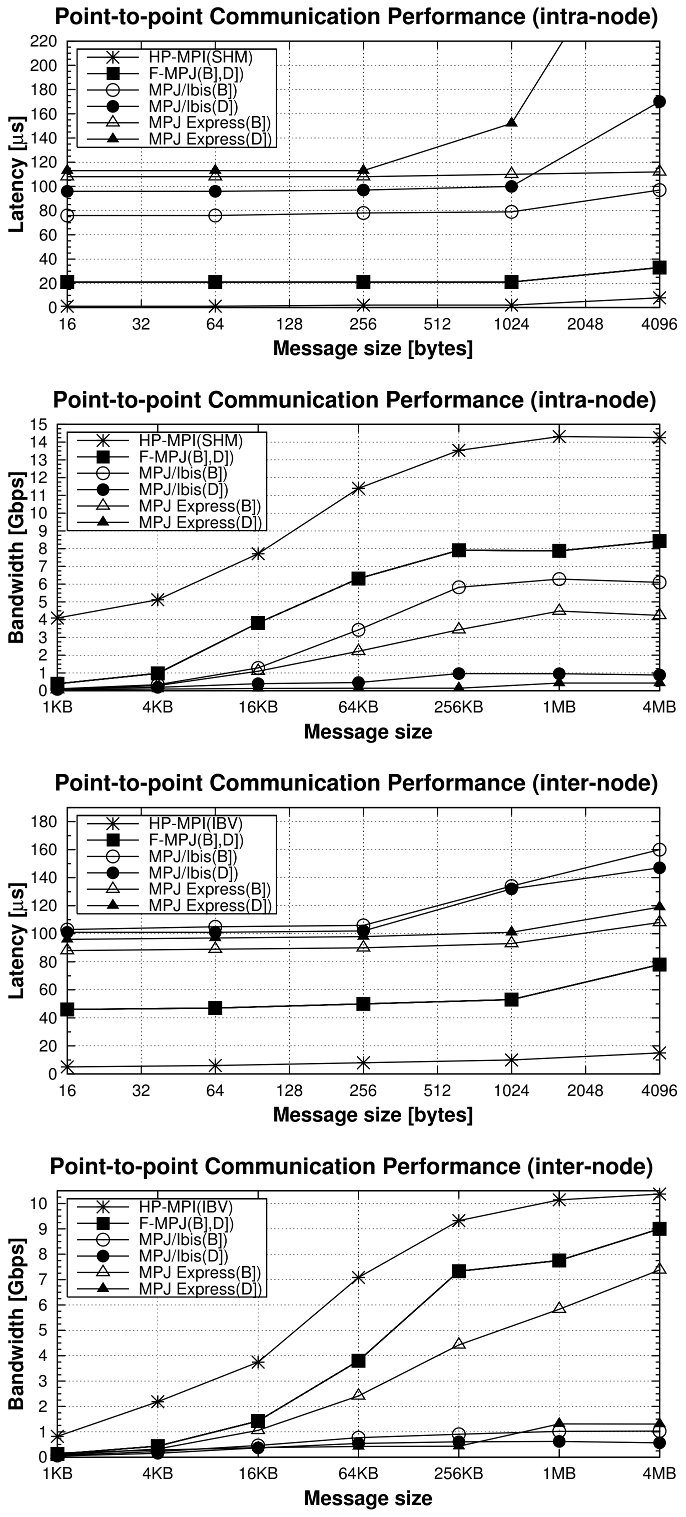


Figure 5.8: MPJ point-to-point primitives performance

F-MPJ handles D] transfers without serialization, obtaining the same results for B] and D] communication. As MPJ/Ibis and MPJ Express have to serialize double arrays, they present a significant performance penalty for D], especially for long messages. Thus, F-MPJ(D]) clearly outperforms MPJ/Ibis(D]) and MPJ Express(D]), showing up to 10 and 20 times higher performance, respectively. The impact of serialization overhead, the difference between D] and B] performance, is especially significant when the MPJ library obtains high B] bandwidths (MPJ/Ibis on intra-node and MPJ Express on inter-node). In these scenarios the serialization is the main performance bottleneck.

The byte array (B]) results are useful for evaluating the data transfer performance itself, without serialization overheads. In this scenario F-MPJ significantly outperforms MPJ Express and MPJ/Ibis, especially for short messages, thanks to its lower start-up latency. Regarding long message intra-node performance, F-MPJ outperforms MPJ/Ibis up to 27% and MPJ Express up to 50%. However, the results vary for inter-node transfers, where F-MPJ outperforms MPJ/Ibis up to 9 times and MPJ Express up to 40%. In these results the impact of the underlying communication middleware is significant. Thus, the high performance SDP library is only supported by F-MPJ and MPJ Express, which obtain significantly higher inter-node performance than MPJ/Ibis, which only supports the low performance IP emulation on InfiniBand (IPoIB). However, MPJ/Ibis outperforms MPJ Express when using the same underlying layer, the native TCP/IP sockets of the system, for intra-node transfers.

The observed point-to-point communication efficiency involves a significant improvement of F-MPJ collective primitives performance, as will be shown next.

5.4.3. Micro-benchmarking of MPJ Collective Primitives

The performance scalability of representative MPJ collective primitives has been evaluated on F-MPJ, MPJ/Ibis and MPJ Express. Figure 5.9 presents the aggregated bandwidth for Broadcast and sum reduction operations, both for the communication of short (using 1 KB as representative message size) and long (using 1 MB as representative message size) double arrays. The Broadcast and Reduce primitives have been selected as representative data movement and computational primitives,

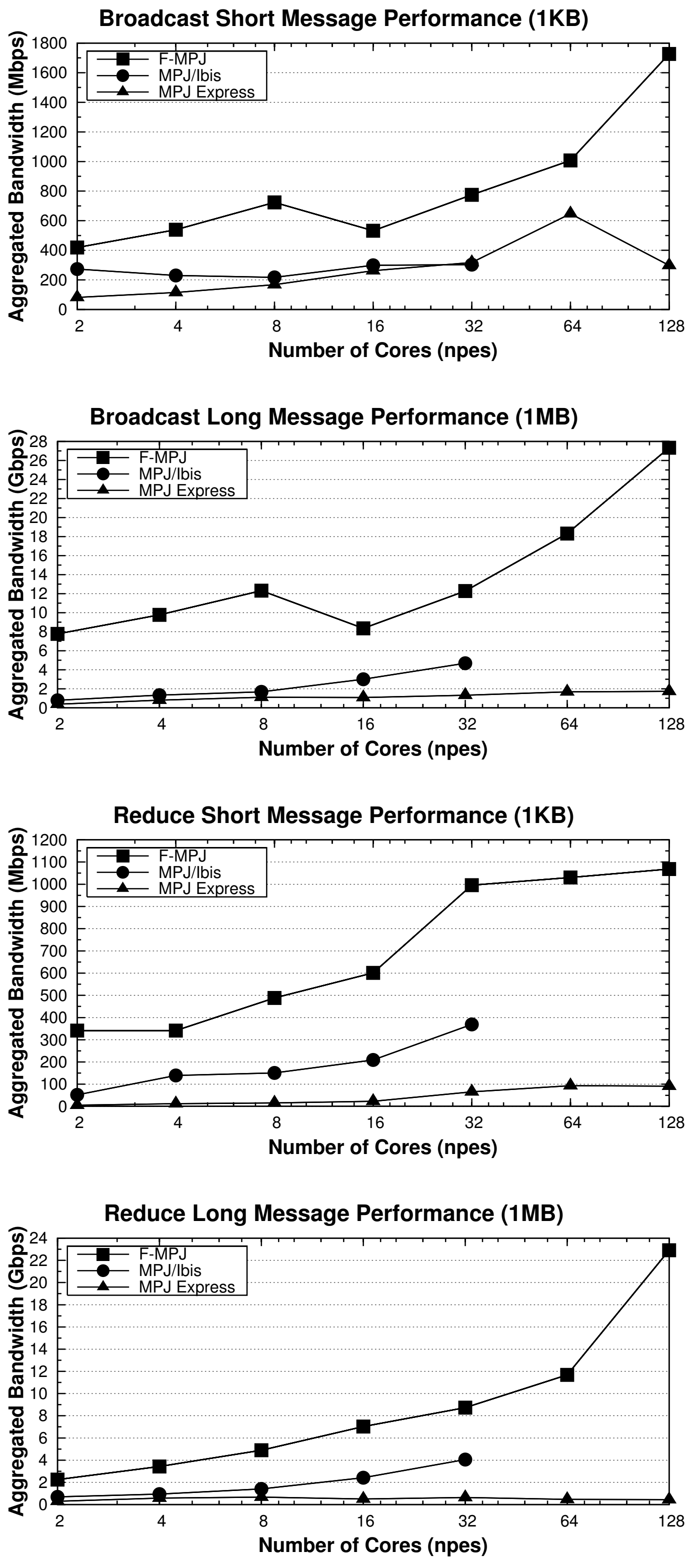


Figure 5.9: MPJ collective primitives performance

respectively. The aggregated bandwidth metric has been selected as it takes into account the global amount of data transferred, $message\_size * (npes - 1)$ for both collectives. The results have been obtained with a maximum of 8 cores per node as this configuration has shown the best performance. Thus, from now on the number of nodes used is $\lceil npes/8 \rceil$. MPJ/Ibis could not be run in our testbed using more than 32 cores due to an Ibis runtime initialization error.

The presented results (Figure 5.9) show that F-MPJ significantly outperforms MPJ Express and MPJ/Ibis. Regarding Broadcast, F-MPJ provides up to 5.8 and 16 times performance increases for short and long messages, respectively. The improvement of the F-MPJ Reduce is up to 60 and 50 times for short and long messages, respectively. The maximum performance increases of F-MPJ have been obtained from the comparison against MPJ Express results. F-MPJ shows scalable performance for both collectives, obtaining usually the highest performance increases on 128 cores. The significant performance improvement of F-MPJ for long messages is mainly due to the serialization avoidance. Moreover, F-MPJ takes significant advantage of intra-node communication (up to 8 cores), especially for the Broadcast. The lowest performance, especially for the Reduce, has been obtained by MPJ Express, whereas MPJ/Ibis results are between F-MPJ and MPJ Express results, although much closer to the latter.

In conclusion, F-MPJ significantly improves MPJ collectives performance due to its efficient intra-node and inter-node point-to-point communication, the serialization avoidance and the use of scalable algorithms (see Table 5.1) based on non-blocking communications overlapping.

5.4.4. MPJ Kernel and Application Benchmarking

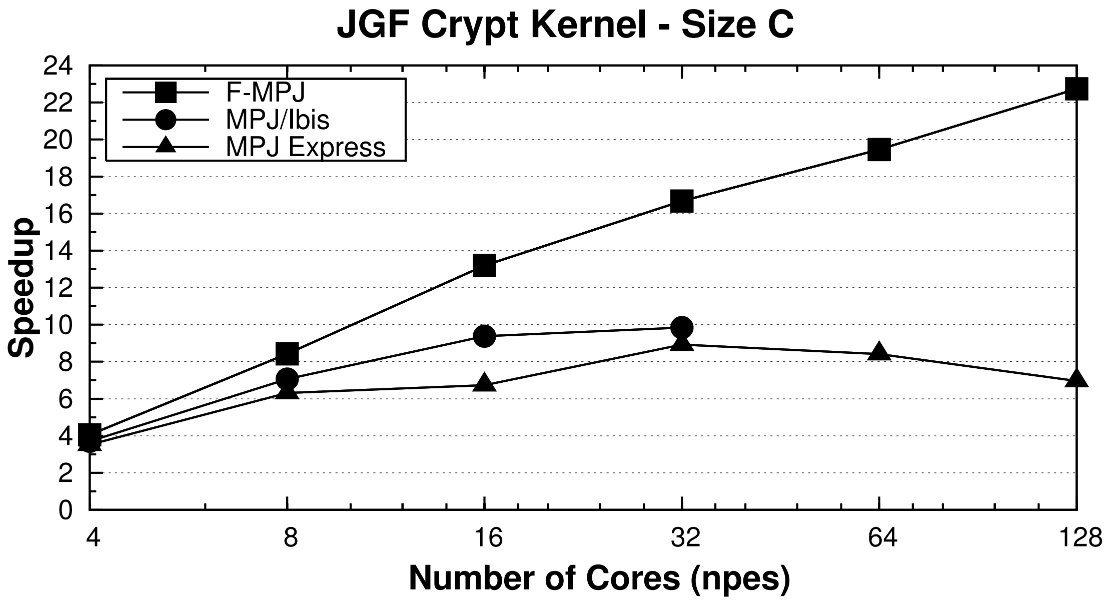
The impact of the use of F-MPJ on representative MPJ benchmarks is analyzed in this subsection. Two kernels and one application from the JGF Benchmark Suite have been evaluated: Crypt, an encryption and decryption kernel; LUFact, an LU factorization kernel; and MolDyn, a molecular dynamics N-body parallel simulation application. These MPJ codes have been selected as they show very poor scalability with MPJ/Ibis and MPJ Express. In fact, Section 2.5 presented a performance evaluation of JGF kernels, obtaining speedups with 32 processes below

4 for LUFact (see Figure 2.5) and below 8 for Crypt (results not shown in Section 2.5 for conciseness purposes) using up to 32 processors. Hence, these are target codes for the evaluation of F-MPJ performance and scalability improvement.

Figure 5.10 presents Crypt and LUFact speedups. Regarding Crypt, F-MPJ clearly outperforms MPJ/Ibis and MPJ Express, up to 330%, in a scenario where the data transfers (byte arrays) do not involve serialization. Thus, both MPJ/Ibis and MPJ Express take advantage of the use of up to 32 cores. LUFact broadcasts double and integer arrays for each iteration of the factorization method. Therefore, the serialization overhead is important for this code. Thus, the use of F-MPJ has a higher impact on performance improvement than for Crypt. Figure 5.10 (bottom) shows that F-MPJ significantly outperforms MPJ/Ibis and MPJ Express for LUFact, up to eight times. This performance increase is due to the use of scalable algorithms and the serialization avoidance. Furthermore, F-MPJ presents its best results on 128 cores, whereas MPJ/Ibis and MPJ Express obtain their best performance on 16 and 8 cores, respectively.

The MolDyn application consists of six Allreduce sum operations for each iteration of the simulation. The transferred data are integer and doubles arrays, so F-MPJ can avoid serialization overhead. For its evaluation an enlarged size C version has been used (it processes a multidimensional double array of $18x18x18x4$ values). Figure 5.11 presents MPJ speedups, where F-MPJ outperforms MPJ/Ibis and MPJ Express up to 3.5 times. This application presents higher speedups than the kernels of Figure 5.10 as it is a less communication-intensive code, and the three libraries use scalable Allreduce algorithms (see Table 5.1). However, the serialization overhead negatively affects MPJ/Ibis and MPJ Express MolDyn performance.

The use of F-MPJ increases significantly MPJ kernels and applications performance, especially for communication-intensive codes. Moreover, the scalable F-MPJ performance allows MPJ codes to take advantage of the use of a large number of cores (up to 128 in our experiments), a significantly higher value than for MPJ/Ibis and MPJ Express.



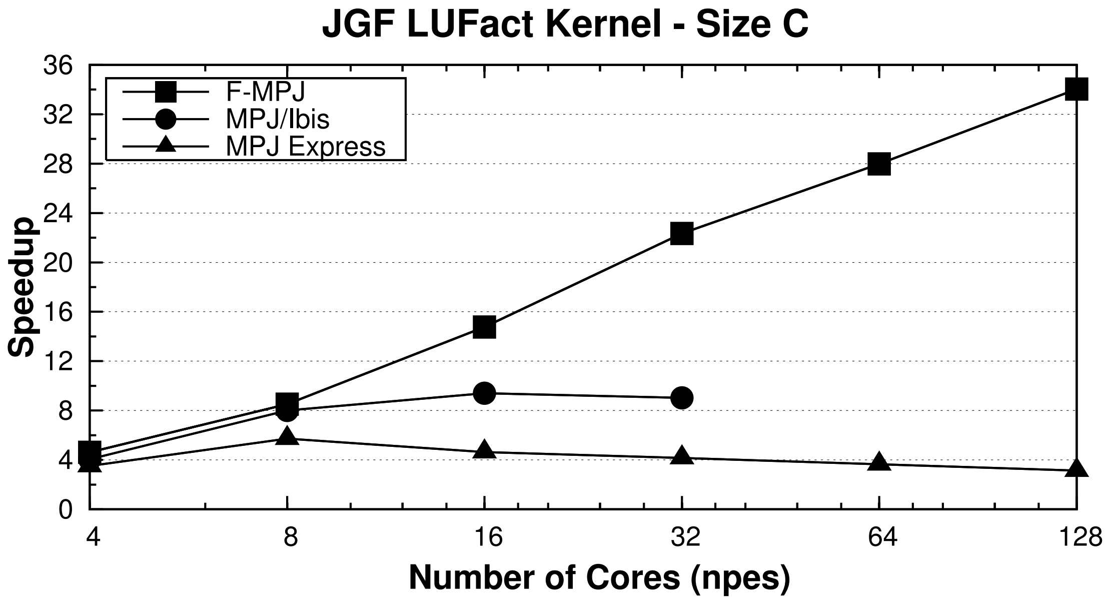


Figure 5.10: Speedups of Crypt and LUFact JGF kernels

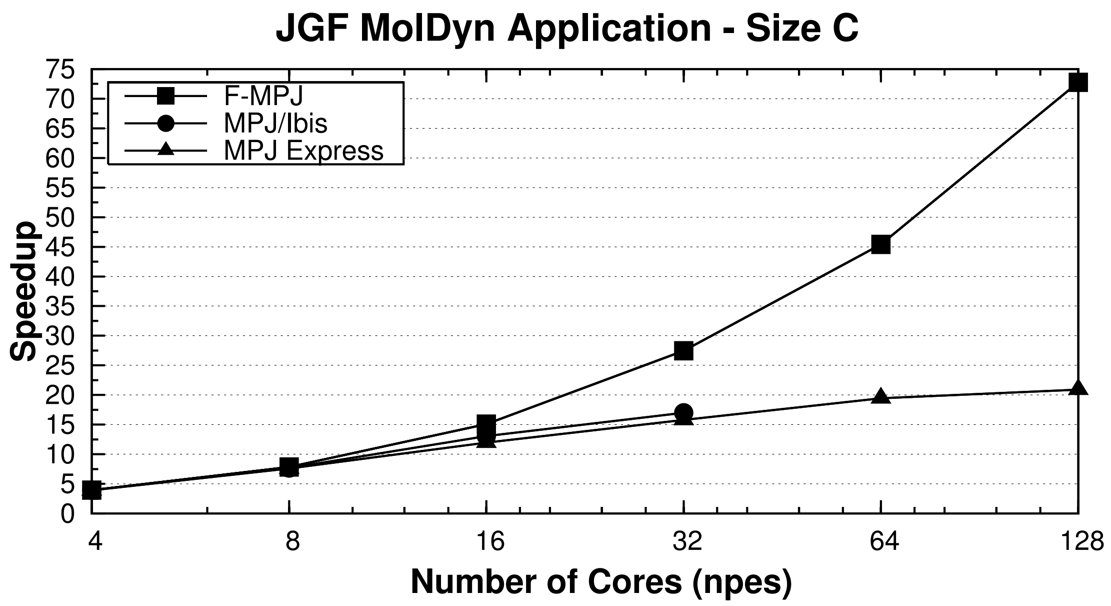


Figure 5.11: Speedups of JGF MolDyn application

5.5. Chapter 5 Conclusions

This chapter has presented F-MPJ, a scalable and efficient Java message-passing library. The increasing interest in Java parallel solutions on multi-core clusters demands efficient communication middleware. F-MPJ pursues to satisfy this need obtaining scalable Java performance in parallel systems. Among its main contributions, F-MPJ:

- Takes advantage of the efficient integration of iodev (see Section 4.2) and JFS (see Section 3.1) into the implementation of the MPJ primitives, obtaining efficient non-blocking communication and high-speed multi-core clusters support.

- Increases MPJ communications performance (obtains lower start-up latencies and higher aggregated bandwidths) through an extensive use of communications overlapping.

- Implements several algorithms per collective primitive, allowing their selection at runtime.

F-MPJ has been evaluated on an InfiniBand multi-core cluster, outperforming significantly two representative MPJ libraries: MPJ Express and MPJ/Ibis. Thus, the micro-benchmarking results showed a performance increase up to 60 times for F-MPJ. Moreover, the subsequent kernels and application benchmarking obtained speedup increases of up to seven times for F-MPJ on 128 cores, depending on the communication intensiveness of the analyzed MPJ benchmarks. F-MPJ improves significantly MPJ performance and scalability, allowing Java message-passing codes that previously increased their speedups only up to 8-16 cores to take advantage of the use of 128 cores.

Chapter 6

Implementation and Performance Evaluation of Efficient MPJ Benchmarks

This chapter presents the development of efficient Message-Passing in Java (MPJ) parallel benchmarks and the analysis of their performance on two representative clusters. These tasks have also been useful for gathering good programming practices for performance in Java for HPC, whose impact on the overall results can be significant. The codes selected for being developed and evaluated have been the NAS Parallel Benchmarks (NPB) [71], the standard suite for parallel benchmarking, due to their representativeness and usefulness. The availability of different Java parallel programming libraries, such as MPJ libraries and RMI-based middleware, such as ProActive [5, 77], eases Java's adoption for HPC. In this scenario, a comparative evaluation of Java for parallel computing against native solutions is required in order to assess its benefits and disadvantages. Thus, an implementation of the NPB is provided for Message-Passing in Java, named the NPB-MPJ suite. The design, implementation and performance optimization of this suite are covered in detail in this chapter.

The structure of this chapter is as follows: Section 6.1 introduces the related work in Java NPB implementations. Section 6.2 describes the design, implementation and optimization of NPB-MPJ, our NPB implementation for MPJ. Moreover, the study of the impact on performance of the optimization techniques used in NPB-MPJ, from which Java HPC applications can potentially benefit, is also discussed. Comprehensive benchmark results from an NPB-MPJ evaluation on two representative multi-core clusters, with InfiniBand and Gigabit Ethernet interconnection networks, are shown in Section 6.3. As JVM technology and MPJ libraries are actively evolving it is important to present an up-to-date evaluation of their performance. Moreover, additional NPB results from different Java and native parallel libraries (Java threads, ProActive, MPI and OpenMP) are also shown for comparison purposes. The main conclusion obtained from this comparative performance evaluation is that MPJ codes can outperform MPI, OpenMP and Java threads scalability. Therefore, MPJ libraries, especially F-MPJ, are an alternative to native languages (C/Fortran) for parallel programming on multi-core systems as it is possible to obtain scalable performance while taking advantage of the Java features. This conclusion together with a summary of the main contributions of this Chapter is presented in Section 6.4.

6.1. NAS Parallel Benchmarks in Java

The NAS Parallel Benchmarks (NPB) [6, 71] consist of a set of kernels and pseudo-applications taken primarily from Computational Fluid Dynamics (CFD) applications. These benchmarks reflect different kinds of computation and communication patterns that are important across a wide range of applications. Therefore, they are the de facto standard in parallel performance benchmarking.

The NPB suite consists of the CG, EP, FT, IS, MG and DT kernels. Among the pseudo-applications, SP has been selected as representative benchmark (due to its higher scalability than the other pseudo-applications) for its implementation and evaluation in this Chapter. The CG kernel is a sparse iterative solver that tests communications performance in sparse matrix-vector multiplications. The EP kernel is an embarrassingly parallel code without significant communications that assesses the floating point performance of the system. The FT kernel performs a series of 3-D FFTs on a 3-D mesh, and tests aggregated communication performance. The IS kernel is a large integer sort that evaluates both integer computation performance and the aggregated communication throughput. MG is a simplified multigrid kernel that communicates both with contiguous and remote processes (e.g., in a multi-core cluster MG performs both intra- and inter-node communications). The DT (Data Traffic) kernel operates with graphs and evaluates communication throughput. The SP (Scalar Pentadiagonal) pseudo-application is a simulated CFD application. This wide range of implemented benchmarks assures a broad performance analysis. Table 6.1 summarizes the main characteristics of these benchmarks, together with the number of Source Lines Of Code (SLOC) of their implementation with Message-Passing in Java (our NPB-MPJ suite).

There are implementations of the NPB for the main parallel programming languages and libraries, such as MPI (from now on NPB-MPI), OpenMP (from now on NPB-OMP), High Performance Fortran (HPF), Unified Parallel C (UPC), and Co-Array Fortran. Regarding Java, currently there are three NPB implementations, apart from our NPB-MPJ suite, namely the multithreaded [71] (from now on NPB-JAV), the ProActive [3, 4] (from now on NPB-PA), and the Titanium [27] implementations. However, these three developments present several drawbacks in order to evaluate the capabilities of Java for parallel computing. NPB-JAV is the Java multi-threaded implementation of the NPB using a master-slave paradigm. It is limited to shared memory systems and thus its scalability is lower than the one provided by distributed memory architectures. With respect to NPB-PA, although it relies on a distributed memory programming model, the use of an inefficient communication middleware such as RMI limits its performance scalability. Titanium is an explicitly parallel dialect of Java, so its portability is quite limited. Moreover, as the reference implementation of the NPB is written in MPI, NPB-MPJ allows the comparison of Java and native languages within the target programming model of the NPB, the message-passing paradigm.

Therefore, MPJ is a highly interesting option to implement the NPB suite in Java. The use of MPJ allows the comparative analysis of the existing MPJ libraries and the comparison between Java and native message-passing performance, using NPB-MPJ and NPB-MPI, respectively. Moreover, it also serves to compare with the performance of different Java parallel libraries that have implemented the NPB, such as Java multithreading (NPB-JAV) and ProActive (NPB-PA).

Previous efforts on the implementation of NPB for MPJ have been associated with the development of MPJ libraries. Thus, JavaMPI [37] included EP and IS kernels, the ones with the lowest number of Source Lines Of Code (SLOC). Then, the CG kernel was implemented for MPJava [78]. Finally, P2P-MPI [36] also implemented the EP and IS kernels.

Another motivation for the implementation of the NPB-MPJ suite is the current lack of parallel benchmarks in Java. The most noticeable related project is the Java Grande Forum (JGF) benchmark suite [17] that consists of: (1) sequential benchmarks, (2) multithreaded codes, (3) MPJ benchmarks, and (4) the language comparison version, which is a subset of the sequential benchmarks translated into C. However, the JGF benchmark suite does not provide the MPI counterparts of the MPJ codes, allowing only the comparison among MPJ libraries and Java threads. Moreover, its codes are less representative of HPC kernels and applications than those of the NPB suite.

NPB-MPJ enhances these previous efforts implementing an extensive number of benchmarks, shown in Table 6.1. An approximate idea of the implementation effort carried out in NPB-MPJ can be estimated using the SLOC metric. NPB-MPJ has, as a whole, approximately 11,000 SLOC. Moreover, NPB-MPJ uses the most extended Java message-passing API, the mpiJava API (used by mpiJava, MPJ Express and F-MPJ, see Table 1.2). Finally, it provides support for automating the benchmarks execution and the graphs and performance reports generation. NPB-MPJ has significantly increased the availability of standard Java parallel benchmarks.

Table 6.1: NPB-MPJ Benchmarks Description

Name	Operation	Communication intensiveness	SLOC	Kernel	Applic.
CG	Conjugate Gradient	Medium	1000	✓	
EP	Embarrassingly Parallel	Low	350	✓	
FT	Fourier Transformation	High	1700	✓	
IS	Integer Sort	High	700	✓	
MG	Multi-Grid	High	2000	✓	
DT	Data Traffic	High	1000	✓	
SP	Scalar Pentadiagonal	Medium	4300		✓

6.2. NPB-MPJ: NAS Parallel Benchmarks for MPJ

NPB-MPJ is the implementation of the standard NPB suite for MPJ performance evaluation. This suite facilitates: (1) the comparison among MPJ implementations; (2) the evaluation of MPJ against other Java parallel libraries (e.g., RMI-based or Java threads); (3) the assessment of the performance gap between MPJ and MPI; and finally, (4) the development of efficient Java code as it provides a compilation of some good programming practices for high performance in Java. This section presents the design of NPB-MPJ, the implementation of an initial version and the subsequent performance optimization.

6.2.1. NPB-MPJ Design

The NPB-MPJ design is based on the NPB-MPI, which are all MPI Fortran codes except IS and DT, which are MPI C kernels. The use of the message-passing programming model determines that NPB-MPJ and NPB-MPI share several characteristics, and thus NPB-MPJ design has followed the SPMD (Single Program Multiple Data) paradigm, the workload distribution among the processes and the communication primitives used in NPB-MPI. Moreover, the NPB-JAV implementation (with Java threads) has also served as basis for the NPB-MPJ design. Although the master-slave paradigm, used in NPB-JAV, has not been selected for NPB-MPJ, its Java-specific solutions, such as the complex numbers support or the timing methods, have been useful for NPB-MPJ.

An important issue tackled in NPB-MPJ has been the choice between a “pure” object-oriented design or an imperative approach through the use of “plain objects”. In order to maximize NPB-MPJ performance, the “plain objects” design has been chosen as it reduces the overhead of the “pure” object-oriented design (up to 95%). Thus, each benchmark uses only one object instead of defining an object per each element of the problem domain (e.g., a data structure with specific operations and complex numbers). The overhead derived from an intensive use of object orientation in numerical codes has been recognized as significant in the related literature [66]. An example of this design decision is the complex numbers support in NPB-MPJ. As Java does not have a complex number primitive datatype and the NPB use them thoroughly, NPB-MPJ has implemented its own support, similar to the one implemented in NPB-JAV. Thus, a complex number is implemented as a two-element array (real and imaginary parts). This approach presents less overhead than the implementation of complex number objects, which trades off a clear design and the encapsulation features for higher access overhead, especially when dealing with arrays of complex number objects.

6.2.2. NPB-MPJ Implementation

The NPB-MPJ suite consists of the CG, EP, FT, IS, MG and DT kernels and the SP pseudo-application. A brief description of these benchmarks has been presented in Table 6.1. The implementation of the benchmarks has presented some common issues, shared among all the codes, such as the handling of the Java arrays.

The NPB handle arrays of up to five dimensions. In native languages it is possible to define multidimensional arrays whose memory space is contiguous, unlike Java, where an n-dimensional array is defined as an array of $n-1$ dimensional arrays. The main drawback for NPB-MPJ is the lack of support for the direct send of logically contiguous elements in multidimensional arrays (e.g., two consecutive rows from a C two-dimensional array). In MPI it is possible to communicate contiguous memory regions. In MPJ this has to be done through multiple communication calls or buffering the data in a one-dimensional array in order to perform a single communication. The latter is the option initially implemented in NPB-MPJ, trying to minimize the communication overhead. However, this technique reveals an important buffering overhead.

6.2.3. NPB-MPJ Optimization

Once a fully functional NPB-MPJ implementation has been developed, several optimizations have been applied to the benchmark codes, such as array flattening and some JVM JIT compiler-based optimizations.

Array Flattening. The use of multidimensional arrays in Java presents an important overhead which can be reduced through array flattening optimization, which consists of the mapping of a multidimensional array in a one-dimensional array. This optimization has been implemented in NPB-MPJ, and thus only one-dimensional arrays are used. In order to reference a concrete element a positioning method that maps an n-dimensional location into its corresponding one-dimensional position is required. NPB-MPJ has implemented this mapping function so that adjacent elements in the C/Fortran versions are contiguous in Java, in order to provide an efficient access to the data. A particular use of the array flattening in NPB-MPJ has been applied to the complex number arrays, replacing the two-dimensional array ($complexNum\_arr[2][N]$) for a one-dimensional array ($complexNum\_arr[2 * N]$). In this case, in order to exploit the data locality, the positioning method maps a complex number to contiguous positions ($complexNum\_arr[x]$ and $complexNum\_arr[x + 1]$). Therefore, the complex numbers support is direct in MPJ communications. The array flattening has yielded significant performance increase, not only in avoiding data buffering and reducing the number of communications calls, but also in accessing the array elements.

JVM JIT Compiler-based Optimization. The JVM JIT compiler-based optimizations exploit the operation of the JVM. The Java bytecode can be either interpreted or compiled for its execution by the JVM, depending on the number of times the method to which the bytecode belongs is invoked. As the bytecode compilation is an expensive operation that significantly increases the runtime, it is reserved for heavily-used methods. However, at JVM start-up it is not always possible to find out these most frequently used methods. Therefore, the JVM gathers, at runtime, information about methods invocation and their computational cost, in order to guide the compiler optimization of the JVM JIT compiler. The JIT compiler compiles Java bytecode to native code or recompiles native code applying further optimizations in order to minimize the overall runtime of a Java application. Its operation is guided by the profiling information of the executed methods and the JVM policy.

Thus, regarding JIT compiler operation, two paradoxes generally occur in Java applications, and in particular in NPB-MPJ: (1) an optimized code yields worse performance than an unoptimized code, and (2) a code with many invocations to simple methods runs faster than a code with all the methods inlined. In the first case, the JIT compiler optimizes more aggressively the methods that fall beyond a certain load threshold. In NPB-MPJ the manual code optimization of some methods resulted in initially lower execution time than the first invocation of the unoptimized methods. Therefore, the JIT compiler does not optimize aggressively their code and eventually the overall execution time is higher than the previous version. In the second case, a Java application with multiple simple methods that are constantly invoked run faster than a Java code with less methods and whose method invocations are inlined. The simple methods are more easily optimized, in terms of compilation time and in quality of the generated code. Moreover, the JVM gathers more runtime information of methods constantly invoked, allowing a more effective optimization of the target bytecode.

NPB-MPJ takes advantage of the JIT compiler operation. Thus, in general, the code has not been manually optimized, relying on the JIT compiler for this task. However, there are few exceptions such as the use in the innermost loops of bit shifting operations instead of integer multiplication or divisions by powers of two, and the optimization of complex numbers operations in the FT kernel. Another exception is the use of the relative positioning. Instead of accessing to contiguous elements every time through global positioning method calls, the location of the first element as base position (loop invariant) is used and then contiguous elements are accessed with their corresponding offsets to the base position. This optimization is only applied in the innermost loops.

Moreover, the benchmark codes have been refactored towards simpler and independent methods. More concretely, simple methods for the multiplication and division of complex numbers, and for mapping elements from multidimensional to one-dimensional arrays have been implemented, rather than inlining these operations in the code in order to avoid the method invocation overhead. The performance improvement for NPB-MPJ of the use of simpler and independent methods has been quite significant, especially for the SP pseudo-application, for which up to 2800% performance increase has been achieved. Furthermore, the presented performance optimization techniques are easily applicable to other codes, whose performance is expected to be greatly improved.

6.3. NPB-MPJ Performance Evaluation

6.3.1. Experimental Configuration

An evaluation of Java for parallel programming using NPB-MPJ has been carried out on two InfiniBand multi-core clusters. The use of the same high-speed network, InfiniBand, as representative interconnect on both systems allows the comparison of the performance results that two quite different platforms can achieve, independently of the interconnection technology. The first testbed is the same as that used in the iodev evaluation (see Section 4.3), the x86-64 (eight dual-processor nodes Pentium IV Xeon 5060 dual-core) cluster. The only changes are the addition of InfiniBand dual 4X NICs (16 Gbps) with OFED 1.4, the OS, CentOS 5.1, the C compiler, Intel C/Fortran compiler 11.0.074 with the flag -fast, and the Intel MPI implementation (version 3.2.0.011) with InfiniBand support. Moreover, MPICH2 1.0.7 with the default communication channel (TCP/IP sockets) has also been used on Gigabit Ethernet for comparative purposes. The performance results on this system have been obtained using one core per node, except for 16 and 32 processes, for which two and four cores per node, respectively, have been used.

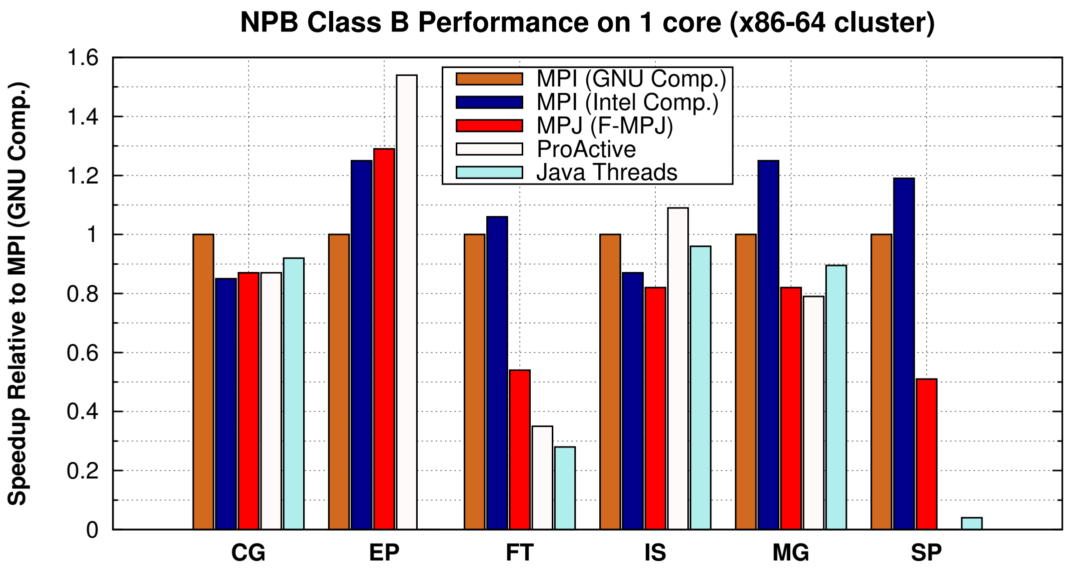
The second system is the Finis Terrae supercomputer, used in the F-MPJ evaluation (see Section 5.4.1). The only changes are the use of more recent versions of OFED (1.3) and C/Fortran compilers (Intel C/Fortran compilers 11.0.074). Regarding the JVM, Sun has released on March 2009 its JVM 1.6 update 13 for Linux IA64, but preliminary tests have shown its poorer NPB-MPJ performance than the JRockit 5.0, so the Sun JVM has not been included in the current evaluation. The performance results from this system have been obtained using up to 8 cores per node. Thus, the number of nodes used is $\lceil cores/8 \rceil$.

Moreover, for comparative purposes, a shared memory system, an HP Integrity Superdome with 128 Itanium2 Montvale cores at 1.6 GHz and 1 TB RAM, has been used. This system is integrated in the Finis Terrae and its architecture is similar to the aggregation of 8 HP Integrity rx7640 nodes (see Section 5.4.1), but using 6 crossbars for interconnecting the 16 cells (thus 8 cores per cell) instead of InfiniBand.

The evaluated MPJ libraries are an internal release of F-MPJ with JFS 0.3.1, MPJ Express 0.27 and mpiJava 1.2.5x. It has been used the NPB-MPI/NPB-OMP version 3.3 and the NPB-JAV version 3.0. The ProActive version used is the 4.0.2, which includes its own implementation of the NPB (NPB-PA). The performance results considered in this work have been derived from the sample of several iterations of the main solver method of the benchmark, ignoring the initialization times and the previous warm-up iterations. The metric that has been considered is the speedup. Moreover, Classes A and B have been used as NPB problem sizes, both on the x86-64 cluster and on the Finis Terrae, as their performance is highly influenced by the efficiency in communications, both the network interconnect and the communication library. Therefore, the differences among parallel libraries can be appreciated more easily. Additionally, Class C results have been obtained on the Finis Terrae in order to evaluate a heavier workload on a significant number of cores (256). Finally, NPB performance has been measured up to the number of available cores on the x86-64 cluster (32), on shared memory up to the number of available cores on the Superdome (128), and finally, up to 128 cores on the rx7640 nodes of the Finis Terrae, except for Class C NPB workload, that has been run on up to 256 cores.

6.3.2. Analysis of the NPB Results

The use of the speedup as measure of the performance obtained by different libraries for parallel programming presents the advantage of showing clearly the scalability of the evaluated libraries, but can hide the actual performance of the benchmarks. Figures 6.1 and 6.2 show a comparison of the performance of several implementations of the NPB on the x86-64 cluster and the Finis Terrae, respectively. The results are shown in terms of speedup relative to the MPI library (using a particular compiler), *Runtime(NPB-MPI benchmark)/Runtime(NPB benchmark)*. Thus, a value higher than 1 means than the evaluated benchmark achieves higher performance (shorter runtime) than the NPB-MPI benchmark, whereas a value lower than 1 means than the evaluated kernel shows poorer performance (longer runtime) than the NPB-MPI benchmark. The NPB implementations selected for evaluation are the MPI one (NPB-MPI), NPB-MPJ (evaluated with three different MPJ libraries: mpiJava, MPJ Express and F-MPJ), ProActive (NPB-PA), Java threads (NPB-JAV) and the OpenMP implementation (NPB-OMP). The benchmarks selected for evaluation in this section are CG, EP, FT, IS, MG and SP (see Table 6.1). The DT kernel has not been selected as it is a recent benchmark (introduced in NPB v. 3.2)



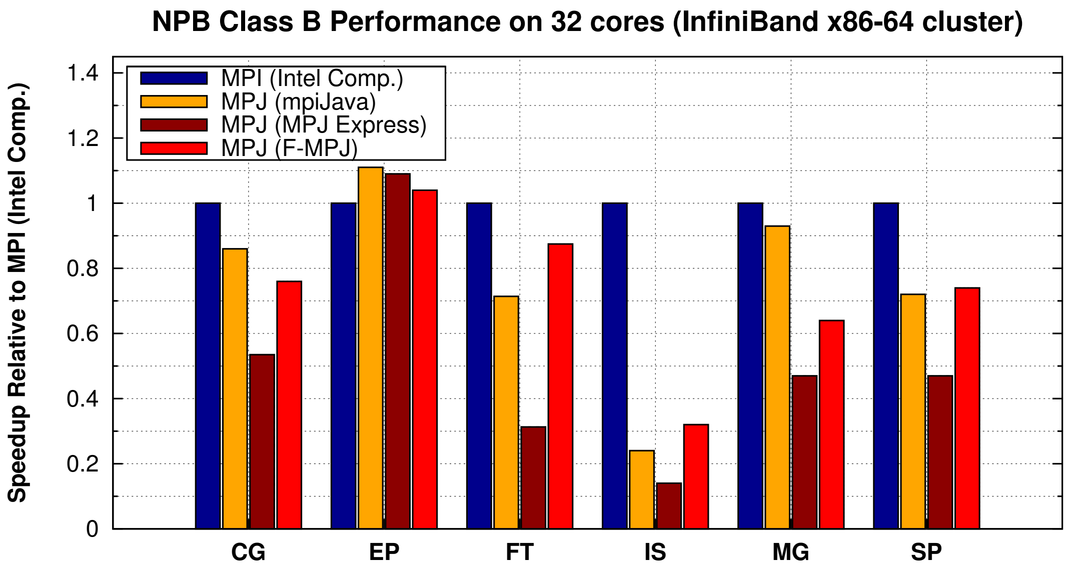
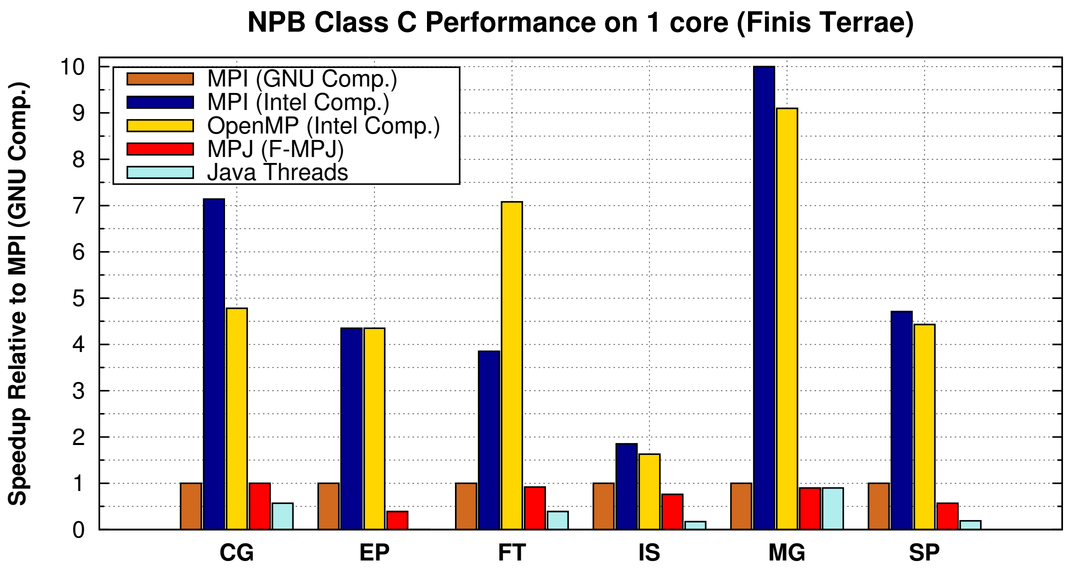


Figure 6.1: NPB relative performance on the x86-64 cluster



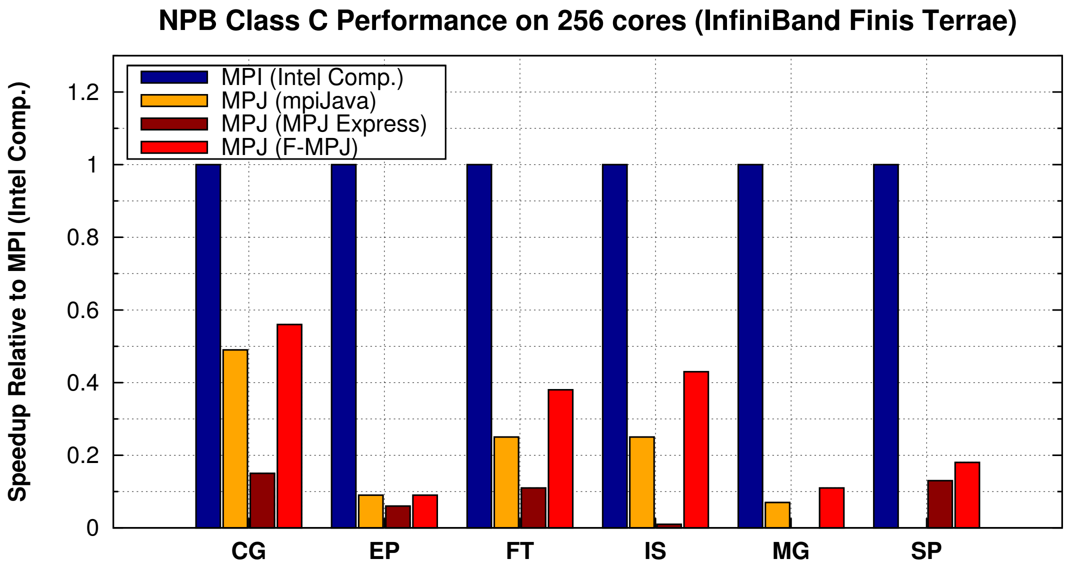


Figure 6.2: NPB relative performance on the Finis Terrae

implemented only in NPB-MPJ and NPB-MPI. Moreover, it has special resource requirements (e.g., a high number of processes) that prevent an exhaustive analysis of its scalability. The comparative analysis of their performance in terms of speedup can assist the discussion on the NPB scalability presented next in Subsections 6.3.3-6.3.5.

With respect to the figures, the graphs at the top present the NPB results using one core, whereas the bottom graphs present the NPB performance using 32 and 256 cores on the x86-64 system and the Finis Terrae, respectively. The workloads used are representative of the computational power of the testbeds: Class B for the x86-64 cluster (NPB Class C workload on one core exceeds the available memory of a single node of this cluster), and Class C workload, commonly used in supercomputers performance evaluation, on the Finis Terrae. The results on one core mainly show the performance of a particular NPB implementation and the compiler/JVM used. The performance on 32/256 cores serves to evaluate the impact of MPJ communication overhead on the overall performance. In this case, it must be taken into account that all MPJ libraries obtain quite similar results on one core (in this scenario the influence of message-passing overhead on performance is minimal and only F-MPJ results are shown for clarity purposes). Java Threads EP and ProActive SP results are missing from Figure 6.1 as these kernels are not implemented in their respective NPB suites (NPB-JAV and NPB-PA). Moreover, MPJ Express MG and mpiJava SP results are also missing from Figure 6.2 (bottom) due to runtime issues with an Allreduce call that prevent these benchmarks from being run.

The NPB-MPI results have been obtained using the Intel MPI and HP MPI libraries for the x86-64 cluster and the Finis Terrae, respectively, with two different compilers: GNU compiler (version 4.2.3 for the x86-64 cluster and 4.1.2 for the Finis Terrae), and the Intel compiler version 11.0.074. The GNU results are shown for one core in order to support the hypothesis that Java performance heavily depends on the compiler used to build the JVM. As the GNU compiler was used for building both JVMs (the publicly available Sun and JRockit JVMs for Linux) used in this evaluation, Java performance is limited by the poorer performance of the GNU compiler compared to Intel compiler results, especially on the Finis Terrae. The availability of JVMs built with the Intel compiler would significantly improve this scenario. From now on only the results of the NPB-MPI built with the Intel compiler are presented as they usually show better performance (especially on the Finis Terrae) than the NPB-MPI built with the GNU compiler.

The most remarkable conclusions that can be obtained from the analysis of Figures 6.1 and 6.2 are: (1) Java results (MPJ, ProActive and Java threads) on one core are usually slightly lower than those of GNU-built benchmarks, although it is possible that Java benchmarks outperform native code (EP on the x86-64 cluster), or, on the contrary, obtain around half of the native performance (see FT and SP results on the x86-64 cluster and EP on the Finis Terrae); and (2) MPJ libraries usually achieve higher relative performance, compared to MPI (Intel Comp.), on 32/256 cores than on one core, especially when Java relative performance on one core is low. MPJ implementations can obtain up to $[0.7 - 1.1]$ of the performance of NPB-MPI benchmarks (except for IS) built with the Intel compiler on the x86-64 cluster with 32 cores, whereas on the Finis Terrae, using 256 cores, MPJ benchmarks increase significantly their relative performance, compared to the results on one core. Thus, MPJ libraries help bridge the gap between Java and native code performance.

6.3.3. NPB-MPJ Scalability on Gigabit Ethernet

Figures 6.3 and 6.4 show NPB-MPI, NPB-MPJ and NPB-PA speedups on the x86-64 cluster using the Gigabit Ethernet network. The NPB-MPJ results have been obtained using three MPJ libraries: mpiJava, MPJ Express and F-MPJ, in order to compare them.

Regarding NPB Class A results (shown in Figure 6.3), CG shows poor speedups, especially for Java, both for MPJ and ProActive (speedups below 3). However, EP presents almost linear speedups, as it is an “Embarrassingly Parallel” kernel, with few communications that have low impact on the overall benchmark performance. In this scenario NPB-MPJ libraries achieve almost the same speedups of NPB-MPI, whereas the NPB-PA implementation has lower speedups as its communications are based on RMI, which presents high overhead. The impact on ProActive of the lack of performance of RMI can be also observed in other kernels (e.g., CG).

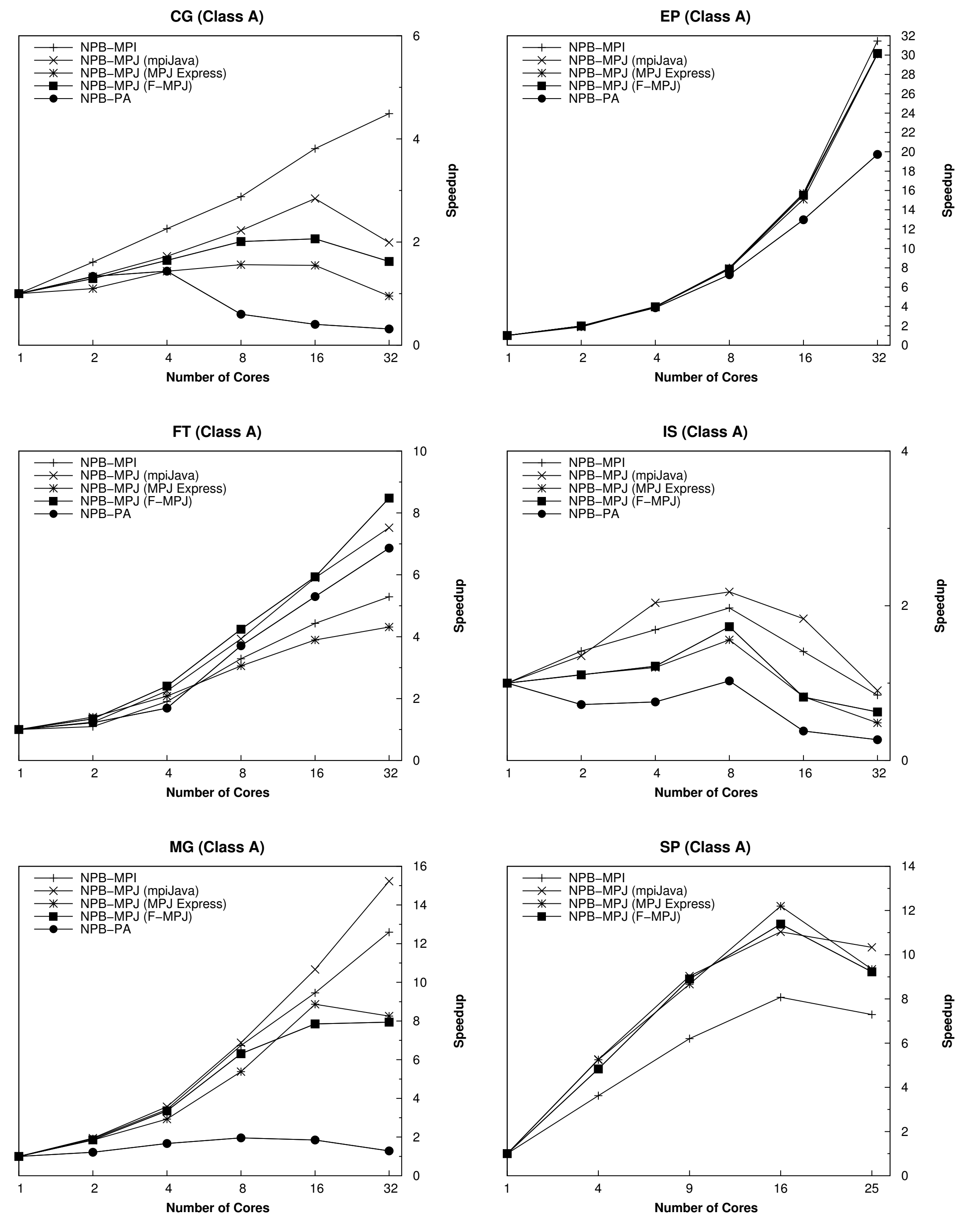


Figure 6.3: NPB Class A results on Gigabit Ethernet

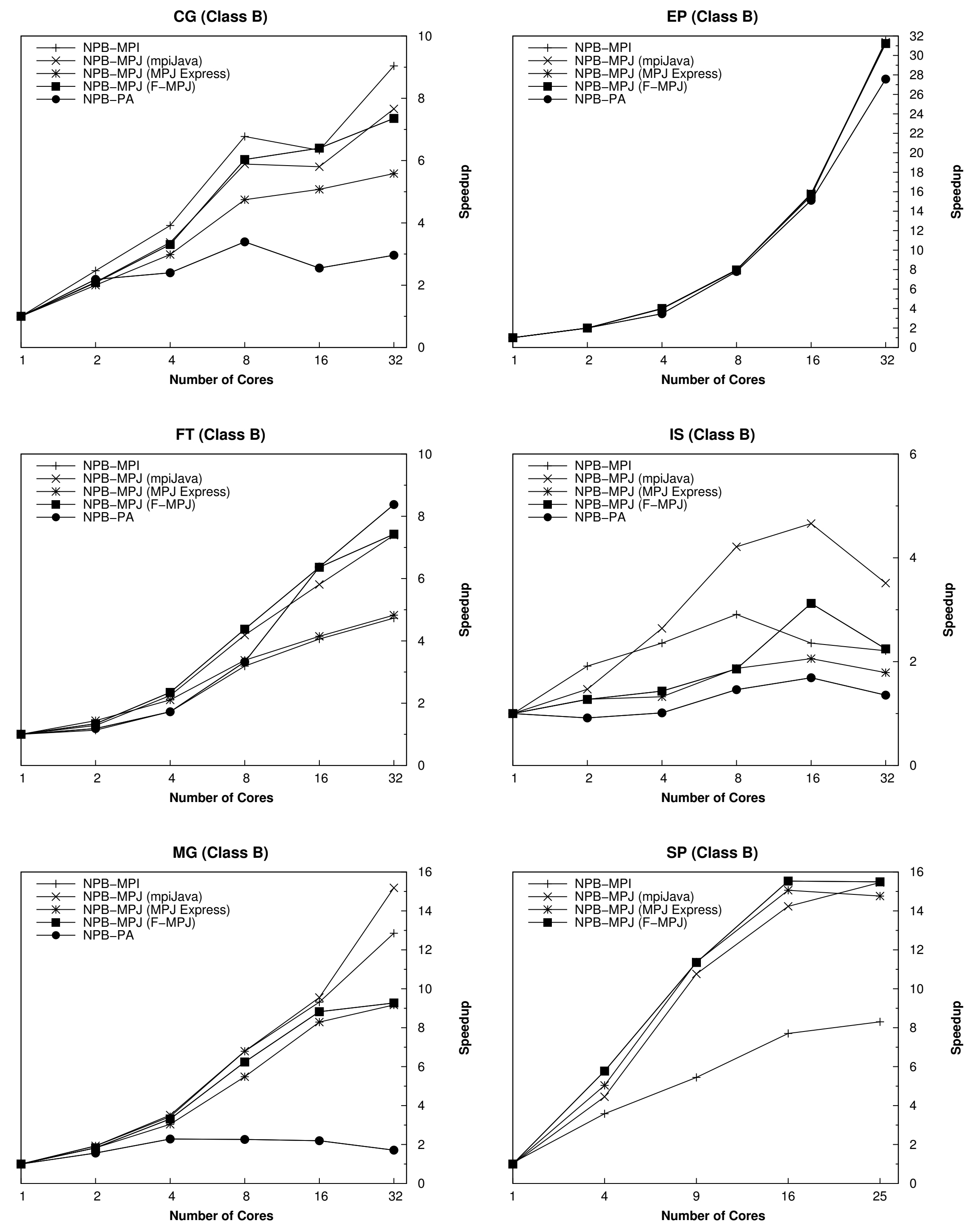


Figure 6.4: NPB Class B results on Gigabit Ethernet

Within FT kernel results, all Java libraries, except MPJ Express, overcome the scalability of MPI. The reason is the poor performance of Java on one core (see top graph in Figure 6.1), especially using ProActive, whose runtime is 2-3 times higher than that of MPI on one core. This allows Java FT implementations to benefit from the parallelization of a longer-running code, as the relative overhead of the communications is smaller than in the NPB-MPI FT. Regarding MPJ libraries, mpiJava usually shows the best NPB performance, followed by F-MPJ, and finally MPJ Express presents the lowest speedups among these libraries. However, F-MPJ achieves the highest performance for FT.

As IS is a quite communication-intensive code, its speedups are extremely low. In fact, only MPI, which shows the best performance, exceeds slightly a speedup of 2. The MG kernel shows quite poor NPB-PA speedups, and poor F-MPJ performance on 16-32 cores. For this kernel mpiJava shows the best performance, followed by MPI.

The NPB-MPJ implementation of the SP pseudo-application (there is no NPB-PA implementation for this benchmark) obtains significant speedups, higher than those of MPI, but this is explained by the poor Java performance for this code on one core (see top graph in Figure 6.1). In this case, MPJ libraries show similar performance among them. A particular feature of SP is that it requires a square number of processes (1, 4, 9, 16, 25...). On the x86-64 cluster one core per node up to 9 processes, two cores per node for 16 processes and three cores per node for 25 processes are used.

The analysis of NPB Class B results (see Figure 6.4) can be done comparatively in terms of Class A performance. Thus, CG results are significantly better, obtaining with MPI the best performance. The only relevant change in EP is that its ProActive implementation presents a higher speedup on 32 cores (a speedup of 28), compared to its Class A speedup on 32 cores (20). The increase of NPB-PA performance is also significant for FT, where ProActive achieves the highest speedup on 32 cores. The mpiJava library achieves again the highest performance for IS, whereas MG results are almost similar to those obtained with Class A. The heavier workload in SP Class B allows this benchmark to take advantage of the use of 25 cores for all libraries except MPJ Express; with Class A the best performance was obtained on 16 cores, showing lower speedups on 25 cores.

The analysis of these results can also be presented in terms of the three main evaluations that can be carried out with NPB-MPJ (mentioned in the first paragraph of Section 6.2). The first one is the comparison among MPJ implementations. The three evaluated libraries behave similarly only for EP, due to the computation-intensive nature of this kernel, whereas they present important variations for the remaining benchmarks. These performance differences between mpiJava, MPJ Express and F-MPJ are mainly explained by their communication efficiency, which is usually higher for mpiJava as it relies on a native MPI library (MPICH2 in our testbed) rather than on "pure" Java communications. CG, IS and MG results clearly confirm this point, where mpiJava outperforms both MPJ Express and F-MPJ. However, F-MPJ also obtains the best MPJ performance for FT Class A and for SP Class B. The use of the efficient communication protocols developed in this Thesis (JFS and iodev) and F-MPJ communication primitives allows to obtain significant performance benefits without the drawbacks of mpiJava, such as its reduced portability and its runtime issues due to the instability of the JVM, compromised with the access to the native MPI code through JNI. Finally, MPJ Express achieves good results on SP, thanks to the efficient non-blocking support provided by Java NIO.

The second evaluation that can be performed is the comparison of MPJ against other Java parallel libraries, in this case ProActive. ProActive is an RMI-based middleware, and for this reason its performance is usually lower than that of MPJ libraries, whose communications are based on MPI or on Java sockets. In fact, the results show that the scalability of NPB-PA is worse than that of NPB-MPJ. However, ProActive presents important features such as: development and runtime environments, profiling tools, fault tolerance and wide interoperability, being a more complete and stable middleware than the MPJ libraries evaluated.

The third analysis that has been done is the comparison of MPJ against native MPI in terms of speedup. The results presented show that MPJ scalability is higher than the MPI one, except for CG. The reason for these higher speedups is the reduced Java performance on one core, which allows MPJ or ProActive to achieve good speedups as the load of the benchmarks is heavier. Thus, MPJ and ProActive middleware help bridge the gap between Java and native codes.

6.3.4. NPB-MPJ Scalability on InfiniBand (x86-64 Cluster)

Figures 6.5 and 6.6 show NPB-MPI and NPB-MPJ speedups on the x86-64 cluster using the high-speed InfiniBand network. The NPB-PA have not been evaluated, as ProActive has not direct support on InfiniBand and the efforts carried out to use IPoIB (IP emulation over InfiniBand) as transport layer in this testbed were unsuccessful. The main motivation of this subsection is the assessment of the impact on MPI and MPJ libraries of the use of a high-speed interconnect.

Regarding NPB Class A results (see Figure 6.5), NPB-MPI shows better scalability than using Gigabit Ethernet, especially for CG and IS, whereas MPJ performance increases significantly FT and IS results. With respect to the performance graphs, it can be observed that the three MPJ libraries present different behavior. Thus, MPJ Express uses IPoIB, obtaining small performance increases compared to MPJ Express on Gigabit Ethernet. F-MPJ relies on the InfiniBand support of JFS, implemented on Sockets Direct Protocol (SDP), and thus achieves much higher speedups. Finally, mpiJava relies on the MPI support on InfiniBand, in this case implemented on IBV (InfiniBand Verbs).

The NPB Class B results on InfiniBand (see Figure 6.6) increase significantly Gigabit Ethernet speedups, obtaining almost twice their performance for CG, FT and IS. Regarding the message-passing library that obtains the highest performance per benchmark, MPI obtains the best results for CG, EP and IS, F-MPJ maximizes FT and SP performance, and finally, mpiJava gets the highest speedups for MG. The message-passing library that experiences the lowest performance increase is MPJ Express, which obtains the poorest performance as it relies on IPoIB support.

The reason for obtaining, in general, relatively low speedups (usually <18 on 32 cores, except for EP and SP Class B) is that the workloads considered (Classes A and B) are relatively small, although this allows a more thorough analysis of the scalability differences among message-passing libraries on a reduced number of cores (32). In this scenario (small workloads) the impact on performance of the high start-up latencies of Gigabit Ethernet or IPoIB is important. Thus, although a message-passing library can take advantage of shared memory transfers, a Gigabit Ethernet network or an IPoIB device represents the main performance bottleneck, especially when several processes are used per cluster node (higher network contention).

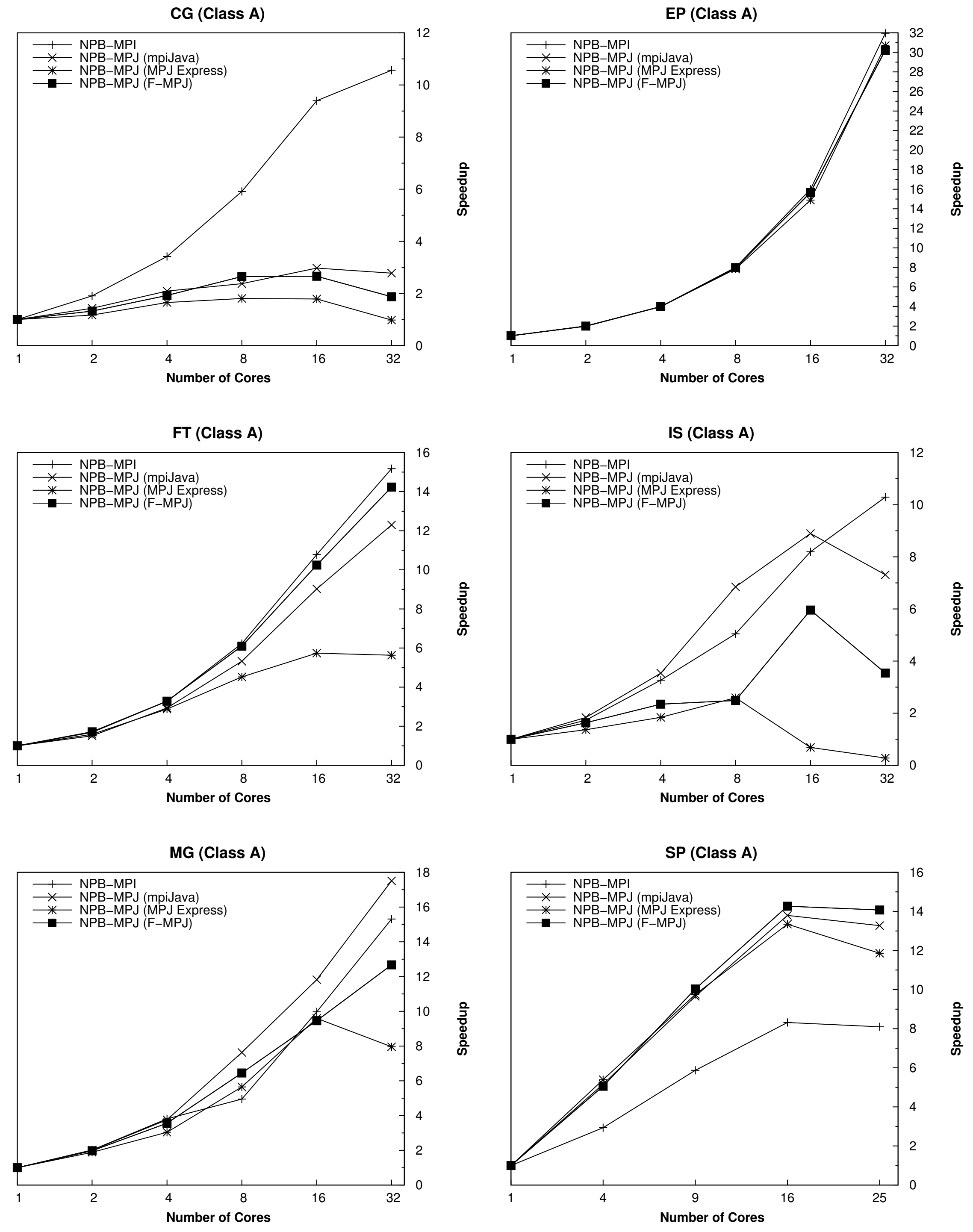


Figure 6.5: NPB Class A results on InfiniBand (x86-64 cluster)

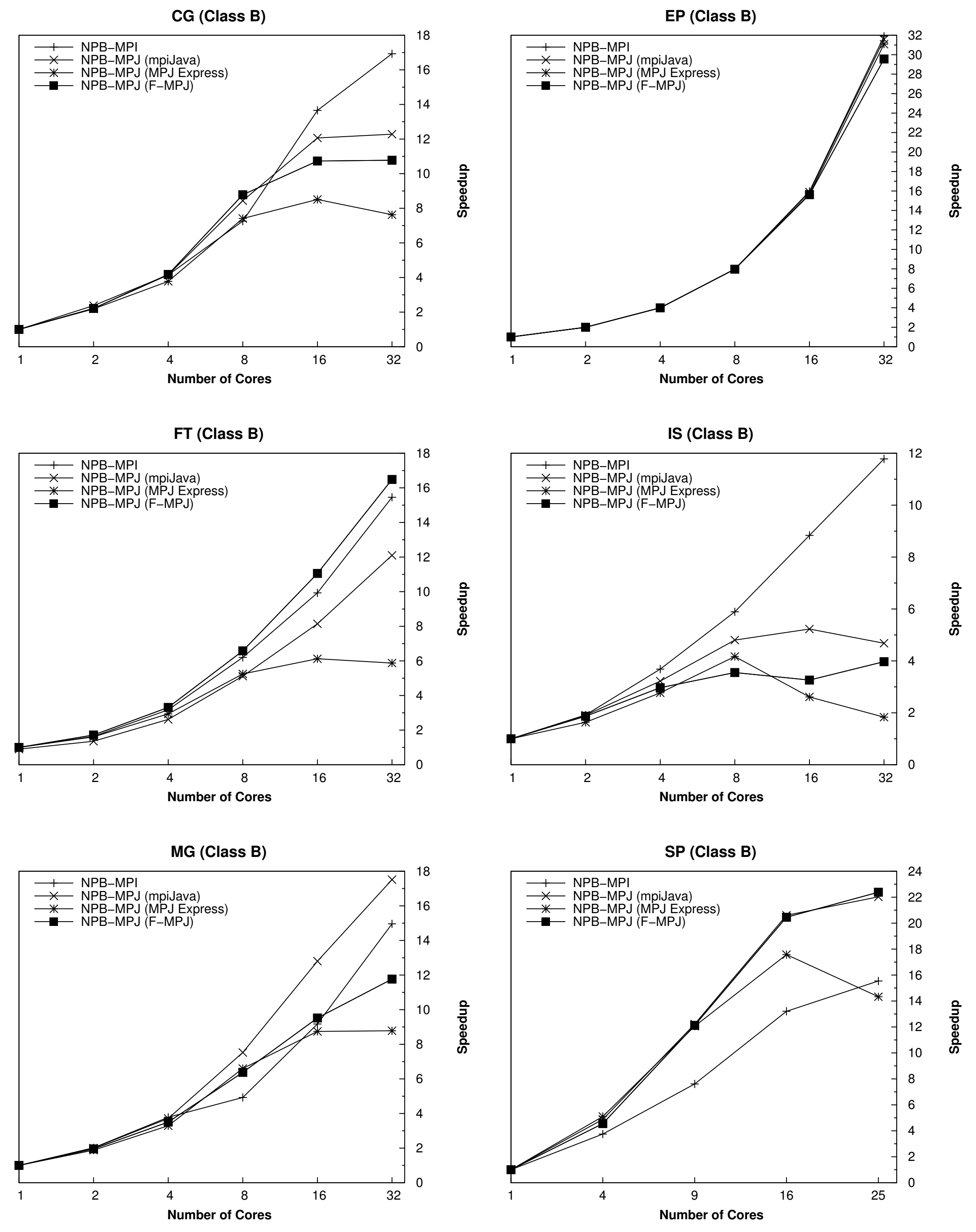


Figure 6.6: NPB Class B results on InfiniBand (x86-64 cluster)

6.3.5. NPB-MPJ Scalability on InfiniBand and Shared Memory (Finis Terrae)

Figures 6.7–6.9 show NPB-MPI, NPB-MPJ, NPB-OMP, and NPB-JAV performance on the Finis Terrae. NPB-PA results could not be obtained as it was not possible to have InfiniBand support for ProActive on this system. The distributed memory programming models (NPB-MPI and NPB-MPJ) have been evaluated on the rx7640 nodes of the Finis Terrae, using up to 8 cores per node, whereas the shared memory results (NPB-OMP and NPB-JAV) have been obtained on the HP Integrity Superdome using up to 128 cores. Although these results are obtained on two different configurations, all their characteristics, except the memory architecture (distributed on the rx7640 nodes and shared on the Superdome), are quite similar, as discussed in Subsection 6.3.1.

Regarding NPB results on the Finis Terrae (see Figures 6.7–6.9), the two MPJ libraries with high performance InfiniBand support, mpiJava and F-MPJ, achieve generally the highest speedups. Among them, mpiJava usually outperforms F-MPJ as its InfiniBand support is provided by HP MPI through its IBV driver, whereas F-MPJ is based on the JFS implementation on SDP, which shows slightly poorer performance. However, the other MPJ library, MPJ Express, usually shows the lowest performance. The high scalability of mpiJava and F-MPJ, significantly higher than that of MPI, is due to the lower performance of the MPJ benchmarks on one core (see top graph in Figure 6.2). Thus, a heavier workload can take more advantage of the message-passing paradigm. With respect to the shared memory libraries, OpenMP and Java threads, they usually show quite good performance on a reduced number of cores, usually up to 16-32-64 cores, whereas they generally obtain poorer speedups when all cores of the Superdome (128) are used. In fact, for some benchmarks (e.g., FT) the best performance is obtained with OpenMP up to 16-32-64 cores (OpenMP or Java threads obtain their highest scalability using up to 64 cores), whereas on 128 cores the message-passing libraries achieve the highest speedups. This behavior can be explained by the fact that only 8 cores per node, out of 16 cores, are used in the message-passing benchmarking as this combination maximizes the performance (the results using 8 nodes and 16 cores per node are lower). Moreover, for message-passing a cluster configuration of 16 nodes and 8 cores per node is more representative than a constellation scenario made up of 8 nodes and 16 cores per node. Thus, the InfiniBand network presents less congestion and only one core per processor is used on average. However, when using 128 cores on the Superdome the two cores of each processor are being used, and the impact on memory access throughput can be significantly higher. The analysis of the performance of shared memory solutions shows that OpenMP usually presents higher speedups than Java threads, except for SP Classes B and C, where NPB-JAV results overcome NPB-OMP. It is important to note that NPB-JAV does not include an implementation of the EP kernel.

The NPB Class A performance has been measured using up to 128 cores (see Figure 6.7). The CG results, quite dependent on the problem workload, show relatively small speedups (below 25), except for mpiJava. The shared memory libraries present good scalability, but only up to 8 and 16 cores for Java threads and OpenMP, respectively. The EP kernel, due to its small number of communications, shows a high parallel efficiency on this system, especially for mpiJava and MPI.

Regarding FT performance, on the one hand, the shared memory implementations (NPB-JAV and NPB-OMP), obtain the best speedups up to 32 and 64 cores, respectively. With respect to the performance on 128 cores, F-MPJ and mpiJava get the best results, increasing slightly OpenMP speedup. MPJ Express, on the other hand, presents the lowest speedups.

The next kernel, IS, is a communication-intensive kernel whose implementations obtain poor speedups, especially Java threads and MPJ Express due to its lack of efficient InfiniBand support. For IS mpiJava achieves the highest speedups on 64 and 128 cores. Regarding MG, mpiJava obtains again the highest scalability. For this benchmark the shared memory libraries present low speedups, especially the NPB-JAV implementation, whose results are as low as those of MPJ Express. Moreover, some of the results of this MPJ library are missing due to a runtime issue with the Allreduce collective, which was also observed for MG Classes B and C.

With respect to the remaining benchmark, SP, mpiJava results could not be obtained due to a runtime Java-to-MPI wrapping error in an Allreduce call. The SP pseudo-application requires the use of a square number of cores so its performance has been evaluated on up to 121 cores for Classes A and B workloads. For this benchmark the message-passing libraries achieve good speedups, especially the na-

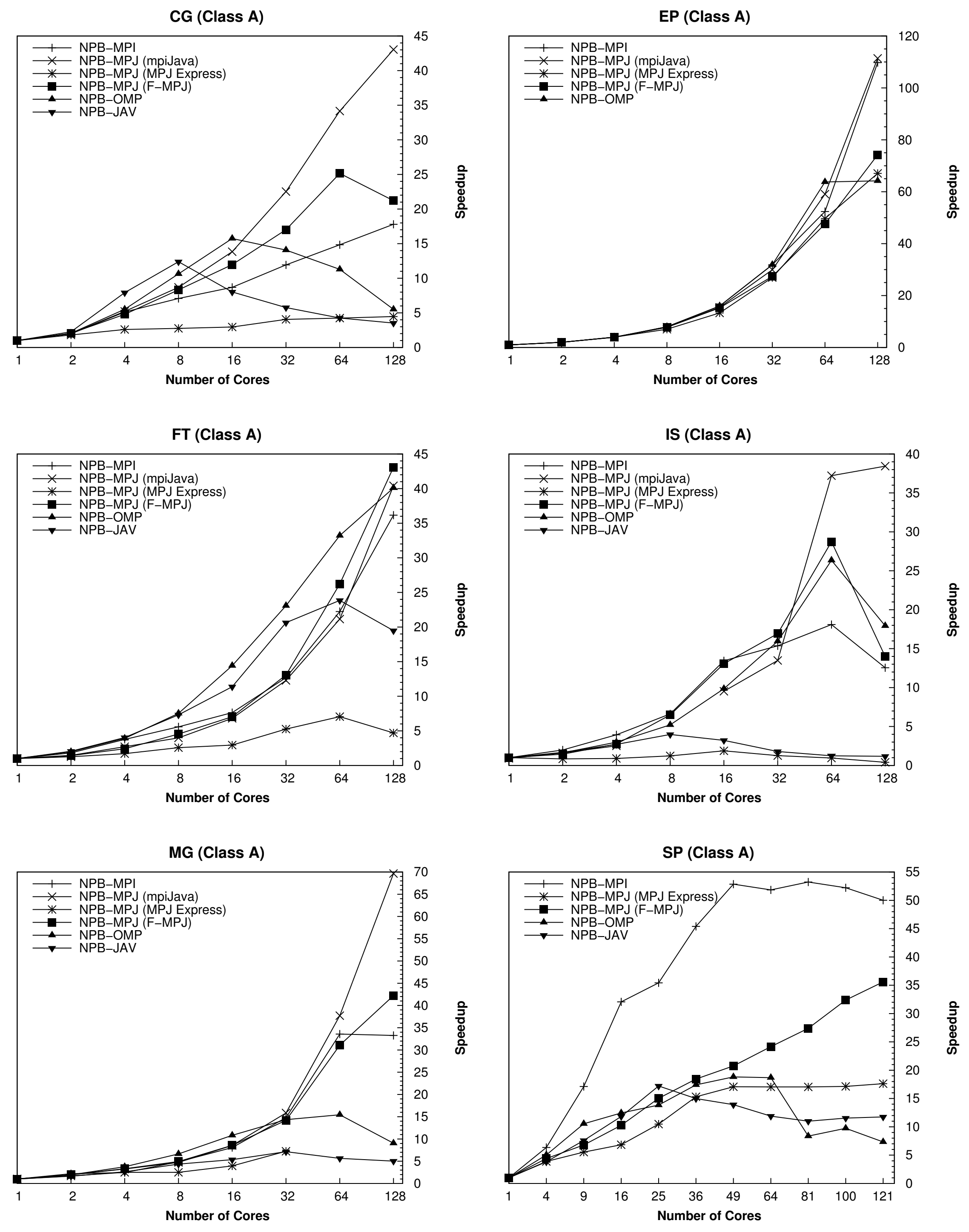


Figure 6.7: NPB Class A results on the Finis Terrae

tive MPI. However, the super-linear speedups obtained only by MPI (e.g., a speedup of 33 on 16 cores) suggest that the MPI SP performance on one core is sub-optimal. Among the MPJ libraries F-MPJ achieves the best results. Regarding the shared memory implementations, both NPB-OMP and NPB-JAV present low scalability for Class A when using more than 36 and 64 cores, respectively, as their implementations have a finer parallel granularity (at loop level) than the message-passing ones (at subtask level). Thus, the use of an important number of threads with small workloads can cause significant performance degradation.

The NPB Class B results on the Finis Terrae have been obtained using up to 128 cores (see Figure 6.8). The speedups obtained are usually higher than the ones of Class A. Nevertheless, most of the Class B results for all implementations of MG and NPB-JAV FT and IS are below the Class A ones due to the good performance obtained by these benchmarks on one core.

Regarding the libraries evaluated, mpiJava achieves the best scalability on CG, EP and MG. However, mpiJava shows runtime problems, apart from the issues experienced with SP, for IS executions on 16 and 32 cores, so their results are missing. These problems are not present with the use of F-MPJ, which achieves good scalability for all benchmarks, especially for FT, IS and SP. With respect to shared memory libraries, the use of a heavier workload allows OpenMP and Java threads to take advantage of a higher number of threads. Thus, OpenMP obtained its highest speedups on 64 cores for CG, and on 128 cores for IS, whereas its best speedups for Class A were obtained with 16 and 64 cores for CG and IS, respectively. NPB-JAV also takes advantage of heavier workloads, although their scalability is usually significantly lower than that of NPB-OMP, except for SP.

The NPB Class C performance has been measured using up to 256 cores (see Figure 6.9). Regarding mpiJava runtime issues, the heavier the workload, the higher the number of results missing. Thus, there are missing Class C results for SP, IS on 16–128 cores, and for MG on 16 and 64 cores. The instability shown by the mpiJava installation in this system allows F-MPJ to obtain the highest speedups. Thus, F-MPJ achieves the highest scalability for Class C workload on 256 cores for CG, FT, MG and SP. With respect to shared memory implementations, NPB-OMP and NPB-JAV, they show the lowest performance on MG and SP due to the finer granularity of their codes compared to the message-passing implementations. An

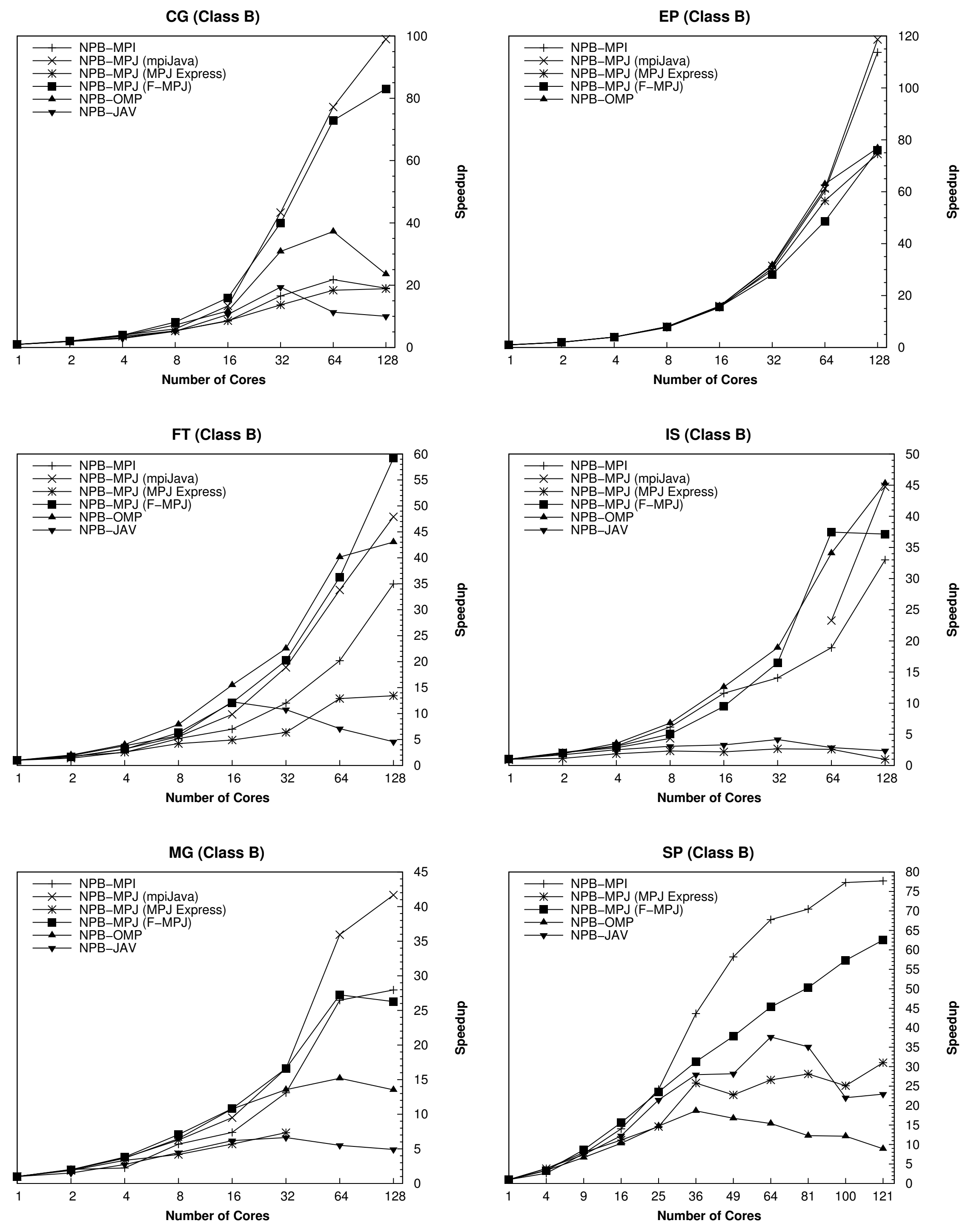


Figure 6.8: NPB Class B results on the Finis Terrae

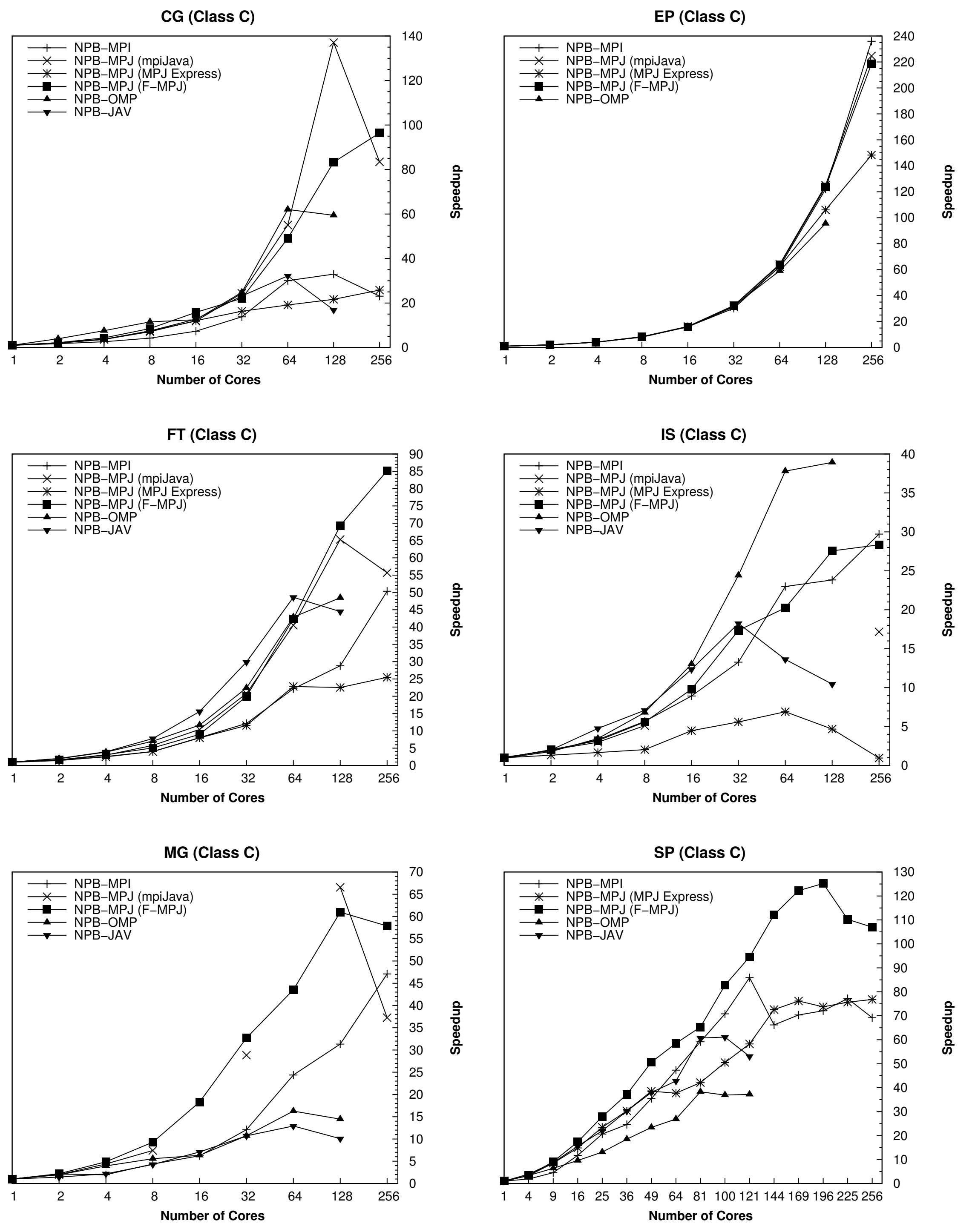


Figure 6.9: NPB Class C results on the Finis Terrae

analysis of the MPI scalability for Class C shows that it is similar to the results achieved for Class B. However, MPJ Class C speedups are significantly higher than those of Class B for EP, FT, MG and SP, especially for F-MPJ, but also for MPJ Express. In fact, MPJ Express only obtains the lowest Class C speedups for FT and IS (MG is not considered as its results could not be obtained).

The SP results, presented for all square numbers up to 256, show the positive impact of Java parallel libraries, especially F-MPJ, in bridging the performance gap between Java and native languages. Thus, NPB-JAV clearly outperforms NPB-OMP scalability, whereas MPJ Express, running on the emulation layer over InfiniBand (IPoIB) achieves similar speedups to MPI; and F-MPJ, using the InfiniBand support of JFS, outperforms MPI speedups by at least 50% when using more than 121 cores.

6.4. Chapter 6 Conclusions

This chapter has analyzed the process of developing efficient parallel Java applications through the design, implementation and performance optimization of NPB-MPJ, which is the first extensive implementation of the standard benchmark suite NPB for Message-Passing in Java (MPJ). These parallel benchmarks have been selected as representative codes for this task as they are widely known and extended in HPC evaluations. NPB-MPJ, allows, as main contributions:

- The evaluation of a significant number of existing MPJ libraries.
- The analysis of MPJ performance against other Java parallel approaches.
- The assessment of MPJ versus native MPI scalability.
- The study of the impact on performance of the optimization techniques used in NPB-MPJ, from which Java HPC applications can potentially benefit.

The evaluation of F-MPJ using the NPB-MPJ on two multi-core systems using InfiniBand and Gigabit Ethernet as interconnects has shown that F-MPJ:

- Can achieve similar, or even higher, speedups than mpiJava without suffering from the runtime and portability issues of an MPJ wrapper library.

- Is generally more scalable than Java threads and RMI-based middleware.
- Can outperform MPI scalability, especially on InfiniBand, with heavy workloads (NPB Class C), and using an important number of cores (256), thus bridging the gap between message-passing Java and native code.

The analysis of the results of this evaluation has shown that MPJ libraries, especially F-MPJ, are an alternative to native languages (C/Fortran) for parallel programming on multi-core systems, as it is possible to take advantage of the features of Java while achieving higher speedups than MPI libraries. Finally, NPB-MPJ can help MPJ library developers in order to detect performance penalties in their implementations and bridge the gap with native solutions.

Conclusions

This PhD Thesis, *"Design of Efficient Java Communications for High Performance Computing"*, has argued that it is possible to develop scalable Java applications for HPC as long as efficient communication middleware is made available. The analysis of the state of the art has revealed that Java lacks efficient communication support, which has prevented its adoption in HPC. Further research with new tools and models developed in this Thesis has identified the main performance penalties in Java communications:

- Poor high-speed networks support.
- The data copies between the Java heap and native code through JNI.
- Costly data serialization.
- Lack of efficient non-blocking communications support on Java IO sockets.
- The use of communication protocols unsuitable for HPC.

Once the main causes of inefficiency in Java communications have been determined, the design of their solutions has been accomplished. The first step was the development of a high performance Java sockets implementation, named Java Fast Sockets (JFS), a communication middleware that provides efficient communication in Java. Among its main contributions, JFS:

- Enables efficient communication on high performance clusters interconnected via high-speed networks (SCI, Myrinet and InfiniBand) through a general and portable solution.

- Avoids the need of primitive data type array serialization.
- Reduces buffering and unnecessary copies.
- Optimizes shared memory (intra-node) communication.
- Does not need source code modification, being user and application transparent.

In order to overcome the blocking nature of Java IO sockets communication it has been implemented its non-blocking support in the iodev low-level communication device, which has been used as base for the development of a Java message-passing library. Among its main characteristics, iodev:

- Provides efficient non-blocking point-to-point communication primitives on Java IO sockets.
- Takes advantage of high-speed networks through the use of JFS.
- Avoids the use of buffers for the message data to be transferred.
- Reduces the serialization overhead, especially for arrays of primitive data types.
- Implements a communication protocol that minimizes the start-up latency and maximizes the bandwidth.

Then, an efficient Java RMI protocol for its use on high-speed clusters has been implemented. The solution proposed is transparent to the user (it does not need source code modification) and improves performance significantly. The RMI optimizations have been focused on:

- Increasing the transport protocol performance through the use of the high performance Java sockets library JFS and reducing the information to be transferred.
- Providing a new method which deals with array communication, avoiding type checks and taking advantage of the reduction of the serialization overhead using JFS.

- Reducing the versioning and data block information as well as class annotations.

Furthermore, a Java message-passing library, named Fast MPJ (F-MPJ), has been implemented. F-MPJ integrates the collection of middleware/library developments of this Thesis, thus outperforming the scalability of previous Java message-passing implementations. Among its main contributions, this library:

- Takes advantage of the efficient integration of iodev (see Section 4.2) and JFS (see Section 3.1) into the implementation of the MPJ primitives, obtaining efficient non-blocking communication and high-speed multi-core clusters support.
- Increases MPJ communications performance (obtains lower start-up latencies and higher aggregated bandwidths) and scalability through an extensive use of communications overlapping.
- Implements several communication algorithms per message-passing collective, allowing its selection at runtime.

Finally, the process of developing efficient parallel Java applications has been analyzed. This has been done through the implementation of the NAS Parallel Benchmarks for message-passing in Java, NPB-MPJ. The evaluation of F-MPJ using the NPB-MPJ (see Section 6.3) has shown that F-MPJ:

- Can achieve similar, or even higher, speedups than mpiJava without suffering from the runtime and portability issues of an MPJ wrapper library.
- Is generally more scalable than Java threads and RMI-based middleware.
- Can outperform MPI scalability, especially on InfiniBand, with heavy workloads (NPB Class C), and using an important number of cores (256), thus bridging the gap between message-passing Java and native code.

The analysis of the impact of the solutions developed in the overall performance of Java parallel applications has confirmed the initial hypothesis, that it is possible to develop scalable Java applications for HPC. However, the development of Java parallel applications also has to take into account the optimization of Java code, which has shown significant impact on performance, as was seen in the micro-benchmarks and kernel/application benchmarks used in the performance evaluations conducted in this Thesis.

Although it has been shown that the use of Java in HPC is feasible, it is necessary to continue improving and expanding the projects started in this work. Thus, the development of new high performance Java sockets implementations such as SCTP sockets (Stream Control Transmission Protocol) [85] would be of great interest. In fact, this protocol is message-oriented, which makes it especially suitable for being the base for Java message-passing implementations. Regarding this point, the completion of the implementation of the API mpiJava 1.2 in F-MPJ (see Section 5.1) would be interesting. It is pending the implementation of group, communicator and process topology management, as well as additional development and runtime tools (debugging and bootstrapping). Additionally, the integration of the optimized RMI protocol and JFS in other implementations of Java communication middleware, such as ProActive, has to be considered. Thus, it would be possible to significantly broaden the scope of the performance improvements obtained with the communication middleware and libraries developed in this Thesis.

Moreover, it should be noted that the rise of multi-core systems demands a further development of shared memory solutions. Thus, the development of an efficient shared memory message-passing device and an implementation of OpenMP in Java are highly interesting. This would allow the combination of the optimizations developed for distributed memory communication libraries (JFS, Opt RMI, iodev, and F-MPJ) with shared memory protocols in order to take advantage of multi-core clusters. Thus, intra-node communications would be shared memory operations, whereas inter-node transfers would be performed by efficient communication middleware.

This Thesis has led to several publications in the area of design and development of Java communication libraries for high-speed clusters. Thus, the state of the art in Java message-passing libraries and an evaluation of their performance on a Fast Ethernet cluster was presented in [95]. Then, a comparative performance analysis of message-passing primitives (both Java and native) on three different interconnection networks (SCI, Myrinet, Fast Ethernet) was shown in [88]. These works have been updated with the proposal of a more accurate performance model and its application for a comprehensive analytical modeling of Java and native message-passing libraries on high-speed clusters in [94]. This latter work includes an up-to-date evaluation of the available Java message-passing projects, a performance optimization process based on analytical communication performance models, and an analysis of the impact of message-passing overhead on systems with multiple processors (multi-core or with hyper-threading).

The design of efficient communications focused on solving the drawbacks of the available Java communication projects was first discussed in [89]. This paper served to introduce the work methodology of this Thesis, a bottom-up approach which started with the optimization of the low-level Java sockets API [90, 91, 92]. Thus, the design and development of JFS improved the JVM sockets performance through an efficient high-speed networks support, and extended the sockets API to avoid serialization. The lack of efficient non-blocking communications support on Java IO sockets was the next target to be tackled. In this case the preliminary design and development of iodev was presented in [96]. The optimization of the RMI protocol was covered in [86]. Furthermore, the design and implementation of F-MPJ, the Java message-passing library that takes advantage of the previous developments, was presented in [93]. Finally, the development of several benchmark codes (NAS Parallel Benchmarks and micro-benchmark suites for MPJ) has allowed the evaluation of the current performance of Java for HPC, especially the communication overhead of F-MPJ comparatively with other MPJ libraries (mpiJava and MPJ Express), as discussed in [61].

Further information, additional documentation and downloads of the projects presented in this Thesis are available from the webpage `http://jfs.des.udc.es`.

Bibliography

[1] A. Alexandrov, M. F. Ionescu, K. E. Schuser and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model – one Step Closer Towards a Realistic Model for Parallel Computation. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1997. pages 22, 24

[2] A. A. G. Alves, A. Pina, J. L. P. Exposto, and J. Rufino. Scalable Multithreading in a Low Latency Myrinet Cluster. In *Proc. 5th Intl. Conf. on High Performance Computing in Computational Sciences (VECPAR'02), Lecture Notes in Computer Science vol. 2565*, pages 579–592, Porto, Portugal, 2002. pages 48

[3] B. Amedro, V. Bodnartchouk, D. Caromel, C. Delbé, F. Huet, and G. L. Taboada. Current State of Java for HPC. In *INRIA Technical Report RT-0353*, pages 1–24, INRIA Sophia Antipolis, Nice, France, 2008, http://hal.inria.fr/inria-00312039/en/ [Last visited: May 2009]. pages 1, 135

[4] B. Amedro, D. Caromel, F. Huet, and V. Bodnartchouk. Java ProActive vs. Fortran MPI: Looking at the Future of Parallel Java. In *Proc. 10th Intl. Workshop on Java and Components for Parallelism, Distribution and Concurrency (IWJacPDC'08), Miami, FL, USA*, page 134b (8 pages), 2008. pages 109, 135

[5] L. Baduel, F. Baude, and D. Caromel. Object-oriented SPMD. In *Proc. 5th IEEE Intl. Symposium on Cluster Computing and the Grid (CCGrid'05)*, pages 824–831, Cardiff, UK, 2005. pages 12, 109, 133

[6] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. Schreiber, H. D.

Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS Parallel Benchmarks - Summary and Preliminary Results. In *Proc. 4th ACM/IEEE Conf. on Supercomputing (SC'91)*, pages 158 – 165, Albuquerque, NM, USA, 1991. pages 134

[7] M. Baker, B. Carpenter, G. Fox, S. Ko, and S. Lim. mpiJava: an Object-Oriented Java Interface to MPI. In *Proc. 1st Intl. Workshop on Java for Parallel and Distributed Computing (IWJPDC'99), Lecture Notes in Computer Science vol. 1586*, pages 748–762, San Juan, Puerto Rico, 1999. pages 14, 29

[8] M. Baker, B. Carpenter, and A. Shafi. A Pluggable Architecture for High-Performance Java Messaging. *IEEE Distributed Systems Online*, 6(10):1–4, 2005. pages 78

[9] M. Baker, B. Carpenter, and A. Shafi. MPJ Express: Towards Thread Safe Java HPC. In *Proc. 8th IEEE Intl. Conf. on Cluster Computing (CLUSTER'06)*, pages 1–10, Barcelona, Spain, 2006. pages 15

[10] P. Balaji, P. Shivan, P. Wyckoff, and D. K. Panda. High Performance User Level Sockets over Gigabit Ethernet. In *Proc. 4th IEEE Intl. Conf. on Cluster Computing (CLUSTER'02)*, pages 179–186, Chicago, IL, USA, 2002. pages 11

[11] S. Bang and J. Ahn. Implementation and Performance Evaluation of Socket and RMI based Java Message Passing Systems. In *Proc. 5th ACIS Intl. Conf. on Software Engineering Research, Management and Applications (SERA'07)*, pages 153 – 159, Busan, Korea, 2007. pages 17

[12] A. Barak, I. Gilderman, and I. Metrik. Performance of the Communication Layers of TCP/IP with the Myrinet Gigabit LAN. *Computer Communications*, 22(11):989–997, 1999. pages 10

[13] L. A. Barchet-Estefanel and G. Mounie. Fast Tuning of Intra-cluster Collective Communications. In *Proc. 11th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'04), Lecture Notes in Computer Science vol. 3241*, pages 28 – 35, Budapest, Hungary, 2004. pages 41, 118

[14] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, 1995. pages 1

[15] R. G. Börger, R. Butenuth, and H.-U. Hei. IP over SCI. In *Proc. 2nd IEEE Intl. Conf. on Cluster Computing (CLUSTER'00)*, pages 73–77, Chemnitz, Germany, 2000. pages 10

[16] M. Bornemann, R. V. v. Nieuwpoort, and T. Kielmann. MPJ/Ibis: a Flexible and Efficient Message Passing Platform for Java. In *Proc. 12th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'05), Lecture Notes in Computer Science vol. 3666*, pages 217–224, Sorrento, Italy, 2005. pages 15, 29, 71

[17] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A Benchmark Suite for High Performance Java. *Concurrency: Practice and Experience*, 12(6):375–388, 2000. pages 71, 123, 136

[18] S. G. Caglar, G. D. Benson, Q. Huang, and C.-W. Chu. USFMPI: A Multi-threaded Implementation of MPI for Linux Clusters. In *Proc. 15th IASTED Intl. Conf. on Parallel and Distributed Computing and Systems (PDCS'03)*, pages 674 – 680, Marina del Rey, CA, USA, 2003. pages 48

[19] K. W. Cameron and R. Ge. Predicting and Evaluating Distributed Communication Performance. In *Proc. 17th ACM/IEEE Conf. on Supercomputing (SC'04)*, page 43 (15 pages), Pittsburgh, PA, USA, 2004. pages 22, 23

[20] K. W. Cameron and X.-H. Sun. Quantifying Locality Effect in Data Access Delay: Memory logP. In *Proc. 17th Intl. Parallel and Distributed Processing Symposium (IPDPS'03)*, page 48 (8 pages), Nice, France, 2003. pages 22

[21] B. Carpenter, G. Fox, S.-H. Ko, and S. Lim. mpiJava 1.2: API Specification. http://www.hpjava.org/reports/mpiJava-spec/mpiJava-spec/mpiJava-spec.html [Last visited: May 2009]. pages 14, 116

[22] B. Carpenter, V. Getov, G. Judd, A. Skjellum, and G. Fox. MPJ: MPI-like Message Passing for Java. *Concurrency: Practice and Experience*, 12(11):1019–1038, 2000. pages 14, 116

[23] E. Chan, M. Heimlich, A. Purkayastha, and R. A. van de Geijn. Collective Communication: Theory, Practice, and Experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, 2007. pages 118

[24] C.-C. Chang and T. von Eicken. Javia: a Java Interface to the Virtual Interface Architecture. *Concurrency: Practice and Experience*, 12(7):573–593, 2000. pages 20

[25] G. Crawford, Y. Dandass, and A. Skjellum. The JMPI Commercial Message Passing Environment and Specification: Requirements, Design, Motivations, Strategies, and Target Users. In *MPI Software Technology Inc. Technical Report*, Starkville, MS, USA, 1998. pages 16

[26] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proc. 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'93)*, pages 1 – 12, San Diego, CA, USA, 1993. pages 22

[27] K. Datta, D. Bonachea, and K. A. Yelick. Titanium Performance and Potential: An NPB Experimental Study. In *Proc. 18th Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC'05), Lecture Notes in Computer Science vol. 4339*, pages 200–214, Hawthorne, NY, USA, 2005. pages 135

[28] K. Dincer. Ubiquitous Message Passing Interface Implementation in Java: jmpi. In *Proc. 13th Intl. Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing (IPPS/SPDP'99)*, pages 203–207, San Juan, Puerto Rico, 1999. pages 16

[29] Dolphin Interconnect Solutions Inc. IP over SCI. http://www.dolphinics.com/products/software.html [Last visited: May 2009]. pages 10

[30] J. Dongarra, D. Gannon, G. Fox, and K. Kennedy. The Impact of Multicore on Computational Science Software. *CTWatch Quarterly*, 3(1):1–10, 2007. pages 1

[31] T. v. Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *19th Intl. Symposium on Computer Architecture (ISCA'92)*, pages 256–266, Gold Coast, Australia, 1992. pages 11

[32] M. Factor, A. Schuster, and K. Shagin. JavaSplit: a Runtime for Execution of Monolithic Java Programs on Heterogenous Collections of Commodity Workstations. In *Proc. 5th IEEE Intl. Conf. on Cluster Computing (CLUSTER'03)*, pages 110–117, Hong Kong, China, 2003. pages 20

[33] A. Ferrari. JPVM: Network Parallel Computing in Java. *Concurrency: Practice and Experience*, 10(11-13):985–992, 1998. pages 17

[34] Finis Terrae Supercomputer. . http://www.top500.org/system/9156 [Last visited: May 2009]. pages 91, 123

[35] M. I. Frank, A. Agarwal, and M. K. Vernon. LoPC: Modeling Contention in Parallel Algorithms. In *Proc. 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'97)*, pages 276 – 287, Las Vegas, NV, USA, 1997. pages 22

[36] S. Genaud and C. Rattanapoka. P2P-MPI: A Peer-to-Peer Framework for Robust Execution of Message Passing Parallel Programs. *Journal of Grid Computing*, 5(1):27–42, 2007. pages 17, 77, 136

[37] V. Getov, Q. Lu, M. Thomas, and M. Williams. Message-passing Computing with Java: Performance Evaluation and Comparisons. In *Proc. 9th Euromicro Workshop on Parallel and Distributed Processing (PDP'01)*, pages 173–177, Mantova, Italy, 2001. pages 135

[38] K. Ghouas, K. Omang, and H. O. Bugge. VIA over SCI: Consequences of a Zero Copy Implementation and Comparison with VIA over Myrinet. In *Proc. 1st Intl. Workshop on Communication Architecture for Clusters (CAC'01)*, pages 1632–1639, San Francisco, CA, USA, 2001. pages 20

[39] A. S. Gokhale and D. C. Schmidt. Measuring and Optimizing CORBA Latency and Scalability Over High-Speed Networks. *IEEE Transactions on Computers*, 47(4):391–413, 1998. pages 20

[40] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996. pages 78

[41] H. Hellwagner and A. Reinefeld, editors. *SCI - Scalable Coherent Interface: Architecture and Software for High-Performance Compute Clusters*. Lecture Notes in Computer Science vol. 1734. Springer-Verlag, 1999. pages 1

[42] W. Huang, H. Zhang, J. He, J. Han, and L. Zhang. Jdib: Java Applications Interface to Unshackle the Communication Capabilities of InfiniBand Networks. In *Proc. 4th IFIP Intl. Conf. Network and Parallel Computing (NPC'07)*, pages 596–601, Dalian, China, 2007. pages 20

[43] IBM. Asynchronous IO for Java. http://www.alphaworks.ibm.com/tech/aio4j [Last visited: May 2009]. pages 81

[44] IETF Draft. IP over IB. http://www.ietf.org/ids.by.wg/ipoib.html. [Last visited: May 2009]. pages 10

[45] InfiniBand Trade Association. http://www.infinibandta.org. [Last visited: May 2009]. pages 1

[46] F. Ino, N. Fujimoto, and K. Hagihara. LogGPS: A Parallel Computational Model for Synchronization Analysis. In *Proc. 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'01)*, pages 133 – 142, Snowbird, UT, USA, 2001. pages 22

[47] Intel Corporation. Offload Sockets Framework and Sockets Direct Protocol High Level Design, Draft 2. http://infiniband.sourceforge.net/archive/-OSF_SDP_HLD.pdf [Last visited: May 2009]. pages 11

[48] Java Grande Forum. http://www.javagrande.org. [Last visited: May 2009]. pages 14

[49] G. Judd, M. Clement, and Q. Snell. DOGMA: Distributed Object Group Metacomputing Architecture. *Concurrency: Practice and Experience*, 10(11-13):977–983, 1998. pages 16

[50] M. E. Kambites, J. Obdrzálek, and J. M. Bull. An OpenMP-like Interface for Parallel Programming in Java. *Concurrency and Computation: Practice and Experience*, 13(8-9):793–814, 2001. pages 9

[51] A. Kaminsky. Parallel Java: A Unified API for Shared Memory and Cluster Parallel Programming in 100% Java. In *Proc. 9th Intl. Workshop on Java and Components for Parallelism, Distribution and Concurrency (IWJacPDC'07)*, page 196a (8 pages), Long Beach, CA, USA, 2007. pages 17

[52] P. J. Keleher. Update Protocols and Cluster-based Shared Memory. *Computer Communications*, 22(11):1045–1055, 1999. pages 20

[53] T. Kielmann, H. Bal, S. Gorlatch, K. Verstoep, and R. Hofman. Network Performance-aware Collective Communication for Clustered Wide Area Systems. *Parallel Computing*, 27(11):1431–1456, 2001. pages 22

[54] J.-S. Kim, K. Kim, and S.-I. Jung. SOVIA: A User-level Sockets Layer Over Virtual Interface Architecture. In *Proc. 3rd IEEE Intl. Conf. on Cluster Computing (CLUSTER'01)*, pages 399–408, New Port Beach, CA, USA, 2001. pages 10, 11

[55] M. Klemm, M. Bezold, R. Veldema, and M. Philippsen. JaMP: an Implementation of OpenMP for a Java DSM. *Concurrency and Computation: Practice and Experience*, 19(18):2333–2352, 2007. pages 10

[56] D. Kurzyniec, T. Wrzosek, V. Sunderam, and A. Slominski. RMIX: A Multiprotocol RMI Framework for Java. In *Proc. 5th Intl. Workshop on Java for Parallel and Distributed Computing (IWJPDC'03)*, page 140 (7 pages), Nice, France, 2003. pages 12

[57] M. Lobosco, C. L. de Amorim, and O. Loques. Java for High-Performance Network-Based Computing: a Survey. *Concurrency and Computation: Practice and Experience*, 14(1):1–31, 2002. pages 1

[58] M. Lobosco, A. F. Silva, O. Loques, and C. L. de Amorim. A New Distributed Java Virtual Machine for Cluster Computing. In *Proc. 9th Intl. Euro-Par Conf. (Euro-Par'03), Lecture Notes in Computer Science vol. 2790*, pages 1207–1215, Klagenfurt, Austria, 2003. pages 20

[59] R. K. K. Ma, C. L. Wang, and F. C. M. Lau. M-JavaMPI: a Java-MPI Binding with Process Migration Support. In *Proc. 2nd IEEE/ACM Intl. Symposium on Cluster Computing and the Grid (CCGrid'02)*, pages 255–262, Berlin, Germany, 2002. pages 16

[60] J. Maassen, R. V. v. Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman. Efficient Java RMI for Parallel Programming. *ACM Transactions on Programming Languages and Systems*, 23(6):747–775, 2001. pages 12

[61] D. A. Mallón, G. L. Taboada, J. Touriño, and R. Doallo. NPB-MPJ: NAS Parallel Benchmarks Implementation for Message-Passing in Java. In *Proc. 17th Euromicro Intl. Conf. on Parallel, Distributed, and Network-Based Processing (PDP'09)*, pages 181–190, Weimar, Germany, 2009. pages xvi, xvi, 3, 5, 109, 167

[62] P. Martin, L. M. Silva, and J. G. Silva. A Java Interface to WMPI. In *Proc. 5th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'98), Lecture Notes in Computer Science vol. 1497*, pages 121–128, Liverpool, UK, 1998. pages 15

[63] J. A. Mathew, H. A. James, and K. A. Hawick. Development Routes for Message Passing Parallelism in Java. In *Proc. 2th ACM Java Grande Conf. (JAVA'00)*, pages 54–61, San Francisco, CA, USA, 2000. pages 16

[64] Message Passing Interface Forum. http://www.mpi-forum.org [Last visited: May 2009]. pages 13

[65] S. Mintchev and V. Getov. Towards Portable Message Passing in Java: Binding MPI. In *Proc. 4th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'97), Lecture Notes in Computer Science vol. 1332*, pages 135–142, Crakow, Poland, 1997. pages 15

[66] J. E. Moreira, S. P. Midkiff, M. Gupta, P. V. Artigas, M. Snir, and R. D. Lawrence. Java Programming for High-Performance Numerical Computing. *IBM Systems Journal*, 39(1):21–56, 2000. pages 137

[67] S. Morin, I. Koren, and C. M. Krishna. JMPI: Implementing the Message Passing Standard in Java. In *Proc. 4th Intl. Workshop on Java for Parallel and Distributed Computing (IWJPDC'02)*, pages 118–123, Fort Lauderdale, FL, USA, 2002. pages 16, 17, 37

[68] C. A. Moritz and M. Frank. LoGPC: Modeling Network Contention in Message-Passing Programs. *IEEE Transactions on Parallel and Distributed Systems*, 12(4):404–415, 2001. pages 22

[69] MPJ Express Project. http://mpj-express.org [Last visited: May 2009]. pages 15

[70] Myricom Inc. GM/MX/Myrinet. http://www.myri.com [Last visited: May 2009]. pages 10, 11

[71] NAS Parallel Benchmarks. http://www.nas.nasa.gov/NAS/NPB [Last visited: May 2009]. pages 133, 134, 135

[72] A. Nelisse, J. Maassen, T. Kielmann, and H. E. Bal. CCJ: Object-Based Message Passing and Collective Communication in Java. *Concurrency and Computation: Practice and Experience*, 15(3-5):341–369, 2003. pages 16, 37

[73] R. V. v. Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H. E. Bal. Ibis: a Flexible and Efficient Java-based Grid Programming Environment. *Concurrency and Computation: Practice and Experience*, 17(7-8):1079–1107, 2005. pages 11, 12, 15, 29

[74] Open MPI Project. http://www.open-mpi.org [Last visited: May 2009]. pages 78

[75] S. Petri, L. Schneidenbach, and B. Schnor. Architecture and Implementation of a Socket Interface on top of GAMMA. In *Proc. 28th IEEE Conf. on Local Computer Networks (LCN'03)*, pages 528–536, Bonn, Germany, 2003. pages 11

[76] M. Philippsen, B. Haumacher, and C. Nester. More Efficient Serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495–518, 2000. pages 12, 106

[77] ProActive Parallel Suite Project. http://proactive.inria.fr [Last visited: May 2009]. pages 12, 109, 133

[78] B. Pugh and J. Spacco. MPJava: High-Performance Message Passing in Java using Java.nio. In *Proc. 16th Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC'03), Lecture Notes in Computer Science vol. 2958*, pages 323–339, College Station, TX, USA, 2003. pages 16, 136

[79] S. H. Rodrigues, T. E. Anderson, and D. E. Culler. High-Performance Local-Area Communication with Fast Sockets. In *Proc. Winter 1997 USENIX Symposium*, pages 257–274, Anaheim, CA, USA, 1997. pages 11

[80] F. Seifert and H. Kohmann. SCI SOCKET - a Fast Socket Implementation over SCI. http://www.dolphinics.com/userfiles/files/Whitepaper/sci-socket.-pdf [Last visited: May 2009]. pages 11

[81] A. Shafi, B. Carpenter, and M. Baker. Nested Parallelism for Multi-core HPC Systems using Java. *Journal of Parallel and Distributed Computing*, 2009 (In press). pages 15, 29, 71, 77

[82] A. Shafi, B. Carpenter, M. Baker, and A. Hussain. A Comparative Study of Java and C Performance in two Large-scale Parallel Applications. *Concurrency and Computation: Practice and Experience*, In press, 2009. pages 1

[83] A. Shafi and J. Manzoor. Towards Efficient Shared Memory Communications in MPJ Express. In *Proc. 11th Intl. Workshop on Java and Components for Parallelism, Distribution and Concurrency (IWJacPDC'09), Rome, Italy*, page 111b (8 pages), 2009. pages 79

[84] L. A. Smith, J. M. Bull, and J. Obdržálek. A Parallel Java Grande Benchmark Suite. In *Proc. 14th ACM/IEEE Conf. on Supercomputing (SC'01)*, page 8 (10 pages), Denver, CO, USA, 2001. pages 4, 41

[85] Stream Control Transmission Protocol (SCTP). http://www.sctp.org [Last visited: May 2009]. pages 166

[86] G. L. Taboada, C. Teijeiro, and J. Touriño. High Performance Java Remote Method Invocation for Parallel Computing on Clusters. In *Proc. 12th IEEE*

Symposium on Computers and Communications (ISCC'07), pages 233–239, Aveiro, Portugal, 2007. pages XVII, 4, 167

[87] G. L. Taboada, J. Touriño, and R. Doallo. Java Message-Passing Micro-Benchmark Suite. http://www.des.udc.es/˜gltaboada/micro-bench/ [Last visited: May 2009]. pages XVI, 3, 21, 26, 124

[88] G. L. Taboada, J. Touriño, and R. Doallo. Performance Analysis of Java Message-Passing Libraries on Fast Ethernet, Myrinet and SCI Clusters. In *Proc. 5th IEEE Intl. Conf. on Cluster Computing (CLUSTER'03)*, pages 118–126, Hong Kong, China, 2003. pages 3, 10, 15, 24, 37, 167

[89] G. L. Taboada, J. Touriño, and R. Doallo. Designing Efficient Java Communications on Clusters. In *Proc. 7th Intl. Workshop on Java for Parallel and Distributed Computing (IWJPDC'05)*, page 182a (8 pages), Denver, CO, USA, 2005. pages 3, 167

[90] G. L. Taboada, J. Touriño, and R. Doallo. Efficient Java Communication Protocols on High-speed Cluster Interconnects. In *Proc. 31st IEEE Conf. on Local Computer Networks (LCN'06)*, pages 264–271, Tampa, FL, USA, 2006. pages 3, 167

[91] G. L. Taboada, J. Touriño, and R. Doallo. High Performance Java Sockets for Parallel Computing on Clusters. In *Proc. 9th Intl. Workshop on Java and Components for Parallelism, Distribution and Concurrency (IWJacPDC'07)*, page 197b (8 pages), Long Beach, CA, USA, 2007. pages 74, 167

[92] G. L. Taboada, J. Touriño, and R. Doallo. Java Fast Sockets: Enabling High-speed Java Communications on High Performance Clusters. *Computer Communications*, 31(17):4049–4059, 2008. pages XVI, 3, 167

[93] G. L. Taboada, J. Touriño, and R. Doallo. F-MPJ: Scalable Java Message-passing Communications on Parallel Systems. *Journal of Supercomputing*, 2009 (In press). pages XVI, XVII, 4, 167

[94] G. L. Taboada, J. Touriño, and R. Doallo. Performance Analysis of Message-Passing Libraries on High-Speed Clusters. *Intl. Journal of Computer Systems Science & Engineering*, 2009 (In press). pages XVI, XVI, XVI, 3, 167

[95] G. L. Taboada, J. Touriño, and R. Doallo. Performance Modeling and Evaluation of Java Message-Passing Primitives on a Cluster. In *Proc. 10th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'03), Lecture Notes in Computer Science vol. 2840*, pages 29–36, Venice, Italy, 2003. pages 166

[96] G. L. Taboada, J. Touriño, and R. Doallo. Non-blocking Java Communications Support on Clusters. In *Proc. 13th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'06), Lecture Notes in Computer Science vol. 4192*, pages 29–36, Bonn, Germany, 2006. pages 4, 167

[97] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of Collective Communication Operations in MPICH. *Intl. Journal of High Performance Computing Applications*, 19(1):49–66, 2005. pages 40, 118

[98] D. Thurman. jPVM – A Native Methods Interface to PVM for the Java Platform. http://web.archive.org/web/20031206085325/www.chmsr.gatech.edu/-jPVM/, 1998. [Last visited: May 2009]. pages 17

[99] TOP500 Supercomputing Site. http://www.top500.org [Last visited: May 2009]. pages 91

[100] J. Touriño and R. Doallo. Characterization of Message-Passing Overhead on the AP3000 Multicomputer. In *Proc. 30th Intl. Conf. on Parallel Processing (ICPP'01)*, pages 321–328, Valencia, Spain, 2001. pages 23, 24

[101] Y. Touyama and S. Horiguchi. Performance Evaluation of Practical Parallel Computation Model LogPQ. In *Proc. IEEE Intl. Symposium on Parallel Architectures, Algorithms and Networks (ISPAN'99)*, pages 215 – 221, Fremantle, Australia, 1999. pages 22

[102] D. Turner and X. Chen. Protocol-Dependent Message-Passing Performance on Linux Clusters. In *Proc. 4th IEEE Intl. Conf. on Cluster Computing (CLUSTER'02)*, pages 187–194, Chicago, IL, USA, 2002. pages 4, 60, 106

[103] S. S. Vadhiyar, G. E. Fagg, and J. J. Dongarra. Towards an Accurate Model for Collective Communications. *Intl. Journal of High Performance Computing Applications*, 18(1):159–167, 2004. pages 41, 118

[104] K. Verstoep, R. Bhoedjang, T. Rühl, H. Bal, and R. Hofman. Cluster Communication Protocols for Parallel-programming Systems. *ACM Transactions on Computer Systems*, 22(3):281–325, 2004. pages 1

[105] B. A. Vianna, A. A. Fonseca, N. T. Moura, L. T. Mendes, J. A. Silva, C. Boeres, and V. E. F. Rebello. A Tool for the Design and Evaluation of Hybrid Scheduling Algorithms for Computational Grids. In *Proc. 2nd ACM Workshop on Middleware for Grid Computing (MGC'04)*, pages 41 – 46, Toronto, Canada, 2004. pages 22

[106] M. Welsh. NBIO: Nonblocking I/O for Java. http://www.eecs.harvard.edu/~mdw/proj/java-nbio [Last visited: May 2009]. pages 11

[107] M. Welsh and D. E. Culler. Jaguar: Enabling Efficient Communication and I/O in Java. *Concurrency: Practice and Experience*, 12(7):519–538, 2000. pages 20

[108] N. Yalamanchilli and W. Cohen. Communication Performance of Java-Based Parallel Virtual Machines. *Concurrency: Practice and Experience*, 10(11-13):1189–1196, 1998. pages 17

[109] B.-Y. Zhang, G.-W. Yang, and W.-M. Zheng. Jcluster: an Efficient Java Parallel Environment on a Large-scale Heterogeneous Cluster. *Concurrency and Computation: Practice and Experience*, 18(12):1541–1557, 2006. pages 16

[110] X. Zhang, S. McIntosh, P. Rohatgi, and J. L. Griffin. XenSocket: A High-throughput Interdomain Transport for VMs. In *Proc. 8th ACM/IFIP/USENIX Intl. Middleware Conf. (Middleware'07), Lecture Notes in Computer Science vol. 4834*, pages 184–203, Newport Beach, CA, USA, November 2007. pages 11

[111] W. Zhu, C.-L. Wang, and F. C. M. Lau. JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support. In *Proc. 4th IEEE Intl. Conf. on Cluster Computing (CLUSTER'02)*, pages 381–388, Chicago, IL, USA, 2002. pages 20